Journal of Statistical Software

September 2022, Volume 104, Issue 5. *doi: 10.18637/jss.v104.i05*

calculus: High-Dimensional Numerical and Symbolic Calculus in R

Emanuele Guidotti
University of Neuchâtel
CREST, Japan Science and Technology Agency

Abstract

The R package **calculus** implements C++-optimized functions for numerical and symbolic calculus, such as the Einstein summing convention, fast computation of the Levi-Civita symbol and generalized Kronecker delta, Taylor series expansion, multivariate Hermite polynomials, high-order derivatives, ordinary differential equations, differential operators and numerical integration in arbitrary orthogonal coordinate systems. The library applies numerical methods when working with functions, or symbolic programming when working with characters or expressions. The package handles multivariate numerical calculus in arbitrary dimensions and coordinates. It implements the symbolic counterpart of the numerical methods whenever possible, without depending on external computer algebra systems. Except for **Rcpp**, the package has no strict dependencies in order to provide a stable self-contained toolbox that invites re-use.

Keywords: symbolic programming, finite difference, differential operators, numerical integration, coordinate systems, Einstein summation, Taylor series, Hermite polynomials, R.

1. Introduction

Multivariate calculus underlies a wide range of applications in the natural and social sciences. In statistics, asymptotic expansion formulas for stochastic processes (Yoshida 1992) can be obtained by solving high dimensional systems of ordinary differential equations. The transition density of multivariate diffusions can be approximated using Hermite polynomials (Aït-Sahalia 2002) or Taylor-like expansions (Li *et al.* 2013). Advances in medical imaging technology as well as telecommunication data-collection have ushered in massive datasets that make multidimensional data more commonplace (Li, Bien, and Wells 2018) and tensors – multidimensional arrays – have recently become ubiquitous in signal and data analytics at the confluence of signal processing, statistics, data mining, and machine learning (Sidiropoulos,

De Lathauwer, Fu, Huang, Papalexakis, and Faloutsos 2017). In Earth sciences, cartography, quantum mechanics, relativity, and engineering, non-Cartesian coordinates are often chosen to match the symmetry of the problem in two, three and higher dimensions.

R (R Core Team 2022) has shown to be a viable computing environment for implementing and applying numerical methods (Borchers, Hankin, and Sokol 2022) as a practical tool for applied statistics. However, such methods are seldom flexible enough to handle multivariate calculus in arbitrary dimensions and coordinates. The package **numDeriv** (Gilbert and Varadhan 2019) sets the standard for numerical differentiation in R, providing numerical gradients, Jacobians, and Hessians, but does not support higher order derivatives or differentiation of tensor-valued functions. **tensorA** (Van den Boogaart 2020) implements the Einstein summing convention but does not support arbitrary expressions involving more than two tensors or tensors with repeated indices. **mpoly** (Kahle 2013) implements univariate but not multivariate Hermite polynomials. In a similar way, **pracma** (Borchers 2022) supports the computation of Taylor series for univariate but not multivariate functions. **cubature** (Narasimhan, Johnson, Hahn, Bouvier, and Kiêu 2022) provides an efficient interface for multivariate integration but limited to Cartesian coordinates.

On the other hand, R is not designed for symbolic computing. 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 to be able to do some relatively basic operations and routines with multivariate symbolic calculus (Kahle 2013). Although there exist packages to interface external computer algebra systems, R still lacks a native support that invites re-use. The package **Ryacas** (Andersen and Højsgaard 2019) interfaces the computer algebra system **Yacas** (Pinkus, Winnitzky, and Mazur 2020), while **caracas** (Andersen and Højsgaard 2021) – based on **reticulate** (Ushey, Allaire, and Tang 2022) – accesses the symbolic algebra system **SymPy** (Meurer *et al.* 2017).

This work presents the R package **calculus** for high dimensional numerical and symbolic calculus in R. The contribution is twofold. First, the package handles multivariate numerical calculus in arbitrary dimensions and coordinates via C++ (Stroustrup 2013) optimized functions, improving the state-of-the-art both in terms of flexibility and efficiency. It achieves approximately the same accuracy for numerical differentiation as the **numDeriv** (Gilbert and Varadhan 2019) package but significantly reduces the computational time. It supports higher order derivatives and the differentiation of possibly tensor-valued functions. Differential operators such as the gradient, divergence, curl, and Laplacian are made available in arbitrary orthogonal coordinate systems. The Einstein summing convention supports expressions involving more than two tensors and tensors with repeated indices. Besides being more flexible, the summation proves to be faster than the alternative implementation found in the **tensorA** package (Van den Boogaart 2020) for advanced tensor arithmetic with named indices. Unlike **mpoly** (Kahle 2013) and **pracma** (Borchers 2022), the package supports multidimensional Hermite polynomials and Taylor series of multivariate functions. The package integrates seamlessly with **cubature** (Narasimhan *et al.* 2022) for efficient numerical integration in C and extends the numerical integration to arbitrary orthogonal coordinate systems. Second, the symbolic counterpart of the numerical methods are implemented whenever possible to meet the growing needs for R to handle basic symbolic operations. Although **calculus** is not to be compared with general-purpose symbolic algebra systems, it provides, among others, symbolic high order derivatives of possibly tensor-valued functions, symbolic differential operators in arbitrary orthogonal coordinate systems, symbolic Einstein summing convention,

and Taylor series expansion of multivariate functions. This is done entirely in R, without depending on external software in order to provide a self-contained toolbox that invites re-use.

The remainder of the paper is organized as follows: Section 2 introduces the package and the underlying philosophy, Section 3 presents basic operations that underlie the whole package, Section 4 and 5 provide basic utilities for vector and matrix algebra, Section 6 presents tensor algebra with particular focus on the Einstein summation, Section 7 provides fast and accurate derivatives, Section 8 presents the Taylor series of possibly multivariate functions, Section 9 describes multidimensional Hermite polynomials, Section 10 solves ordinary differential equations, Section 11 and 12 introduce differential operators and integrals in arbitrary orthogonal coordinate systems before Section 13 concludes.

2. The R package calculus

The R package **calculus** implements C++ optimized functions for numerical and symbolic calculus, such as the Einstein summing convention, fast computation of the Levi-Civita symbol and generalized Kronecker delta, Taylor series expansion, multivariate Hermite polynomials, high-order derivatives, ordinary differential equations, differential operators and numerical integration in arbitrary orthogonal coordinate systems.

2.1. Testing

Several unit tests are implemented via the standard framework offered by **testthat** (Wickham 2011) and run via continuous integration on GitHub Actions.

2.2. Dependencies

The package integrates seamlessly with **cubature** (Narasimhan *et al.* 2022) for efficient numerical integration in C. However, except for **Rcpp** (Eddelbuettel and François 2011), the package has no strict dependencies in order to provide a stable self-contained toolbox that invites re-use.

2.3. Installation

The stable release version of **calculus** (Guidotti 2022) is hosted on the Comprehensive R Archive Network (CRAN) at https://CRAN.R-project.org/package=calculus and it can be installed using:

```
R> install.packages("calculus")
```

2.4. Philosophy

The package provides a unified interface to work with mathematical objects in R. The library applies numerical methods when working with functions, or symbolic programming when working with characters or expressions. To describe multidimensional objects, such as vectors, matrices, and tensors, the package uses the class '`array`' regardless of the dimension. This is done to prevent unwanted results due to operations among different classes, such as '`vector`'

for unidimensional objects or 'matrix' for bidimensional objects. Particular attention is given to correctly handle the dimensions of the arrays and differentiate between e.g., a 2×2 matrix and a $2 \times 2 \times 1$ tensor. In other words, the dimensions are not dropped by default as done in base R.

The philosophy of the package is that of providing a consistent way to handle numerical and symbolic calculus, unidimensional and multidimensional objects, and different coordinate systems. To this end, the implementation is designed around three main concepts. First, the functions should support both numerical and symbolic inputs, whenever possible, and return the corresponding numerical or symbolic output. Second, vectors and matrices are seen as special cases of generic tensors, so that operations between vectors, matrices, and tensors should be written and implemented in generic Einstein notation. Third, the user should be able to specify and use arbitrary orthogonal coordinate systems.

These principles translate in a set of implementation choices. First, the package implements C++ templates that operate with generic types whenever needed. This makes easy for the corresponding R wrappers to work both with numerical and symbolic calculations. In some cases, the R functions behave differently depending on the data type to improve performance. For instance, the function `mx` implements the matrix product for numerical and symbolic matrices. If a numerical matrix is provided, then the function is basically a simple wrapper for the matrix product available in base R. If a symbolic matrix is provided, the function computes the symbolic matrix product in C++ via Einstein notation. In both cases, the interface provided to the end user is the same. The second principle gives a central role to the Einstein notation. The notation is at the core of many standard operations among vectors, matrices, and tensors, and it is of particular usefulness in supporting the implementation of the finite difference scheme for high-order derivatives as well as differential operators. The third principle guides the implementation of the differential operators and numerical integration towards the adoption of scale factors, in that they provide a unified way to represent arbitrary orthogonal coordinates systems. Therefore, no operation is hard coded for a particular coordinate system, but it is rather coded in terms of generic scale factors that the end user will be able to define arbitrary with maximum flexibility.

Finally, the package does not define additional classes as it aims at integrating seamlessly with base R and in particular with the class 'array'. In the same way, the package avoids depending on external computer algebra systems in order to provide a self-contained toolbox that invites re-use.

2.5. Intended use

This package is not designed for didactic purposes, nor it is intended to offer a feature-complete computer algebra system. The reader may refer to the package **mosaic** (Pruim, Kaplan, and Horton 2017) to teach calculus in R, and to **Ryacas** (Andersen and Højsgaard 2019), or **caracas** (Andersen and Højsgaard 2021), to access general-purpose computer algebra systems within R.

This package is conceived for academic research. It is intended to be used as a low-level toolbox to implement academic papers and novel methodologies in R.

As an example, **calculus** is used by the **difNLR** package (Hladka and Martinkova 2020) for detection of so-called differential item functioning, a situation when respondents from different groups but the same overall ability (or other latent trait) have different probability to answer

correctly to an item (or to endorse an item) in multi-item measurement (Drabinova and Martinkova 2017). **difNLR** implements nonlinear regression models to detect between-group differences in item characteristic curves. Function `hessian` of the package **calculus** is used for calculation of sandwich estimator for covariance matrix, which is then used to obtain more precise standard errors and confidence intervals of item parameters.

Another use case is the implementation of asymptotic expansion formulas for diffusion processes in **yuima** (Brouste *et al.* 2014). Here it is possible to expand the characteristic function of arbitrary diffusions by solving a system of thousands of ordinary differential equations. The high-dimensional system needs to be generated symbolically by recursive differentiation of the drift and diffusion terms. The solution of the system is then used to generate a representation of the transition density in terms of multivariate Hermite polynomials. **calculus** is used to generate the symbolic system, to solve it numerically, and to produce the corresponding Hermite polynomials.

2.6. Contributing

All the code is open source and the development version of the package is hosted on GitHub at `https://github.com/eguidotti/calculus`. Contributions are welcome both in terms of bug reports and feature enhancements, via the standard mechanism of GitHub issues and pull requests.

3. Basic operations

3.1. Arithmetic

Basic arithmetic is supported for arrays of the same dimensions with the following data types: `numeric`, `complex`, `character`, `expression`. Automatic type conversion is supported and string manipulation is performed in C++ to improve performance. Below a unidimensional example on the sum, difference, product, and division among the different data types.

```
R> ("a + b" %prod% 1i) %sum% (0 %prod% "c") %diff% (expression(d + e) %div% 3)

[1] "((a + b) * (0+1i)) - ((d + e) / 3)"
```

A minimal simplification algorithm is included to simplify operations involving zeros. No other simplification rule is implemented, as such rules typically come at the cost of longer computational times. The user may consider applying simplification engines on top of **calculus** when this is needed. The same user may refer to the function `Simplify` in the **Deriv** package (Clausen and Sokol 2021) for a dependency-free simplification engine in R, or e.g., **caracas** to access a feature-complete computer algebra system. However, there exist cases, such as recursive calculations, where the simplification of intermediate results leads to significant speed gains. In this cases, **calculus** does implement ad-hoc simplification schemes. See e.g., Section 9 for the calculation of multivariate Hermite polynomials.

3.2. Evaluation

To evaluate a symbolic result in base R, the typical approach is to convert a `character` into an `expression` and to evaluate the `expression`:

```
R> eval(parse(text = "a" %sum% "a"), list(a = 1))

[1] 2
```

However, the standard `eval` in base R only supports the evaluation of one single element. For an expression vector this is the result of evaluating the last element. To simplify the evaluation of symbolic objects, the package **calculus** implements the function `evaluate`. The function takes care of the type conversion, evaluates all the elements, and reshapes the output in order to preserve the dimensions of the input object.

```
R> x <- array(letters[1:6], dim = c(2, 3))
R> evaluate(x, var = c(a = 1, b = 2, c = 3, d = 4, e = 5, f = 6))

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

To evaluate the object multiple times at once, a `data.frame` can be used instead of a named vector for the argument `var`. In this case, the return is a matrix with columns corresponding to the (coalesced) entries of the input and rows corresponding to the rows of `var`.

```
R> x <- array(letters[1:6], dim = c(2, 3))
R> var <- data.frame(a = 1:2, b = 2:3, c = 3:4, d = 4:5, e = 5:6, f = 6:7)
R> evaluate(x, var = var)

     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    2    3    4    5    6
[2,]    2    3    4    5    6    7
```

As the function is vectorized, this allows for fast evaluation of the object at multiple points. See e.g., Section 5.1 for a speed test.

3.3. Options

When performing symbolic operations, the input is automatically sanitized by wrapping all its elements in parentheses, e.g., `"a"` becomes `"(a)"`. This is done to prevent unwanted results, e.g., $a + b \cdot c + d$ instead of $(a + b) \cdot (c + d)$. To disable this behavior, the user can set `options(calculus.auto.wrap = FALSE)`. This is a global option that affects all the functions in the package. Disabling the option may lead to small speed gains and avoid redundant parentheses, but the user would need to make sure that the input is properly sanitized. For instance, if a matrix contains the element `"a + b"`, this should be converted to `"(a + b)"` before applying a matrix multiplication or a generic Einstein summation, while there would be no need to convert `"a * b"` into `"(a * b)"`.

4. Vector algebra

A vector can be regarded as a 1-dimensional tensor. In R, it can be regarded as a 1-dimensional `array` so that the methods presented for multidimensional tensors in Section 6 are available for vectors. The package also implements a few vector-specific utilities, such as the cross product.

4.1. Cross product

The cross product or vector product is an operation on $n-1$ vectors in n-dimensional space. The results is a n-dimensional vector that is perpendicular to the $n-1$ vectors. For example in $\mathbb{R}^3$:

```
R> cross(c(1, 0, 0), c(0, 1, 0))

[1] 0 0 1
```

And in $\mathbb{R}^4$:

```
R> cross(c(1, 0, 0, 0), c(0, 1, 0, 0), c(0, 0, 0, 1))

[1] 0 0 1 0
```

The implementation for numerical vectors is based on matrix determinants as in **pracma** (Borchers 2022). Consistently with the philosophy of the package, the same interface is provided for `character` vectors, where determinants are computed symbolically (Section 5.1).

```
R> cross(c("a", "b", "c"), c("d", "e", "f"))

[1] "(b*(f) + -e*(c)) * 1"  "(a*(f) + -d*(c)) * -1" "(a*(e) + -d*(b)) * 1"
```

5. Matrix algebra

A matrix can be regarded as a 2-dimensional tensor. In R, it can be regarded as a 2-dimensional `array` so that the methods presented for multidimensional tensors in Section 6 are available for matrices. The package also implements a few matrix-specific utilities, such as the symbolic determinant, inverse, and matrix product.

5.1. Determinant

The function `mxdet` computes the numerical or symbolic determinant of matrices depending on the data type. If the elements of the `matrix` are of type `numeric`, then the determinant is computed via the function `det` available in base R.

```
R> mxdet(matrix(1:4, nrow = 2))

[1] -2
```

If the elements are of type `character`, then the symbolic determinant is computed recursively in C++ via Laplace expansion (Wikipedia 2022j).

```
R> mxdet(matrix(letters[1:4], nrow = 2))
```

```
[1] "a*(d) + -b*(c)"
```

The symbolic determinant offers a significant gain in performance when computing determinants for a large number of matrices. The following test compares the performance of two different approaches to compute the determinant of 2^{16} 4×4-matrices. Method *numeric*: compute the numeric determinant for each matrix. Method *symbolic*: compute the symbolic determinant of a 4×4-matrix and evaluate it for each matrix.

```
R> n <- 4
R> e <- letters[1:n^2]
R> grid <- expand.grid(lapply(1:n^2, function(e) runif(2)))
R> colnames(grid) <- e
R> microbenchmark(
+    "numeric" = {
+      x <- apply(grid, 1, function(e) det(matrix(e, nrow = n)))
+    },
+    "symbolic" = {
+      x <- evaluate(mxdet(matrix(e, nrow = n)), grid)
+    }
+ )
```

```
Unit: milliseconds
     expr      min       lq     mean   median       uq     max neval
  numeric 314.3092 326.0248 345.3065 345.1600 354.4539 411.874   100
 symbolic   2.7434   3.0642   4.3564   3.3349   4.8917  42.609   100
```

5.2. Matrix inverse

The function `mxinv` computes the numerical or symbolic inverse of matrices depending on the data type. If the elements of the `matrix` are of type `numeric`, then the inverse is computed via the function `solve` available in base R.

```
R> mxinv(matrix(1:4, byrow = TRUE, nrow = 2))
```

```
     [,1] [,2]
[1,] -2.0  1.0
[2,]  1.5 -0.5
```

If the elements are of type `character`, then the symbolic inverse is computed based on the determinants in Section 5.1 via the analytical solution obtained with Cramer's rule (Wikipedia 2022h). This recursive method is an efficient way to calculate the inverse of small symbolic matrices, but inefficient for large matrices.

```
R> mxinv(matrix(letters[1:4], byrow = TRUE, nrow = 2))

     [,1]                         [,2]
[1,] "(d) / (a*(d) + -c*(b))"  "-(b) / (a*(d) + -c*(b))"
[2,] "-(c) / (a*(d) + -c*(b))" "(a) / (a*(d) + -c*(b))"
```

The symbolic inverse offers a gain in performance when inverting a large number of matrices, as shown by replicating the test in Section 5.1 and replacing the determinant with the inverse.

```
R> n <- 4
R> e <- letters[1:n^2]
R> grid <- expand.grid(lapply(1:n^2, function(e) runif(2)))
R> colnames(grid) <- e
R> microbenchmark(
+   "numeric" = {
+     x <- apply(grid, 1, function(e) solve(matrix(e, nrow = n)))
+   },
+   "symbolic" = {
+     x <- evaluate(mxinv(matrix(e, nrow = n)), grid)
+   }
+ )

Unit: milliseconds
     expr    min      lq    mean median     uq    max neval
  numeric 577.56 609.486 635.178 630.98 653.74 727.72   100
 symbolic  58.15  68.664  99.692 100.13 125.72 196.49   100
```

5.3. Matrix product

The matrix product can be expressed in Einstein notation as shown in Section 6, thus inheriting the support for symbolic calculations.

```
R> a <- matrix(1:4, nrow = 2, byrow = TRUE)
R> b <- matrix(letters[1:4], nrow = 2, byrow = TRUE)
R> a %mx% b

     [,1]                 [,2]
[1,] "1 * (a) + 2 * (c)" "1 * (b) + 2 * (d)"
[2,] "3 * (a) + 4 * (c)" "3 * (b) + 4 * (d)"
```

6. Tensor algebra

A tensor may be represented as a multidimensional array. Just as a vector in an n-dimensional space is represented by a 1-dimensional array with n components, a matrix is represented by a 2-dimensional array with $n_1 \times n_2$ components, and a generic tensor can be represented by a

d-dimensional array with $n_1 \times \cdots \times n_d$ components. This makes the class 'array' available in base R an ideal candidate to represent mathematical tensors. In particular, the class stores its dimensions in the attribute `dim` that contains a vector giving the length for each dimension.

```
R> A <- array(1:24, dim = c(2, 3, 4))
R> attributes(A)

$dim
[1] 2 3 4
```

The package **calculus** reads this attribute to represent tensors in index notation, such as A_{ijk}. In particular, the function `index` is used to assign indices to the dimensions of the tensor by setting names to the attribute `dim`.

```
R> index(A) <- c("i", "j", "k")
R> attributes(A)

$dim
i j k
2 3 4
```

In this way, a tensor with named indices is represented by an `array` with named `dim`. The package **calculus** builds a set of tools to work with tensors on top of this class and provides the implementation of the Levi-Civita symbol and generalized Kronecker delta that often appears in tensor algebra. At the time of writing, the package makes no distinction between upper and lower indices, i.e., vectors and covectors (Wikipedia 2022d).

Levi-Civita symbol

In mathematics, particularly in linear algebra, tensor analysis, and differential geometry, the Levi-Civita symbol represents a collection of numbers; defined from the sign of a permutation of the natural numbers $1, 2, \dots, n$, for some positive integer n. It is named after the Italian mathematician and physicist Tullio Levi-Civita. Other names include the permutation symbol, antisymmetric symbol, or alternating symbol, which refer to its antisymmetric property and definition in terms of permutations (Wikipedia 2022l). In the general n-dimensional case, the Levi-Civita symbol is defined by:

$$\varepsilon_{i_1 i_2 \dots i_n} = \begin{cases} +1 & \text{if } (i_1, i_2, \dots, i_n) \text{ is an even permutation of } (1, 2, \dots, n) \\ -1 & \text{if } (i_1, i_2, \dots, i_n) \text{ is an odd permutation of } (1, 2, \dots, n) \\ 0 & \text{otherwise} \end{cases}$$

The function `epsilon` determines the parity of the permutation in C++ via efficient cycle decomposition (GeeksforGeeks 2022b) and constructs the Levi-Civita symbol in arbitrary dimension. For example the 2-dimensional Levi-Civita symbol is given by:

$$\varepsilon_{ij} = \begin{cases} +1 & \text{if } (i, j) = (1, 2) \\ -1 & \text{if } (i, j) = (2, 1) \\ 0 & \text{if } i = j \end{cases}$$

```
R> epsilon(2)

     [,1] [,2]
[1,]    0    1
[2,]   -1    0
```

And in 3 dimensions:

$$\varepsilon_{ijk} = \begin{cases} +1 & \text{if } (i,j,k) \text{ is } (1,2,3),(2,3,1), \text{ or } (3,1,2), \\ -1 & \text{if } (i,j,k) \text{ is } (3,2,1),(1,3,2), \text{ or } (2,1,3), \\ 0 & \text{if } i=j, \text{ or } j=k, \text{ or } k=i \end{cases}$$

```
R> epsilon(3)

, , 1

     [,1] [,2] [,3]
[1,]    0    0    0
[2,]    0    0    1
[3,]    0   -1    0

, , 2

     [,1] [,2] [,3]
[1,]    0    0   -1
[2,]    0    0    0
[3,]    1    0    0

, , 3

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

Generalized Kronecker delta

The generalized Kronecker delta or multi-index Kronecker delta of order $2p$ is a type (p,p) tensor that is a completely antisymmetric in its p upper indices, and also in its p lower indices (Wikipedia 2022i). In terms of the indices, the generalized Kronecker delta is defined as (Frankel 2011):

$$\delta^{\mu_1\ldots\mu_p}_{\nu_1\ldots\nu_p} = \begin{cases} +1 & \text{if } (\nu_1\ldots\nu_p) \text{ is an even permutation of } (\mu_1\ldots\mu_p) \\ -1 & \text{if } (\nu_1\ldots\nu_p) \text{ is an odd permutation of } (\mu_1\ldots\mu_p) \\ 0 & \text{otherwise} \end{cases}$$

When $p = 1$, the definition reduces to the standard Kronecker delta that corresponds to the $n \times n$ identity matrix $I_{ij} = \delta^i_j$ where i and j take the values $1, 2, \ldots, n$. The implementation is based on efficient cycle decomposition to determine the parity of the permutations as done for the Levi-Civita symbol.

```
R> delta(n = 3, p = 1)

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

6.1. Tensor contraction

Tensor contraction can be seen as a generalization of the trace for a square matrix. In the general case, a tensor can be contracted by summing over pairs of repeated indices that share the same dimensions. This is achieved via the C++ optimized function `contraction`. For each set of repeated indices, the function first permutes the `array` to move the repeated indices to the end, e.g., $A_{iji} \to A_{jii}$. Then, the array is coalesced into a vector and passed to C++. As the dummy dimensions have been sorted, C++ only needs the length of the dummy dimension in order to identify the elements to sum up. This is done via simple and efficient `for` loops. The result is returned to R and the next set of repeated indices is processed.

Consider the following $2 \times 2 \times 2$ tensor:

```
R> x <- array(1:8, dim = c(2, 2, 2))
R> print(x)

, , 1

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

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8
```

The trace of the tensor $T = \sum_i T_{iii}$ is obtained with:

```
R> contraction(x)

[1] 9
```

The contraction on the first and third dimension $T_j = \sum_i T_{iji}$ can be computed with:

```
R> index(x) <- c("i", "j", "i")
R> contraction(x)

[1]  7 11
```

Finally, it is possible to preserve the dummy dimensions $T_{ij} = T_{iji}$ by setting the argument `drop = FALSE`:

```
R> index(x) <- c("i", "j", "i")
R> contraction(x, drop = FALSE)

     [,1] [,2]
[1,]    1    6
[2,]    3    8
```

In this way, it is possible to compute arbitrary contraction of tensors such as $T_{klm} = \sum_{ij} T_{ikiiljjm}$ or $T_{ijklm} = T_{ikiiljjm}$ to preserve the dummy dimensions.

6.2. Einstein summation

In mathematics, the Einstein notation or Einstein summation convention is a notational convention that implies summation over a set of repeated indices. When an index variable appears twice, it implies summation over all the values of the index (Wikipedia 2022d). For instance the matrix product can be written in terms of Einstein notation as:

$$C_{ij} = A_{ik}B_{kj} \equiv \sum_k A_{ik}B_{kj}$$

An arbitrary summation of the kind

$$D_k = A_{ijj}B_{iijk}C_j \equiv \sum_{ij} A_{ijj}B_{iijk}C_j = \sum_j \left(\sum_i A_{ijj}B_{iijk} \right) C_j$$

is implemented as follows:

1. Contract the first tensor and preserve the dummy dimensions: $A_{ijj} \to A_{ij}$.
2. Contract the second tensor and preserve the dummy dimensions: $B_{iijk} \to B_{ijk}$.
3. Permute and move the summation indices to the end: $A_{ij} \to A_{ij}$, $B_{ijk} \to B_{kij}$.
4. Compute the elementwise product on the repeated indices: $(AB)_{kij} = A_{ij}B_{kij}$.
5. Sum over the summation indices that do now appear in the other tensors:

$$(AB)_{kj} = \sum_i (AB)_{kij}$$

6. Contract the third tensor and preserve the dummy dimensions: $C_j \to C_j$.
7. Permute and move the summation indices to the end: $(AB)_{kj} \to (AB)_{kj}$, $C_j \to C_j$.

8. Compute the elementwise product on the repeated indices: $(ABC)_{kj} = (AB)_{kj}C_j$.

9. Sum over the summation indices that do now appear in the other tensors:

$$D_k = (ABC)_k = \sum_j (ABC)_{kj}$$

10. Iterate until all the tensors in the summation are considered.

The function `einstein` provides a convenient way to compute general Einstein summations among two or more tensors, with or without repeated indices appearing in the same tensor. The function supports both numerical and symbolical calculations implemented via the usage of C++ templates that operate with generic types and allow the function to work on the different data types without being rewritten for each one. The following example illustrates a sample Einstein summation with mixed data types:

$$D_{jk} = A_{ij}B_{ki}C_{ii}$$

```
R> A <- array(1:6, dim = c(i = 2, j = 3))
R> print(A)

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

R> B <- array(1:4, dim = c(k = 2, i = 2))
R> print(B)

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

R> C <- array(letters[1:4], dim = c(i = 2, i = 2))
R> print(C)

     [,1] [,2]
[1,] "a"  "c"
[2,] "b"  "d"

R> einstein(A, B, C)

     [,1]                 [,2]
[1,] "1 * (a) + 6 * (d)"  "2 * (a) + 8 * (d)"
[2,] "3 * (a) + 12 * (d)" "6 * (a) + 16 * (d)"
[3,] "5 * (a) + 18 * (d)" "10 * (a) + 24 * (d)"
```

In the particular case of Einstein summations between two numeric tensors that, after proper contraction and permutation, can be rewritten as

$$C_{i_1 \dots i_a, j_1 \dots j_b} = A_{i_1 \dots i_a, k_1 \dots k_n} B_{k_1 \dots k_n, j_1 \dots j_b}$$

the function implements the following scheme:

1. Reshape the tensor $A_{i_1 \dots i_a, k_1 \dots k_n}$ in the matrix $A_{I,K}$ where the dimension of I is the product of the dimensions of $i_1 \dots i_a$ and the dimensions of K is the product of the dimensions of $k_1 \dots k_n$.
2. Reshape the tensor $B_{k_1 \dots k_n, j_1 \dots j_b}$ in the matrix $B_{K,J}$ where the dimension of K is the product of the dimensions of $k_1 \dots k_n$ and the dimensions of J is the product of the dimensions of $j_1 \dots j_b$.
3. Compute the matrix product $C_{IJ} = A_{IK} B_{KJ}$.
4. Reshape the matrix C_{IJ} in the tensor $C_{i_1 \dots i_a, j_1 \dots j_b}$.

In this way, it is sufficient to change the attribute `dim` of the `arrays` and the Einstein summation is written in terms of a matrix product that can be computed efficiently in base R. This approach is almost twice as fast as the alternative implementation for the Einstein summation in the R package **tensorA** for advanced tensor arithmetic with named indices (Van den Boogaart 2020).

```
R> a <- array(1:1000000, dim = c(a = 2, i = 5, j = 100, k = 50, d = 20))
R> b <- array(1:100000, dim = c(a = 2, j = 100, i = 5, l = 100))
R> Ta <- tensorA::to.tensor(a)
R> Tb <- tensorA::to.tensor(b)
R> microbenchmark(
+    "calculus" = calculus::einstein(a, b),
+    "tensorA" = tensorA::einstein.tensor(Ta, Tb)
+ )

Unit: milliseconds
     expr     min      lq    mean  median      uq    max neval
 calculus  39.078  39.644  41.976  40.189  41.476  66.10   100
  tensorA 126.918 129.155 132.025 130.104 131.315 156.18   100
```

6.3. Inner product

The inner product is computed in base R for `numeric` arrays or via Einstein summation for `character` arrays:

$$A_{i_1 \dots i_n} B_{i_1 \dots i_n}$$

```
R> 1:3 %inner% letters[1:3]

[1] "1 * (a) + 2 * (b) + 3 * (c)"
```

Dot product

The dot product between arrays with different dimensions is computed by taking the inner product on the last dimensions of the two arrays. It is written in Einstein notation as:

$$A_{i_1 \dots i_a j_1 \dots j_n} B_{j_1 \dots j_n}$$

```
R> matrix(1:6, byrow = TRUE, nrow = 2, ncol = 3) %dot% letters[1:3]
```

```
[1] "1 * (a) + 2 * (b) + 3 * (c)" "4 * (a) + 5 * (b) + 6 * (c)"
```

6.4. Outer product

The outer product is computed in base R for `numeric` arrays or via Einstein summation for `character` arrays:

$$A_{i_1 \dots i_a} B_{j_1 \dots j_b}$$

```
R> 1:3 %outer% letters[1:3]
```

```
     [,1]      [,2]      [,3]
[1,] "1 * (a)" "1 * (b)" "1 * (c)"
[2,] "2 * (a)" "2 * (b)" "2 * (c)"
[3,] "3 * (a)" "3 * (b)" "3 * (c)"
```

6.5. Kronecker product

The package extends the generalized `kronecker` product available in base R with support for arrays of type `character`.

```
R> 1:3 %kronecker% letters[1:3]
```

```
[1] "1 * (a)" "1 * (b)" "1 * (c)" "2 * (a)" "2 * (b)" "2 * (c)" "3 * (a)"
[8] "3 * (b)" "3 * (c)"
```

7. Derivatives

The function `derivative` performs high-order symbolic and numerical differentiation for generic tensors with respect to an arbitrary number of variables. The function behaves differently depending on the arguments `order`, the order of differentiation, and `var`, the variable names with respect to which the derivatives are computed.

When multiple variables are provided and `order` is a single integer n, then the n-th order derivative is computed for each element of the tensor with respect to each variable:

$$D = \partial^{(n)} \otimes F$$

that is:

$$D_{i,\dots,j,k} = \partial_k^{(n)} F_{i,\dots,j}$$

where F is the tensor of functions and $\partial_k^{(n)}$ denotes the n-th order partial derivative with respect to the k-th variable.

When `order` matches the length of `var`, it is assumed that the differentiation order is provided for each variable. In this case, each element is derived n_k times with respect to the k-th variable, for each of the m variables.

$$D_{i,\dots,j} = \partial_1^{(n_1)} \cdots \partial_m^{(n_m)} F_{i,\dots,j}$$

The same applies when `order` is a named vector giving the differentiation order for each variable. For example, `order = c(x = 1, y = 2)` differentiates once with respect to x and twice with respect to y. A call with `order = c(x = 1, y = 0)` is equivalent to `order = c(x = 1)`.

To compute numerical derivatives or to evaluate symbolic derivatives at a point, the function accepts a named vector for the argument `var`; e.g., `var = c(x = 1, y = 2)` evaluates the derivatives in $x = 1$ and $y = 2$. For functions where the first argument is used as a parameter vector, `var` should be a `numeric` vector indicating the point at which the derivatives are to be calculated.

7.1. Symbolic derivatives

Symbolic derivatives are computed via the `D` function available in base R. The function is iterated multiple times for second and higher order derivatives.

7.2. Numerical derivatives

Numerical derivatives are computed via the scheme described in Eberly (2008) for central finite differences. In particular, the derivative of a function f with respect to one or more variables is approximated up to the degree $O(h_1^p \dots h_m^p)$ by:

$$\begin{aligned}\partial_{n_1,\dots,n_m} f &= \partial_{x_1}^{(n_1)} \dots \partial_{x_m}^{(n_m)} f(x_1,\dots,x_m) = \\ &= \frac{n_1!\dots n_m!}{h_1^{n_1}\dots h_m^{n_m}} \sum_{j_1=-i^{(n_1)}}^{i^{(n_1)}} \cdots \sum_{j_m=-i^{(n_m)}}^{i^{(n_m)}} C_{j_1}^{(n_1)} \cdots C_{j_m}^{(n_m)} f(x_1 + j_1 h_1, \dots, x_m + j_m h_m)\end{aligned}$$

where n_k is the order of differentiation with respect to the k-th variable, h are the step sizes, i are equal to $i^{(n)} = \lfloor (n + p - 1)/2 \rfloor$, and the coefficients $C_j^{(n)}$ are computed by solving the following linear system for each n:

$$\begin{bmatrix} C_{-i} \\ C_{-i+1} \\ C_{-i+2} \\ \vdots \\ C_{-i+n+1} \\ \vdots \\ C_i \end{bmatrix} = \begin{bmatrix} (-i)^0 & \dots & (-1)^0 & 0 & 1^0 & \dots & i^0 \\ (-i)^1 & \dots & (-1)^1 & 0 & 1^1 & \dots & i^1 \\ (-i)^2 & \dots & (-1)^2 & 0 & 1^2 & \dots & i^2 \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ (-i)^{n+1} & \dots & (-1)^{n+1} & 0 & 1^{n+1} & \dots & i^{n+1} \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ (-i)^{2i} & \dots & (-1)^{2i} & 0 & 1^{2i} & \dots & i^{2i} \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

The summation is computed via Einstein notation by setting:

$$C^{(n_1)}_{j_1} \cdots C^{(n_m)}_{j_m} F_{j_1,\ldots,j_m} \equiv \sum_{j_1=-i^{(n_1)}}^{i^{(n_1)}} \cdots \sum_{j_m=-i^{(n_m)}}^{i^{(n_m)}} C^{(n_1)}_{j_1} \cdots C^{(n_m)}_{j_m} f(x_1 + j_1 h_1, \ldots, x_m + j_m h_m)$$

7.3. Examples

Symbolic derivatives of univariate functions: $\partial_x \sin(x)$.

```
R> derivative(f = "sin(x)", var = "x")
```

```
[1] "cos(x)"
```

Evaluation of symbolic and numerical derivatives: $\partial_x \sin(x)|_{x=0}$.

```
R> sym <- derivative(f = "sin(x)", var = c(x = 0))
R> num <- derivative(f = function(x) sin(x), var = c(x = 0))
```

```
Symbolic  Numeric
       1        1
```

High order symbolic and numerical derivatives: $\partial^{(4)}_x \sin(x)|_{x=0}$.

```
R> sym <- derivative(f = "sin(x)", var = c(x = 0), order = 4)
R> num <- derivative(f = function(x) sin(x), var = c(x = 0), order = 4)
```

```
   Symbolic     Numeric
 0.0000e+00 -2.922023e-11
```

Symbolic derivatives of multivariate functions: $\partial^{(1)}_x \partial^{(2)}_y y^2 \sin(x)$.

```
R> derivative(f = "y^2 * sin(x)", var = c("x", "y"), order = c(1, 2))
```

```
[1] "2 * cos(x)"
```

Numerical derivatives of multivariate functions: $\partial^{(1)}_x \partial^{(2)}_y y^2 \sin(x)|_{x=0,y=0}$ with degree of accuracy $O(h^6)$.

```
R> f <- function(x, y) y^2 * sin(x)
R> derivative(f, var = c(x = 0, y = 0), order = c(1, 2), accuracy = 6)
```

```
[1] 2
```

Symbolic gradient of multivariate functions: $\partial_{x,y} x^2 y^2$.

```
R> derivative("x^2 * y^2", var = c("x", "y"))
```

```
     [,1]           [,2]
[1,] "2 * x * y^2" "x^2 * (2 * y)"
```

High order derivatives of multivariate functions: $\partial^{(6)}_{x,y} x^6 y^6$.

```
R> derivative("x^6 * y^6", var = c("x", "y"), order = 6)
```

```
     [,1]                               [,2]
[1,] "6 * (5 * (4 * (3 * 2))) * y^6" "x^6 * (6 * (5 * (4 * (3 * 2))))"
```

Numerical gradient of multivariate functions: $\partial_{x,y} x^2 y^2 |_{x=1,y=2}$.

```
R> f <- function(x, y) x^2 * y^2
R> derivative(f, var = c(x = 1, y = 2))
```

```
     [,1] [,2]
[1,]    8    4
```

Numerical Jacobian of vector valued functions: $\partial_{x,y} [xy, x^2 y^2] |_{x=1,y=2}$.

```
R> f <- function(x, y) c(x*y, x^2 * y^2)
R> derivative(f, var = c(x = 1, y = 2))
```

```
     [,1] [,2]
[1,]    2    1
[2,]    8    4
```

Numerical Jacobian of vector valued functions where the first argument is used as a parameter vector: $\partial_X [\sum_i x_i, \prod_i x_i] |_{X=0}$.

```
R> f <- function(x) c(sum(x), prod(x))
R> derivative(f, var = c(0, 0, 0))
```

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

7.4. Performance

Table 1 compares the accuracy of Richardson extrapolation (Wikipedia 2022n) implemented in the package **numDeriv** (Gilbert and Varadhan 2019) with central finite differences implemented in **calculus** using `accuracy = 4` by default. 10^4 derivatives have been computed for the four functions: $x^2 e^x$, $x \sin(x^2)$, $x \log(x^2)$, $e^{\sin(x)}$. The table shows the mean relative error and the corresponding standard deviation.

Although it is known that Richardson extrapolation is usually more accurate than finite differences, the results of the two packages are very similar. Both packages produce accurate derivatives with relative errors close to the precision of the machine (10^{-16}). On the other hand, **calculus** proves to be significantly faster than **numDeriv** for multivariate functions as shown in the following benchmarking.

	Package	N	Mean	SD
$x^2 \exp(x)$	**calculus**	1.0×10^4	9.3×10^{-14}	1.0×10^{-11}
	numDeriv	1.0×10^4	-3.1×10^{-15}	5.3×10^{-12}
$x \sin(x^2)$	**calculus**	1.0×10^4	3.9×10^{-13}	1.8×10^{-11}
	numDeriv	1.0×10^4	-1.5×10^{-13}	1.2×10^{-11}
$x \log(x^2)$	**calculus**	1.0×10^4	4.4×10^{-14}	1.3×10^{-11}
	numDeriv	1.0×10^4	-5.0×10^{-15}	6.1×10^{-12}
$\exp(\sin(x))$	**calculus**	1.0×10^4	-7.3×10^{-13}	6.0×10^{-11}
	numDeriv	1.0×10^4	-1.8×10^{-13}	3.1×10^{-11}

Table 1: Comparison between Richardson extrapolation (**numDeriv**) and finite differences (**calculus**) for four test functions. The table reports the number of derivatives computed for each function (N), the mean relative error (Mean), and the standard deviation of the error (SD).

```
R> x <- rep(0, 1000)
R> f <- function(x) sum(x)
R> microbenchmark(
+    "calculus O(2)" = calculus::derivative(f, x, accuracy = 2),
+    "calculus O(4)" = calculus::derivative(f, x, accuracy = 4),
+    "calculus O(6)" = calculus::derivative(f, x, accuracy = 6),
+    "calculus O(8)" = calculus::derivative(f, x, accuracy = 8),
+    "numDeriv" = numDeriv::grad(f, x)
+ )
```

```
Unit: milliseconds
          expr    min     lq   mean median     uq     max neval
 calculus O(2) 11.883 12.628 14.415 13.102 16.066  43.296   100
 calculus O(4) 19.853 20.731 22.484 22.997 23.807  25.663   100
 calculus O(6) 28.001 31.235 31.967 31.796 32.393  57.650   100
 calculus O(8) 38.242 39.133 39.983 39.757 40.395  43.658   100
      numDeriv 68.153 72.075 75.907 73.408 75.944 103.168   100
```

8. Taylor series

Based on the derivatives in the previous section, the function `taylor` provides a convenient way to compute the Taylor series of arbitrary unidimensional or multidimensional functions. The mathematical function can be specified both as a `character` string or as a `function`. Symbolic or numerical methods are applied accordingly. For univariate functions, the n-th order Taylor approximation centered in x_0 is given by:

$$f(x) \simeq \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k$$

where $f^{(k)}(x_0)$ denotes the k-th order derivative evaluated in x_0. By using multi-index notation, the Taylor series is generalized to multidimensional functions with an arbitrary number of variables:

$$f(x) \simeq \sum_{|k|=0}^{n} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k$$

where now $x = (x_1, \ldots, x_d)$ is the vector of variables, $k = (k_1, \ldots, k_d)$ gives the order of differentiation with respect to each variable $f^{(k)} = \frac{\partial^{(|k|)} f}{\partial_{x_1}^{(k_1)} \cdots \partial_{x_d}^{(k_d)}}$, and:

$$|k| = k_1 + \cdots + k_d \qquad k! = k_1! \cdots k_d! \qquad x^k = x_1^{k_1} \cdots x_d^{k_d}$$

The summation runs for $0 \le |k| \le n$ and identifies the set

$$\{(k_1, \cdots, k_d) : k_1 + \cdots k_d \le n\}$$

that corresponds to the partitions of the integer n. These partitions can be computed with the function **partitions** that is included in the package and optimized in C++ for speed and flexibility. The implementation computes the next partition using the values in the current partition (GeeksforGeeks 2022a), permutes them and fills them with zeros if needed. For example, the following call generates the partitions needed for the 2-nd order Taylor expansion for a function of 3 variables:

```
R> partitions(n = 2, length = 3, fill = TRUE, perm = TRUE, equal = FALSE)

     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    0    0    0    1    0    0    2    0    1     1
[2,]    0    0    1    0    0    2    0    1    0     1
[3,]    0    1    0    0    2    0    0    1    1     0
```

Based on these partitions, the function **taylor** computes the corresponding derivatives and builds the Taylor series. The output is a **list** containing the Taylor series, the order of the expansion, and a **data.frame** containing the variables, coefficients and degrees of each term in the Taylor series.

```
R> taylor("exp(x)", var = "x", order = 2)

$f
[1] "(1) * 1 + (1) * x^1 + (0.5) * x^2"

$order
[1] 2

$terms
  var coef degree
0   1  1.0      0
1 x^1  1.0      1
2 x^2  0.5      2
```

By default, the series is centered in $x_0 = 0$ but the function also supports $x_0 \neq 0$, the multivariable case, and the approximation of user defined R functions.

```
R> f <- function(x, y) log(y) * sin(x)
R> taylor(f, var = c(x = 0, y = 1), order = 2)

$f
[1] "(0.999999999969436) * x^1*(y-1)^1"

$order
[1] 2

$terms
             var coef degree
0,0            1    0      0
0,1      (y-1)^1    0      1
1,0          x^1    0      1
0,2      (y-1)^2    0      2
2,0          x^2    0      2
1,1 x^1*(y-1)^1    1      2
```

9. Hermite polynomials

Hermite polynomials are obtained by differentiation of the Gaussian kernel:

$$H_\nu(x, \Sigma) = \exp\left(\frac{1}{2}x_i\Sigma_{ij}x_j\right)(-\partial_x)^\nu \exp\left(-\frac{1}{2}x_i\Sigma_{ij}x_j\right)$$

where Σ is a d-dimensional square matrix and $\nu = (\nu_1 \dots \nu_d)$ is the vector representing the order of differentiation for each variable $x = (x_1 \dots x_d)$. In the case where $\Sigma = 1$ and $x = x_1$ the formula reduces to the standard univariate Hermite polynomials:

$$H_\nu(x) = e^{\frac{x^2}{2}}(-1)^\nu \frac{d^\nu}{dx^\nu}e^{-\frac{x^2}{2}}$$

High order derivatives of the kernel $e^{-\frac{x^2}{2}}$ cannot performed efficiently in base R. The following example shows the naive calculation of $\frac{d^2}{dx^2}e^{-\frac{x^2}{2}}$ via the function `D`:

```
R> D(D(expression(exp(-x^2 / 2)), "x"), "x")

-(exp(-x^2/2) * (2/2) - exp(-x^2/2) * (2 * x/2) * (2 * x/2))
```

The resulting expression is not simplified and this leads to more and more iterations of the chain rule to compute higher order derivatives. The expression grows fast and soon requires long computational times and gigabytes of storage.

```
R> f <- expression(exp(-x^2 / 2))
R> for (i in 1:14) f <- D(f, "x")
R> object.size(f)

7925384376 bytes
```

To overcome this difficulty, the function `hermite` implements the following scheme. First, it differentiates the Gaussian kernel. Then, the kernel is dropped from the resulting expression. In this way, the expression becomes a polynomial of degree 1. The `taylor` series of order 1 is computed in order to extract the coefficients of the polynomial and rewrite it compact form. The polynomial is now multiplied by the Gaussian kernel and differentiated again. The kernel is dropped so that the expression becomes a polynomial of degree 2. The `taylor` series of order 2 is computed and the scheme is iterated until reaching the desired degree ν. The same applies when $\nu = (\nu_1 \dots \nu_d)$ represents the multi index of multivariate Hermite polynomials. The scheme allows to reduce the computational time and storage, return a well formatted output, and generate recursively all the Hermite polynomials of degree ν' where $|\nu'| \leq |\nu|$. The output is a `list` of Hermite polynomials of degree ν', where each polynomial is represented by the corresponding `taylor` series.

10. Ordinary differential equations

The function `ode` provides solvers for systems of ordinary differential equations of the type:

$$\frac{dy}{dt} = f(t, y), \quad y(t_0) = y_0$$

where y is the vector of state variables. Two solvers are available: the simpler and faster Euler scheme (Wikipedia 2022f) or the more accurate 4-th order Runge-Kutta method (Wikipedia 2022o). Although many packages already exist to solve ordinary differential equations in R (Petzoldt and Soetaert 2022), they usually represent the function f either with an R function – see e.g., **deSolve** (Soetaert, Petzoldt, and Setzer 2010), **odeintr** (Keitt 2017), and **pracma** (Borchers 2022) – or with characters – see e.g., **yuima** (Brouste *et al.* 2014). While the representation via R functions is usually more efficient, the symbolic representation is easier to adopt for beginners and more flexible for advanced users to handle systems that might have been generated via symbolic programming. The package **calculus** supports both the representations and uses hashed `environments` to accelerate symbolic evaluations. Consider the following system:

$$\frac{d}{dt}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x \\ x(1+\cos(10t)) \end{bmatrix}, \quad \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

The vector-valued function f representing the system can be specified as a vector of characters, or a function returning a numeric vector, giving the values of the derivatives at time t. The initial conditions are set with the argument `var` and the time variable can be specified with `timevar`.

```
R> sim <- ode(f = c("x", "x * (1 + cos(10 * t))"), var = c(x = 1, y = 1),
+    times = seq(0, 2 * pi, by = 0.001), timevar = "t")
```

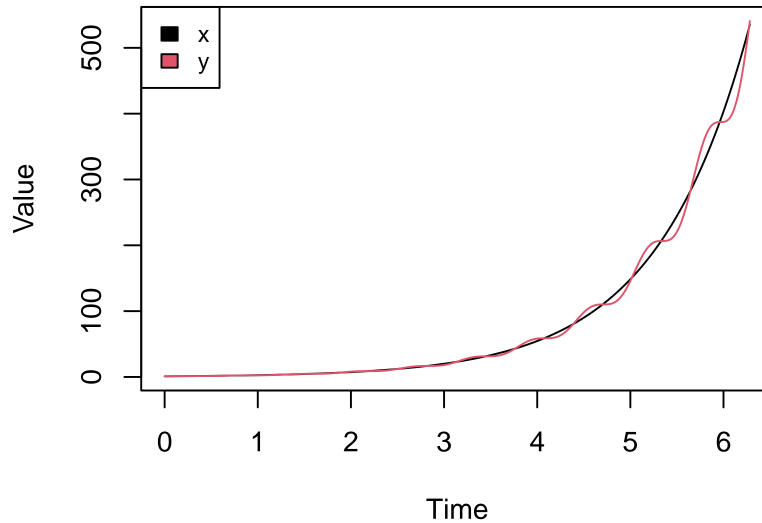

Figure 1: Solution to the system of ordinary differential equations in Section 10 obtained with the function `ode` using the Runge-Kutta method.

The solution to the system is represented in Figure 1.

11. Differential operators

Orthogonal coordinates are a special but extremely common case of curvilinear coordinates where the coordinate surfaces all meet at right angles. The chief advantage of non-Cartesian coordinates is that they can be chosen to match the symmetry of the problem. For example, spherical coordinates are the most common curvilinear coordinate systems and are used in Earth sciences, cartography, quantum mechanics, relativity, and engineering (Wikipedia 2022b). These coordinates may be derived from a set of Cartesian coordinates by using a transformation that is locally invertible (a one-to-one map) at each point. This means that one can convert a point given in a Cartesian coordinate system to its curvilinear coordinates and back. Differential operators such as the gradient, divergence, curl, and Laplacian can be transformed from one coordinate system to another via the usage of scale factors (Table 2) (Wikipedia 2022m). The package implements these operators in Cartesian, polar, spherical, cylindrical, parabolic coordinates, and supports arbitrary orthogonal coordinates systems defined by custom scale factors.

11.1. Gradient

The gradient of a scalar-valued function F is the vector $(\nabla F)_i$ whose components are the partial derivatives of F with respect to each variable i. In arbitrary orthogonal coordinate systems, the gradient is expressed in terms of the scale factors h_i as follows:

$$(\nabla F)_i = \frac{1}{h_i}\partial_i F$$

The function `gradient` implements the symbolic and numeric gradient for R functions, expressions, and characters. In Cartesian coordinates:

Curvilinear coordinates (q_1, q_2, q_3)	Transformation from cartesian (x, y, z)	Scale factors
Spherical polar coordinates (r, θ, ϕ)	$x = r \sin\theta \cos\phi$ $y = r \sin\theta \sin\phi$ $z = r \cos\theta$	$h_1 = 1$ $h_2 = r$ $h_3 = r \sin\theta$
Cylindrical polar coordinates (r, ϕ, z)	$x = r \cos\phi$ $y = r \sin\phi$ $z = z$	$h_1 = h_3 = 1$ $h_2 = r$
Parabolic coordinates (u, v, ϕ)	$x = uv \cos\phi$ $y = uv \sin\phi$ $z = \frac{1}{2}(u^2 - v^2)$	$h_1 = h_2 = \sqrt{u^2 + v^2}$ $h_3 = uv$
Parabolic cylindrical coordinates (u, v, z)	$x = \frac{1}{2}(u^2 - v^2)$ $y = uv$ $z = z$	$h_1 = h_2 = \sqrt{u^2 + v^2}$ $h_3 = 1$

Table 2: Example of scale factors for common coordinate systems.

```
R> gradient("x * y * z", var = c("x", "y", "z"))

[1] "y * z" "x * z" "x * y"
```

and in spherical coordinates:

```
R> gradient("x * y * z", var = c("x", "y", "z"), coordinates = "spherical")

[1] "1/1 * (y * z)"          "1/x * (x * z)"          "1/(x*sin(y)) * (x * y)"
```

To support arbitrary orthogonal coordinate systems, it is possible to pass custom scale factors to the argument `coordinates`. For instance, the following call is equivalent to the previous example in spherical coordinates where the scale factors are now explicitly specified:

```
R> gradient("x * y * z", var = c("x", "y", "z"),
+    coordinates = c(1, "x", "x * sin(y)"))

[1] "1/(1) * (y * z)"        "1/(x) * (x * z)"        "1/(x*sin(y)) * (x * y)"
```

Numerical methods are applied when working with functions with the same sintax introduced for derivatives in Section 7:

```
R> f <- function(x, y, z) x * y * z
R> gradient(f, var = c(x = 1, y = pi/2, z = 0), coordinates = "spherical")
```

```
[1] 0.0000 0.0000 1.5708
```

or in vectorized form:

```
R> f <- function(x) x[1] * x[2] * x[3]
R> gradient(f, var = c(1, pi/2, 0), coordinates = "spherical")
```

```
[1] 0.0000 0.0000 1.5708
```

When the function F is a tensor-valued function $F_{d_1,\dots,d_n}$, the gradient is computed for each scalar component.

$$(\nabla F_{d_1,\dots,d_n})_i = \frac{1}{h_i}\partial_i F_{d_1,\dots,d_n}$$

In particular, this reduces to the Jacobian matrix for vector-valued functions F_{d_1}:

```
R> f <- function(x) c(prod(x), sum(x))
R> gradient(f, var = c(3, 2, 1))
```

```
     [,1] [,2] [,3]
[1,]    2    3    6
[2,]    1    1    1
```

that may be expressed in arbitrary orthogonal coordinate systems.

```
R> f <- function(x) c(prod(x), sum(x))
R> gradient(f, var = c(3, 2, 1), coordinates = "cylindrical")
```

```
     [,1]    [,2] [,3]
[1,]    2 1.00000    6
[2,]    1 0.33333    1
```

Jacobian

The function `jacobian` is a wrapper for `gradient` that always returns the Jacobian as a `matrix`, even in the case of unidimensional scalar-valued functions.

```
R> f <- function(x) x^2
R> jacobian(f, var = c(1))
```

```
     [,1]
[1,]    2
```

Hessian

In Cartesian coordinates, the Hessian of a scalar-valued function F is the square matrix of second-order partial derivatives:

$$(H(F))_{ij} = \partial_{ij}F$$

It might be tempting to apply the definition of the Hessian as the Jacobian of the gradient to write it in terms of the scale factors. However, this results in a Hessian matrix that is not symmetric and ignores the distinction between vector and covectors in tensor analysis (see e.g., Masi 2007). The generalization to arbitrary coordinate system is out of the scope of this paper and only Cartesian coordinates are supported in this case:

```
R> f <- function(x, y, z) x * y * z
R> hessian(f, var = c(x = 3, y = 2, z = 1))
```

```
             [,1]         [,2]         [,3]
[1,] 1.222284e-11 1.00000e+00 2.000000e+00
[2,] 1.000000e+00 2.75014e-11 3.000000e+00
[3,] 2.000000e+00 3.00000e+00 1.100056e-10
```

When the function F is a tensor-valued function $F_{d_1,\ldots,d_n}$, the `hessian` is computed for each scalar component.

$$(H(F_{d_1,\ldots,d_n}))_{ij} = \partial_{ij} F_{d_1,\ldots,d_n}$$

In this case, the function returns an `array` of Hessian matrices:

```
R> f <- function(x, y, z) c(x * y * z, x + y + z)
R> h <- hessian(f, var = c(x = 3, y = 2, z = 1))
```

that can be extracted with the corresponding indices.

```
R> h[1,,]
```

```
             [,1]         [,2]         [,3]
[1,] 1.222284e-11 1.00000e+00 2.000000e+00
[2,] 1.000000e+00 2.75014e-11 3.000000e+00
[3,] 2.000000e+00 3.00000e+00 1.100056e-10
```

```
R> h[2,,]
```

```
              [,1]          [,2]          [,3]
[1,] -1.833426e-11  7.883472e-12 -3.627321e-12
[2,]  7.883472e-12 -6.416993e-11  1.090683e-11
[3,] -3.627321e-12  1.090683e-11  9.167132e-11
```

11.2. Divergence

The divergence of a vector-valued function F_i produces a scalar value $\nabla \cdot F$ representing the volume density of the outward flux of the vector field from an infinitesimal volume around a given point (Wikipedia 2022c). In terms of scale factors, it is expressed as follows:

$$\nabla \cdot F = \frac{1}{J} \sum_i \partial_i \left(\frac{J}{h_i} F_i \right)$$

where $J = \prod_i h_i$. When F is an `array` of vector-valued functions $F_{d_1,\dots,d_n,i}$, the `divergence` is computed for each vector:

$$(\nabla \cdot F)_{d_1,\dots,d_n} = \frac{1}{J}\sum_i \partial_i \left(\frac{J}{h_i} F_{d_1,\dots,d_n,i}\right) = \frac{1}{J}\sum_i \partial_i (J h_i^{-1}) F_{d_1,\dots,d_n,i} + J h_i^{-1} \partial_i (F_{d_1,\dots,d_n,i})$$

where the last equality is preferable in practice as the derivatives of the scale factor can be computed symbolically and the computation of the derivatives of F is more efficient than the direct computation of $\partial_i \left(\frac{J}{h_i} F_{d_1,\dots,d_n,i}\right)$ via finite differences. In Cartesian coordinates:

```
R> f <- c("x^2", "y^2", "z^2")
R> divergence(f, var = c("x","y","z"))

[1] "2 * x + 2 * y + 2 * z"
```

In polar coordinates:

```
R> f <- c("sqrt(r) / 10", "sqrt(r)")
R> divergence(f, var = c("r", "phi"), coordinates = "polar")

[1] "(0.5 * r^-0.5/10 * r + (sqrt(r)/10)) / (1*r)"
```

And for tensors of vector-valued functions:

```
R> f <- matrix(c("x^2", "y^2", "z^2", "x", "y", "z"),
+    nrow = 2, byrow = TRUE)
R> divergence(f, var = c("x", "y", "z"))

[1] "2 * x + 2 * y + 2 * z" "1 + 1 + 1"
```

The same syntax holds for functions where numerical methods are automatically applied:

```
R> f <- function(x,y,z) {
+    matrix(c(x^2, y^2, z^2, x, y, z), nrow = 2, byrow = TRUE)
+ }
R> divergence(f, var = c(x = 0, y = 0, z = 0))

[1] 0 3
```

11.3. Curl

The curl of a vector-valued function F_i at a point is represented by a vector whose length and direction denote the magnitude and axis of the maximum circulation (Wikipedia 2022a). In 2 dimensions, the curl is written in terms of the scale factors h and the Levi-Civita symbol ϵ as follows:

$$\nabla \times F = \frac{1}{h_1 h_2}\sum_{ij} \epsilon_{ij} \partial_i \left(h_j F_j\right) = \frac{1}{h_1 h_2}\left(\partial_1 \left(h_2 F_2\right) - \partial_2 \left(h_1 F_1\right)\right)$$

In 3 dimensions:

$$(\nabla \times F)_k = \frac{h_k}{J} \sum_{ij} \epsilon_{ijk} \partial_i \left(h_j F_j\right)$$

where $J = \prod_i h_i$. This suggests to implement the `curl` in $m+2$ dimensions in such a way that the formula reduces correctly to the previous cases:

$$(\nabla \times F)_{k_1 \dots k_m} = \frac{h_{k_1} \cdots h_{k_m}}{J} \sum_{ij} \epsilon_{ijk_1 \dots k_m} \partial_i \left(h_j F_j\right)$$

And in particular, when F is an `array` of vector-valued functions $F_{d_1,\dots,d_n,i}$ the `curl` is computed for each vector:

$$\begin{aligned}
(\nabla \times F)_{d_1 \dots d_n, k_1 \dots k_m} &= \frac{h_{k_1} \cdots h_{k_m}}{J} \sum_{ij} \epsilon_{ijk_1 \dots k_m} \partial_i \left(h_j F_{d_1 \dots d_n, j}\right) \\
&= \sum_{ij} \frac{1}{h_i h_j} \epsilon_{ijk_1 \dots k_m} \partial_i \left(h_j F_{d_1 \dots d_n, j}\right) \\
&= \sum_{ij} \frac{1}{h_i h_j} \epsilon_{ijk_1 \dots k_m} \left(\partial_i (h_j) F_{d_1 \dots d_n, j} + h_j \partial_i (F_{d_1 \dots d_n, j})\right)
\end{aligned}$$

where the last equality is preferable in practice as the derivatives of the scale factor can be computed symbolically and the computation of the derivatives of F is more efficient than the direct computation of $\partial_i \left(h_j F_{d_1 \dots d_n, j}\right)$ via finite differences. In 2-dimensional Cartesian coordinates:

```
R> f <- c("x^3 * y^2","x")
R> curl(f, var = c("x", "y"))

[1] "(1) * 1 + (x^3 * (2 * y)) * -1"
```

In 3 dimensions, for an irrotational vector field:

```
R> f <- c("x", "-y", "z")
R> curl(f, var = c("x", "y", "z"))

[1] "0" "0" "0"
```

And for tensors of vector-valued functions:

```
R> f <- matrix(c("x", "-y", "z", "x^3 * y^2", "x", "0"),
+    nrow = 2, byrow = TRUE)
R> curl(f, var = c("x", "y", "z"))

     [,1] [,2] [,3]
[1,] "0"  "0"  "0"
[2,] "0"  "0"  "(1) * 1 + (x^3 * (2 * y)) * -1"
```

The same syntax holds for functions where numerical methods are automatically applied and for arbitrary orthogonal coordinate systems as shown in the previous sections.

11.4. Laplacian

The Laplacian is a differential operator given by the divergence of the gradient of a scalar-valued function F, resulting in a scalar value giving the flux density of the gradient flow of a function. The Laplacian occurs in differential equations that describe many physical phenomena, such as electric and gravitational potentials, the diffusion equation for heat and fluid flow, wave propagation, and quantum mechanics (Wikipedia 2022k). In terms of the scale factor, the operator is written as:

$$\nabla^2 F = \frac{1}{J}\sum_i \partial_i \left(\frac{J}{h_i^2}\partial_i F\right)$$

where $J = \prod_i h_i$. When the function F is a tensor-valued function $F_{d_1,\dots,d_n}$, the `laplacian` is computed for each scalar component:

$$(\nabla^2 F)_{d_1\dots d_n} = \frac{1}{J}\sum_i \partial_i \left(\frac{J}{h_i^2}\partial_i F_{d_1\dots d_n}\right) = \frac{1}{J}\sum_i \partial_i \left(J h_i^{-2}\right)\partial_i F_{d_1\dots d_n} + J h_i^{-2}\partial_i^2 F_{d_1\dots d_n}$$

where the last equality is preferable in practice as the derivatives of the scale factor can be computed symbolically and the computation of the derivatives of F is more efficient than the direct computation of $\partial_i \left(\frac{J}{h_i^2}\partial_i F\right)$ via finite differences. In Cartesian coordinates:

```
R> f <- "x^3 + y^3 + z^3"
R> laplacian(f, var = c("x", "y", "z"))

[1] "3 * (2 * x) + 3 * (2 * y) + 3 * (2 * z)"
```

And for tensors of scalar-valued functions:

```
R> f <- array(c("x^3 + y^3 + z^3", "x^2 + y^2 + z^2", "y^2", "z * x^2"),
+    dim = c(2, 2))
R> laplacian(f, var = c("x", "y", "z"))

     [,1]                                      [,2]
[1,] "3 * (2 * x) + 3 * (2 * y) + 3 * (2 * z)" "2"
[2,] "2 + 2 + 2"                               "z * 2"
```

The same syntax holds for functions where numerical methods are automatically applied and for arbitrary orthogonal coordinate systems as shown in the previous sections.

12. Integrals

The package integrates seamlessly with **cubature** (Narasimhan *et al.* 2022) for efficient numerical integration in C. The function `integral` provides the interface for multidimensional

integrals of functions, expressions, and characters in arbitrary orthogonal coordinate systems. If the package **cubature** is not installed, the package implements a naive Monte Carlo integration by default. The function returns a `list` containing the `value` of the integral as well as other information on the estimation uncertainty. The integration bounds are specified via the argument `bounds`: a list containing the lower and upper bound for each variable. If the two bounds coincide, or if a single number is specified, the corresponding variable is not integrated and its value is fixed. For arbitrary orthogonal coordinates $q_1 \dots q_n$ the integral is computed as:

$$\int J \cdot f(q_1 \dots q_n) dq_1 \dots dq_n$$

where $J = \prod_i h_i$ is the Jacobian determinant of the transformation and is equal to the product of the scale factors $h_1 \dots h_n$.

12.1. Examples

Univariate integral $\int_0^1 x dx$:

```
R> i <- integral(f = "x", bounds = list(x = c(0, 1)))
R> i$value

[1] 0.5
```

that is equivalent to:

```
R> i <- integral(f = function(x) x, bounds = list(x = c(0, 1)))
R> i$value

[1] 0.5
```

Univariate integral $\int_0^1 yxdx|_{y=2}$:

```
R> i <- integral(f = "y * x", bounds = list(x = c(0, 1), y = 2))
R> i$value

[1] 1
```

Multivariate integral $\int_0^1 \int_o^1 yxdxdy$:

```
R> i <- integral(f = "y * x", bounds = list(x = c(0, 1), y = c(0, 1)))
R> i$value

[1] 0.25
```

Area of a circle $\int_0^{2\pi} \int_0^1 dA(r, \theta)$

```
R> i <- integral(f = 1, bounds = list(r = c(0, 1), theta = c(0, 2 * pi)),
+    coordinates = "polar")
```

```
R> i$value

[1] 3.1416
```

Volume of a sphere $\int_0^\pi \int_0^{2\pi} \int_0^1 dV(r, \theta, \phi)$

```
R> i <- integral(f = 1, bounds = list(r = c(0, 1), theta = c(0, pi),
+    phi = c(0, 2 * pi)), coordinates = "spherical")
R> i$value

[1] 4.1888
```

As a final example consider the electric potential in spherical coordinates arising from a unitary point charge $V = \frac{1}{4\pi r}$:

```
R> V <- "1 / (4 * pi * r)"
```

The electric field is determined by the gradient of the potential (Wikipedia 2022e): $E = -\nabla V$:

```
R> var <- c("r", "theta", "phi")
R> E <- -1 %prod% gradient(V, var = var, coordinates = "spherical")
```

Then, by Gauss's law (Wikipedia 2022g), the total charge enclosed within a given volume is equal to the surface integral of the electric field $q = \int E \cdot dA$ where $\cdot$ denotes the scalar product between the two vectors. In spherical coordinates, this reduces to the surface integral of the radial component of the electric field $\int E_r dA$. The following code computes this surface integral on a sphere with fixed radius $r = 1$:

```
R> i <- integral(E[1], bounds = list(r = 1, theta = c(0, pi),
+    phi = c(0, 2 * pi)), coordinates = "spherical")
R> i$value

[1] 1.0000
```

As expected $q = \int E \cdot dA = \int E_r dA = 1$, the unitary charge generating the electric potential.

13. Summary

This work has presented the **calculus** package for high dimensional numerical and symbolic calculus in R. The library applies numerical methods when working with functions or symbolic programming when working with characters or expressions. To describe multidimensional objects such as vectors, matrices, and tensors, the package uses the class '`array`' regardless of the dimension. This is done to prevent unwanted results due to operations among different classes such as '`vector`' for unidimensional objects or '`matrix`' for bidimensional objects.

The package handles multivariate numerical calculus in arbitrary dimensions and coordinates via C++ optimized functions. It achieves approximately the same accuracy for numerical

differentiation as the **numDeriv** (Gilbert and Varadhan 2019) package but significantly reduces the computational time. It supports higher order derivatives and the differentiation of possibly tensor-valued functions. Differential operators such as the gradient, divergence, curl, and Laplacian are made available in arbitrary orthogonal coordinate systems. The Einstein summing convention supports expressions involving more than two tensors and tensors with repeated indices. Besides being more flexible, the summation proves to be faster than the alternative implementation found in the **tensorA** (Van den Boogaart 2020) package for advanced tensor arithmetic with named indices. Unlike **mpoly** (Kahle 2013) and **pracma** (Borchers 2022), the package supports multidimensional Hermite polynomials and Taylor series of multivariate functions. The package integrates seamlessly with **cubature** (Narasimhan *et al.* 2022) for efficient numerical integration in C and extends the numerical integration to arbitrary orthogonal coordinate systems.

The symbolic counterpart of the numerical methods are implemented whenever possible to meet the growing needs for R to handle basic symbolic operations. Although **calculus** is not to be compared with general-purpose symbolic algebra systems, it provides, among others, symbolic high order derivatives of possibly tensor-valued functions, symbolic differential operators such as the gradient, divergence, curl, and Laplacian in arbitrary orthogonal coordinate systems, symbolic Einstein summing convention and Taylor series expansion of multivariate functions. This is done entirely in R, without depending on external computer algebra systems.

Except for **Rcpp** (Eddelbuettel and François 2011), the **calculus** package has no strict dependencies in order to provide a stable self-contained toolbox that invites re-use.

14. Computational details

The results in this paper were obtained using R 4.2.1 (R Core Team 2022) with the packages **numDeriv** 2016.8-1.1 (Gilbert and Varadhan 2019), **tensorA** 0.36.2 (Van den Boogaart 2020), **cubature** 2.0.4.4 (Narasimhan *et al.* 2022), **microbenchmark** 1.4-9 (Mersmann 2021), **calculus** 1.0.0. R itself and all packages used are available from CRAN at `https://CRAN.R-project.org/`.

Acknowledgments

This work was in part supported by Japan Science and Technology Agency CREST JPMJCR14D7, JPMJCR2115.

References

Aït-Sahalia Y (2002). “Maximum Likelihood Estimation of Discretely Sampled Diffusions: A Closed-Form Approximation Approach.” *Econometrica*, **70**(1), 223–262.

Andersen MM, Højsgaard S (2019). “**Ryacas**: A Computer Algebra System in R.” *Journal of Open Source Software*, **4**(42). `doi:10.21105/joss.01763`.

Andersen MM, Højsgaard S (2021). "**caracas**: Computer Algebra in R." *The Journal of Open Source Software*, **6**(63), 3438. doi:10.21105/joss.03438.

Borchers HW (2022). ***pracma**: Practical Numerical Math Functions.* R package version 2.3.8, URL https://CRAN.R-project.org/package=pracma.

Borchers HW, Hankin R, Sokol S (2022). *CRAN Task View: Numerical Mathematics.* Version 2022-08-08, URL https://CRAN.R-project.org/view=NumericalMathematics.

Brouste A, Fukasawa M, Hino H, Iacus SM, Kamatani K, Koike Y, Masuda H, Nomura R, Ogihara T, Shimuzu Y, Uchida M, Yoshida N (2014). "The YUIMA Project: A Computational Framework for Simulation and Inference of Stochastic Differential Equations." *Journal of Statistical Software*, **57**(4), 1–51. doi:10.18637/jss.v057.i04.

Clausen A, Sokol S (2021). ***Deriv**: Symbolic Differentiation.* R package version 4.1.3, URL https://CRAN.R-project.org/package=Deriv.

Drabinova A, Martinkova P (2017). "Detection of Differential Item Functioning with Nonlinear Regression: A Non-IRT Approach Accounting for Guessing." *Journal of Educational Measurement*, **54**(4), 498–517. doi:10.1111/jedm.12158.

Eberly D (2008). *Derivative Approximation by Finite Differences.* Magic Software, Inc. URL https://www.geometrictools.com/Documentation/FiniteDifferences.pdf.

Eddelbuettel D, François R (2011). "**Rcpp**: Seamless R and C++ Integration." *Journal of Statistical Software*, **40**(8), 1–18. doi:10.18637/jss.v040.i08.

Frankel T (2011). *The Geometry of Physics: An Introduction.* Cambridge University Press.

GeeksforGeeks (2022a). "Generate All Unique Partitions of an Integer — GeeksforGeeks | A Computer Science Portal for Geeks." URL https://www.geeksforgeeks.org/generate-unique-partitions-of-an-integer/, accessed 2022-09-05.

GeeksforGeeks (2022b). "Number of Transpositions in a Permutation — GeeksforGeeks | A Computer Science Portal for Geeks." URL https://www.geeksforgeeks.org/number-of-transpositions-in-a-permutation/, accessed 2022-09-05.

Gilbert P, Varadhan R (2019). ***numDeriv**: Accurate Numerical Derivatives.* R package version 2016.8-1.1, URL https://CRAN.R-project.org/package=numDeriv.

Guidotti E (2022). ***calculus**: High Dimensional Numerical and Symbolic Calculus.* R package version 1.0.0, URL https://CRAN.R-project.org/package=calculus.

Hladka A, Martinkova P (2020). "**difNLR**: Generalized Logistic Regression Models for DIF and DDF Detection." *The R Journal*, **12**(1), 300–323. doi:10.32614/rj-2020-014.

Kahle D (2013). "**mpoly**: Multivariate Polynomials in R." *The R Journal*, **5**(1), 162–170. doi:10.32614/rj-2013-015.

Keitt TH (2017). ***odeintr**: C++ ODE Solvers Compiled On-Demand.* R package version 1.7.1, URL https://CRAN.R-project.org/package=odeintr.

Li C, *et al.* (2013). "Maximum-Likelihood Estimation for Diffusion Processes via Closed-Form Density Expansions." *The Annals of Statistics*, **41**(3), 1350–1380.

Li J, Bien J, Wells MT (2018). "**rTensor**: An R Package for Multidimensional Array (Tensor) Unfolding, Multiplication, and Decomposition." *Journal of Statistical Software*, **87**(1), 1–31. doi:10.18637/jss.v087.i10.

Masi M (2007). "On Compressive Radial Tidal Forces." *American Journal of Physics*, **75**(2), 116–124. doi:10.1119/1.2366736.

Mersmann O (2021). ***microbenchmark**: Accurate Timing Functions.* R package version 1.4-9, URL https://CRAN.R-project.org/package=microbenchmark.

Meurer A, Smith CP, Paprocki M, Čertík O, Kirpichev SB, Rocklin M, Kumar A, Ivanov S, Moore JK, Singh S, Rathnayake T, Vig S, Granger BE, Muller RP, Bonazzi F, Gupta H, Vats S, Johansson F, Pedregosa F, Curry MJ, Terrel AR, Roučka v, Saboo A, Fernando I, Kulal S, Cimrman R, Scopatz A (2017). "**SymPy**: Symbolic Computing in Python." *PeerJ Computer Science*, **3**, e103. doi:10.7717/peerj-cs.103.

Narasimhan B, Johnson SG, Hahn T, Bouvier A, Kiêu K (2022). ***cubature**: Adaptive Multivariate Integration over Hypercubes.* R package version 2.0.4.4, URL https://CRAN.R-project.org/package=cubature.

Petzoldt T, Soetaert K (2022). *CRAN Task View: Differential Equations.* Version 2022-03-08, URL https://CRAN.R-project.org/view=DifferentialEquations.

Pinkus A, Winnitzky S, Mazur G (2020). ***Yacas**: Yet Another Computer Algebra System, Version 1.9.1.* URL http://www.yacas.org/.

Pruim R, Kaplan DT, Horton NJ (2017). "The **mosaic** Package: Helping Students to "Think with Data" Using R." *The R Journal*, **9**(1), 77–102. doi:10.32614/rj-2017-024.

R Core Team (2022). *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Sidiropoulos ND, De Lathauwer L, Fu X, Huang K, Papalexakis EE, Faloutsos C (2017). "Tensor Decomposition for Signal Processing and Machine Learning." *IEEE Transactions on Signal Processing*, **65**(13), 3551–3582. doi:10.1109/tsp.2017.2690524.

Soetaert K, Petzoldt T, Setzer RW (2010). "Solving Differential Equations in R: Package **deSolve**." *Journal of Statistical Software*, **33**(9), 1–25. doi:10.18637/jss.v033.i09.

Stroustrup B (2013). *The C++ Programming Language.* 4th edition. Addison-Wesley.

Ushey K, Allaire JJ, Tang Y (2022). ***reticulate**: Interface to Python.* R package version 1.25, URL https://CRAN.R-project.org/package=reticulate.

Van den Boogaart KG (2020). ***tensorA**: Advanced Tensor Arithmetic with Named Indices.* R package version 0.36.2, URL https://CRAN.R-project.org/package=tensorA.

Wickham H (2011). "**testthat**: Get Started with Testing." *The R Journal*, **3**(1), 5–10. doi:10.32614/rj-2011-002.

Wikipedia (2022a). "Curl (Mathematics) — Wikipedia, The Free Encyclopedia." URL https://en.wikipedia.org/wiki/Curl_(mathematics), accessed 2022-09-05.

Wikipedia (2022b). "Curvilinear Coordinates — Wikipedia, The Free Encyclopedia." URL https://en.wikipedia.org/wiki/Curvilinear_coordinates, accessed 2022-09-05.

Wikipedia (2022c). "Divergence — Wikipedia, The Free Encyclopedia." URL https://en.wikipedia.org/wiki/Divergence, accessed 2022-09-05.

Wikipedia (2022d). "Einstein Notation — Wikipedia, The Free Encyclopedia." URL https://en.wikipedia.org/wiki/Einstein_notation, accessed 2022-09-05.

Wikipedia (2022e). "Electric Potential — Wikipedia, The Free Encyclopedia." URL https://en.wikipedia.org/wiki/Electric_potential, accessed 2022-09-05.

Wikipedia (2022f). "Euler Method — Wikipedia, The Free Encyclopedia." URL https://en.wikipedia.org/wiki/Euler_method, accessed 2022-09-05.

Wikipedia (2022g). "Gauss's Law — Wikipedia, The Free Encyclopedia." URL https://en.wikipedia.org/wiki/Gauss%27s_law, accessed 2022-09-05.

Wikipedia (2022h). "Invertible Matrix — Wikipedia, The Free Encyclopedia." URL https://en.wikipedia.org/wiki/Invertible_matrix, accessed 2022-09-05.

Wikipedia (2022i). "Kronecker Delta — Wikipedia, The Free Encyclopedia." URL https://en.wikipedia.org/wiki/Kronecker_delta, accessed 2022-09-05.

Wikipedia (2022j). "Laplace Expansion — Wikipedia, The Free Encyclopedia." URL https://en.wikipedia.org/wiki/Laplace_expansion, accessed 2022-09-05.

Wikipedia (2022k). "Laplace Operator — Wikipedia, The Free Encyclopedia." URL https://en.wikipedia.org/wiki/Laplace_operator, accessed 2022-09-05.

Wikipedia (2022l). "Levi-Civita Symbol — Wikipedia, The Free Encyclopedia." URL https://en.wikipedia.org/wiki/Levi-Civita_symbol, accessed 2022-09-05.

Wikipedia (2022m). "Orthogonal Coordinates — Wikipedia, The Free Encyclopedia." URL https://en.wikipedia.org/wiki/Orthogonal_coordinates, accessed 2022-09-05.

Wikipedia (2022n). "Richardson Extrapolation — Wikipedia, The Free Encyclopedia." URL https://en.wikipedia.org/wiki/Richardson_extrapolation, accessed 2022-09-05.

Wikipedia (2022o). "Runge-Kutta Methods — Wikipedia, The Free Encyclopedia." URL https://en.wikipedia.org/wiki/Runge-Kutta_methods, accessed 2022-09-05.

Yoshida N (1992). "Asymptotic Expansion for Statistics Related to Small Diffusions." *Journal of the Japan Statistical Society, Japanese Issue*, **22**(2), 139–159. doi:10.11329/jjss1970.22.139.

Affiliation:

Emanuele Guidotti
University of Neuchâtel
Institute of Financial Analysis
Rue Abram-Louis-Breguet 2, 2000 Neuchâtel, Switzerland
and
CREST, Japan Science and Technology Agency
E-mail: emanuele.guidotti@unine.ch

Journal of Statistical Software https://www.jstatsoft.org/
published by the Foundation for Open Access Statistics https://www.foastat.org/
September 2022, Volume 104, Issue 5 *Submitted:* 2021-02-08
doi:10.18637/jss.v104.i05 *Accepted:* 2022-03-29