Sparse arrays and multivariate polynomials in R

Robin K. S. Hankin
Auckland University of Technology

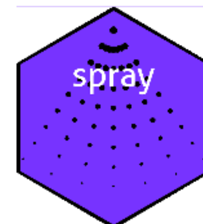
Abstract

In this short article I introduce the **spray** package, which provides some functionality for handling sparse arrays. The package uses the C++ Standard Template Library's `map` class (Musser, Derge, and Saini 2009) to store and retrieve elements. One natural application for sparse arrays is multivariate polynomials and I give two examples of the package in use, one drawn from the fields of random walks on lattices and one from the field of recreational combinatorics.

To cite the **spray** package, use Hankin (2022c).

Keywords: Multivariate polynomials, R, sparse arrays.

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. Some of the underlying design philosophy is discussed in the appendix. Here, 'sparse multinomial' is defined as one whose array representation is sufficiently sparse to make taking advantage of is sparseness worthwhile. However, other definitions of sparseness may be useful and I outline some below.

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 **rSympy** package does not, as of 2017, implement sparse 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**.

(The `mvp` package (Hankin 2022b) is now available; this uses a more powerful concept of sparsity).

2. Sparse arrays

Base R has extensive support for multidimensional arrays. Consider

```
R> a <- array(1, dim=2:5)
```

The resulting object requires storage of $2 \times 3 \times 4 \times 5 = 120$ floating point numbers, which are represented in an elegant format amenable to Cartesian extraction and replacement. However, arrays in which many of the elements are zero are common and in this case storing only the nonzero elements and their positions would be a more compact and efficient representation. To create a sparse array object in the **spray** package, one specifies a matrix of indices `M` with each row corresponding to the position of a nonzero element, and a numeric vector of values:

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

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

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

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

Thus `S1[0,0,2] = 2`. 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. This is because the STL map class does not preserve the orders of its elements. This does not matter, as the order in which the elements are stored is immaterial in the use-cases presented here.

Extract and replace methods require the index to be a matrix[1]:

[1] Indexing with a vector (interpreted as a row of the index matrix) is problematic. The package requires idiom such as `S[1,2,3]` and `S[1,1:3,3]` to work as expected; and because `[.spray()` and `[<-.spray()` dispatch on the first argument, the package does not attempt to guess what the user intended.

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

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

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

```
R> M2 <- matrix(c(
+      6, -7,  8,
+      0,  0,  2,
+      1,  1,  3), byrow=TRUE, ncol=3)
R> S2 <- spray(M2, c(17, 11 ,-4))
R> S2

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

R> S1 <- S1 + S2
R> S1

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

Thus element `[0,0,2]` becomes $2+11 = 13$, while element `[1,1,3]` cancels and thus vanishes. There is no requirement for indices to be positive: Element `[6,-7,8]` is new (with value 17). Even though the representation of the spray object does not preserve the order of the index rows in the argument, a particular index row is associated unambiguously with a unique numeric value. The package uses **disordR** discipline, discussed below in section 4.

3. The spray package and multivariate polynomials

One natural application for `spray` objects is multivariate polynomials (Hankin 2008). I will first discuss the univariate case and then progress to multivariate polynomials.

3.1. Univariate polynomials

Univariate polynomials are a good place to start. Suppose the polynomial

$$A = 1 + 2x^3 + 6x^8$$

were to be represented using R objects. One natural approach, taken in the **polynomial** package (Venables, Hornik, and Maechler 2016), is to store the coefficients in a vector:

```
R> library("polynom")
R> A <- polynomial(c(1, 0, 0, 2, 0, 0, 0, 0, 6))
R> dput(A)

structure(c(1, 0, 0, 2, 0, 0, 0, 0, 6), class = "polynomial")

R> A

1 + 2*x^3 + 6*x^8
```

But again see how the R object thus created stores zero elements, which can be problematic if the polynomial in question is large degree and sparse. Similar issues arise in the case of multivariate polynomials. The **multipol** package (Hankin 2008) uses a similar methodology, storing coefficients as an arbitrary-dimensional array. However, as noted above, this often leads to inefficient computation.

3.2. Multivariate polynomials

One natural and useful interpretation of a sparse array is as a multivariate polynomial. Consider the following sparse array:

```
R> S3 <- spray(matrix(c(0,0,0, 0,0,1, 1,1,1, 3,0,0), byrow=TRUE, ncol=3), 1:4)
R> S3

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

It is natural to interpret the rows of the index matrix as powers of different variables of a multivariate polynomial, and the values as being the coefficients. This is realized in the package using the `polyform` option, which if set to `TRUE`, modifies the print method:

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

-3*z -3*y +13*z^2 -3*x +17*x^6*y^-7*z^8
```

(only the print method has changed; `S1` is as before). The print method interprets, by default, the three columns as variables x, y, z although this behaviour is user-definable. With this interpretation, multiplication and addition have natural definitions as multivariate polynomial multiplication and addition:

```
R> S1 + S2

-3*z -3*y +24*z^2 -4*x*y*z^3 -3*x +34*x^6*y^-7*z^8

R> S1 * S2

+12*x^2*y*z^3 -51*x^7*y^-7*z^8 +289*x^12*y^-14*z^16
-68*x^7*y^-6*z^11 -33*y*z^2 -52*x*y*z^5 +143*z^4 +12*x*y*z^4
+12*x*y^2*z^3 +408*x^6*y^-7*z^10 -51*x^6*y^-6*z^8 -33*x*z^2
-33*z^3 -51*x^6*y^-7*z^9

R> S1^2

+442*x^6*y^-7*z^10 +9*y^2 +18*y*z +9*z^2 +18*x*y -78*z^3
+18*x*z +169*z^4 +9*x^2 -102*x^6*y^-7*z^9 -78*y*z^2
-102*x^6*y^-6*z^8 -78*x*z^2 -102*x^7*y^-7*z^8
+289*x^12*y^-14*z^16
```

It is possible to introduce an element of symbolic calculation, exhibiting familiar algebraic identities. Consider the `lone()` function, which creates a sparse array whose multivariate polynomial interpretation is a single variable:

```
R> x <- lone(1, 3)
R> y <- lone(2, 3)
R> z <- lone(3, 3)
R> options(polyform = FALSE)
R> list(x, y, z)

[[1]]
          val
 1 0 0  =   1

[[2]]
          val
 0 1 0  =   1

[[3]]
          val
 0 0 1  =   1

R> options(polyform = TRUE)
R> (1 + x + y)^3
```

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

R> (x + y) * (y + z) * (x + z) - (x + y + z) * (x*y + x*z + y*z)

-x*y*z

R> (x + y) * (x - y) - (x^2 - y^2)

the NULL multinomial of arity 3
```

3.3. The null polynomial and arity issues

The package is intended to provide functionality for sparse arrays, one interpretation of which is multivariate polynomials. The package, implementing sparse arrays, forbids the addition of two sparse arrays with different dimensionalities:

```
R>  lone(1, 2) + lone(1, 1)

Error: arity(S1) == arity(S2) is not TRUE
```

One problematic object is the empty array. Zero multinomials are represented with a zero-row index matrix and zero-length numeric vector of values. Because a spray object is a sparse array, a zero multinomial must have a specific arity[2].

3.4. Algebraic identities

Similar but more involved techniques can be used to prove Euler's four-square identity and Degen's eight-square identity, given in the package's test suite. However, it should be noted that the **mpoly** package has a more natural idiom and does not suffer from the visual defect of arbitrary term ordering.

3.5. Further functionality

Multivariate polynomials have a natural interpretation as functions:

```
R> (S4 <- spray(cbind(1:3, 3:1), 1:3))

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

R> f <- as.function(S4)
R> f(c(1, 2))

 X
22
```

[2]This philosophy is different from earlier versions of the software which treated the empty array as the zero multinomial, with which addition and multiplication were defined algebraically. I would like to thank an anonymous *R Journal* referee for this insight.

The last line showing the result of substituting $x = 1, y = 2$ into S4. Other algebraic operations include substitution and partial differentiation. Consider the homogeneous polynomial in three variables:

```
R> (S5 <- 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
```

Interpreting S5 as a multivariate polynomial with variables x, y, z we may substitute $y = 5$ using the `subs()` function:

```
R> subs(S5, 2, 5)

125 +5*x^2 +x^3 +y^3 +x^2*y +25*x +5*x*y +25*y +x*y^2 +5*y^2
```

Differentiation is also straightforward. Suppose we wish to calculate the multivariate polynomial corresponding to

$$\frac{\partial^6}{\partial x\, \partial^2 y\, \partial^3 z}\left(xyz + x + 2y + 3z\right)^3$$

This would be

```
R> aderiv((xyz(3) + linear(1:3))^3, 1:3)

+216*x +108*x^2*y
```

4. Manipulation of coefficients

The **spray** package uses **disordR** discipline (Hankin 2022a) for manipulation of the coefficients. Thus:

```
R> set.seed(0)
R> (A <- rspray())

6 +10*x*y*z +2*y^2*z^2 +4*z^2 +3*x^2*y^2 +5*x +15*x^2*y

R> coeffs(A)

A disord object with hash b49f3c545bd4a0385b46c9ddd7a56399bce51692 and elements
[1] 10  2  6  4  3  5 15
(in some order)
```

We see above that the coefficients of object `A` are not an ordinary R vector but a `disord` object. Because the coefficients are stored in an implementation-specific order, we cannot access or manipulate `coeffs(a)[2]`, for example. But some operations are allowed; we may consider the coefficients modulo 3:

```
R> coeffs(A) <- coeffs(A) %% 3
R> A

+x*y*z +2*y^2*z^2 +z^2 +2*x
```

A detailed motivation and use-case for **spray** is provided by Hankin (2022a).

5. The package in use: some examples

Multivariate polynomials are useful and efficient structures in a variety of applications. Here I give two examples of the package in use: One drawn from the field of random walks on lattices, and one from recreational combinatorics.

5.1. Random walks on lattices

Random walks on periodic lattices find application in a wide range of applied mathematics including the study of molecular and ionic crystals (den Hollander and Kasteleyn 1982), polymers (Scheunders and Naudts 1989) and photosynthetic units (Montroll 1969). The basic idea is that some entity (exciton, ion, etc) has a well-defined position on a periodic lattice $(\mathbb{Z}/n\mathbb{Z})^d$; it then moves on the lattice, performing a random walk. The examples here have $d = 2$ but extension to arbitrary dimensions is immediate.

The periodic lattice itself may be identified with a multivariate polynomial in d variables, here x and y; the probability of the entity being at point (n, m) is the coefficient of $x^n y^m$.

The entity typically moves between adjacent nodes according to a *kernel* polynomial whose coefficients are the probabilities of the moves. Periodicity may be enforced simply by wrapping the polynomial using modular arithmetic.

In many cases, entities are not necessarily conserved: the entity may decay (usually with a fixed probability per timestep), or be annihilated when it encounters a particular node in the lattice (a ‘trap’ (Scheunders and Naudts 1989), corresponding to the formation of sugar in the cell). Here, we work on a 17×17 lattice as a large but computationally tractable domain.

All these processes have natural and efficient R idiom in the **spray** package. We may specify a kernel allowing movement to adjacent nodes, or to stay in the same place with equal probability:

```
R> d <- 2
R> kernel <- spray(rbind(0, diag(d), -diag(d)))/(1 + 2*d)
```

At the first timestep, the the entity is at, say, point $(10, 10)$ with probability 1:

```
R> initial <- spray(rep(10, d))
```

Finding the probability mass function of the entity after, say 14 timesteps, is straightforward:

```
R> t14 <- initial * kernel^14
```

Traps may be assigned using standard indexing and we will work with a 17×17 array:

```
R> traps <- matrix(c(2, 3, 3, 5), 2, 2)
R> n <- 17
```

Then we may calculate the evolution of the probability mass function as follows:

```
R> timestep <- function(state, kernel, traps){
+    state <- state * kernel
+    state <- spray(index(state)%%n, coeffs(state), addrepeats = TRUE)
+    state[traps] <- 0
+    return(state)
+  }
```

In function `timestep()`, the first line uses standard multivariate polynomial multiplication to advance the state of the entity; the second enforces periodic boundary conditions, and the third implements the traps' annihilation of the entity. The probability of the entity still existing after 100 timesteps is then:

```
R> state <- initial
R> for(i in 1:100){state <- timestep(state, kernel, traps)}
R> sum(coeffs(state))

[1] 0.9006642
```

Note the streamlined R idiom: It is not clear how such manipulations could be performed using the **mpoly** or the **multipol** packages.

5.2. Recreational combinatorics

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.

On an infinite chessboard, we might define the multivariate generating polynomial[3] k for a knight as

$$k = x^2y + x^2y^{-1} + x^{-2}y + x^{-2}y^{-1} + xy^2 + xy^{-2} + x^{-1}y^2 + x^{-1}y^{-2}$$

where we have identified powers of x with squares moved horizontally (counted algebraically, negative powers mean move to the left), and powers of y with squares moved vertically. Then

[3] Standard terminology, although it might be more accurately referred to as a multivariate Laurent polynomial.

the coefficient of $x^a y^b$ in k is the number of ways of moving from the origin [that is, $x^0 y^0$] to square (a, b). Similarly, k^n is the generating function for a knight which makes n moves: The coefficient of $x^a y^b$ in k^n is the number of ways of moving from the origin to square (a, b).

The R idiom for this is straightforward; we define `chess_knight`, a spray object with rows corresponding to the possible moves the chess piece may make:

```
R> 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))
R> options(polyform = FALSE)
R> 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
```

```
R> options(polyform = TRUE)
R> chess_knight
```

```
+x*y^2 +x*y^-2 +x^-1*y^2 +x^-1*y^-2 +x^2*y +x^2*y^-1 +x^-2*y
+x^-2*y^-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:

```
R> 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:

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

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

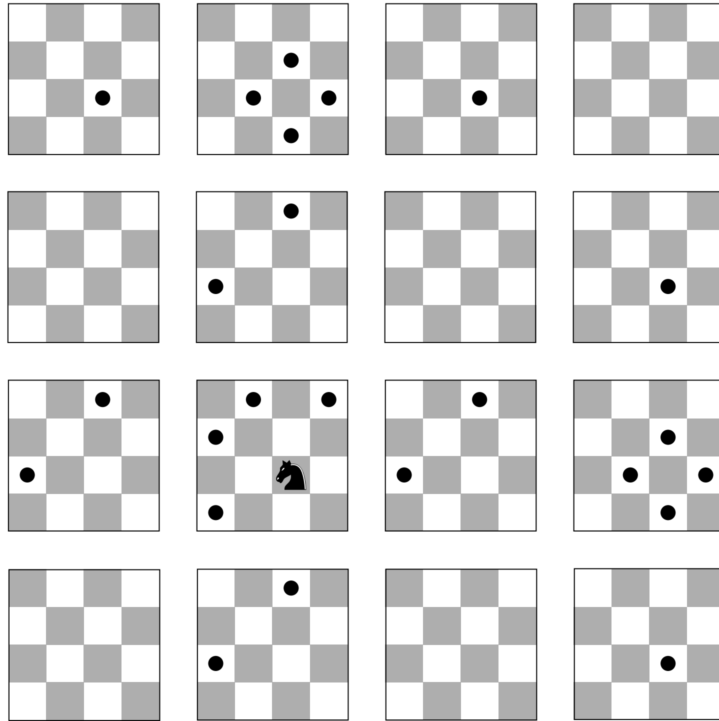


Figure 1: Four-dimensional knight on a $4 \times 4 \times 4 \times 4$ board. Cells attacked by the knight shown by dots

```
R> 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 0.4 seconds of elapsed time whereas the same[4] calculation took over 173 seconds using **mpoly**.

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:

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

```
[1] 10306561
```

(0.6 seconds for **spray** vs 275 seconds for **mpoly**). For 8 moves, the differences are more pronounced, with **spray** taking 4.0 seconds and **mpoly** requiring more than 1500 seconds).

[4]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.

6. Conclusions and further work

The **spray** package provides functionality for sparse, arbitrarily-dimensioned arrays. One natural interpretation of a sparse array is as a multivariate polynomial and the package leverages the `map` class of C++ to give fast polynomial multiplication.

The functionality provided overlaps with that of **multipol** and **mpoly**. The **multipol** package is too slow to be of practical value for any but the smallest illustrative objects.

The different name-based philosophy employed by the **mpoly** package is certainly an advantage in terms of natural R idiom, although there is a performance penalty. There are also occasional applications of multivariate polynomials (such as random walks on lattices) in which the structure of **spray** is a conceptual advantage.

British mathematician J. H. Wilkinson famously defined a matrix to be "sparse" if it has enough zeros that it pays to take advantage of them; this definition applies to the arrays considered here. However, other definitions of sparsity are possible. Consider the following example, taken from Kahle (2013):

$$ab^2 + bc^2 + cd^2 + \cdots + yz^2 + za^2$$

.

The **spray** idiom for such an expression is

```
R> a <- diag(26)
R> options(sprayvars = letters)
R> a[1 + cbind(0:25, 1:26) %% 26] <- 2
R> spray(a)
```

```
+r*s^2 +c*d^2 +f*g^2 +e*f^2 +d*e^2 +g*h^2 +i*j^2 +h*i^2 +j*k^2
+y*z^2 +k*l^2 +b*c^2 +s*t^2 +l*m^2 +m*n^2 +w*x^2 +n*o^2 +a*b^2
+v*w^2 +p*q^2 +q*r^2 +t*u^2 +o*p^2 +u*v^2 +x*y^2 +a^2*z
```

—but it is clear that the index matrix has a large degree of sparseness which **mpoly** takes advantage of and `spray` does not. Further work might include the development of name-based multivariate polynomials using concepts from STL to provide the best of both worlds (and indeed the **mvp** package (Hankin 2022b) does just this, and more).

References

den Hollander WTF, Kasteleyn PW (1982). "Random walks with 'spontaneous emission' on lattices with periodically distributed imperfect traps." *Physica A*, **112A**, 523–543.

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

Hankin RKS (2022a). "Disordered vectors in R: introducing the **disordR** package." `doi:10.48550/ARXIV.2210.03856`.

Hankin RKS (2022b). "Fast multivariate polynomials in R: the **mvp** package." doi:10.48550/ARXIV.2210.15991.

Hankin RKS (2022c). "Sparse arrays in R: the **spray** package." doi:10.48550/ARXIV.2210.10848.

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 https://journal.r-project.org/archive/2013-1/kahle.pdf.

Montroll EW (1969). "Random walks on lattices II. Calculation of first-passage times with applications to exciton trapping on photosynthetic units." *Journal of Mathematical Physics*, **10**(4), 753–765.

Musser DR, Derge GJ, Saini A (2009). *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. 3rd edition. Addison-Wesley Professional. ISBN 0321702123, 9780321702128.

Scheunders P, Naudts J (1989). "Random walks on lattices with a random distribution of perfect traps." *Condensed Matter*, **73**, 551–553.

Venables B, Hornik K, Maechler M (2016). ***polynom**: A Collection of Functions to Implement a Class for Univariate Polynomial Manipulations*. R package version 1.3-9. S original by Bill Venables, packages for R by Kurt Hornik and Martin Maechler., URL https://CRAN.R-project.org/package=polynom.

A. 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 (Musser *et al.* 2009) 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.

In the STL, a map object stores keys and associated values in whatever order the software considers to be most propitious. This allows faster access and modification times but the order in which the maps are stored is implementation specific. In the case of sparse arrays, this is not an issue because the nonzero entries do not possess a natural order, unlike dense arrays in which lexicographic ordering is used. For multivariate polynomials, the order of storage is not important algebraically because addition is commutative and associative. These issues are accommodated using **disordR** discipline which forbids implementation-specific idiom (Hankin 2022a).

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.

Affiliation:

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