MultiZ: A Library for Computation of High-order Derivatives Using Multicomplex or Multidual Numbers

ANDRES M. AGUIRRE-MESA, Universidad EAFIT, Colombia.
MANUEL J. GARCIA, Angelo State University, USA.
HARRY MILLWATER, University of Texas at San Antonio, USA

Multicomplex and multidual numbers are two generalizations of complex numbers with multiple imaginary axes, useful for numerical computation of derivatives with machine precision. The similarities between multicomplex and multidual algebras allowed us to create a unified library to use either one for sensitivity analysis. This library can be used to compute arbitrary order derivates of functions of a single variable or multiple variables. The storage of matrix representations of multicomplex and multidual numbers is avoided using a combination of one-dimensional resizable arrays and an indexation method based on binary bitwise operations. To provide high computational efficiency and low memory usage, the multiplication of hypercomplex numbers up to sixth order is carried out using a hard-coded algorithm. For higher hypercomplex orders, the library uses by default a multiplication method based on binary bitwise operations. The computation of algebraic and transcendental functions is achieved using a Taylor series approximation. Fortran and Python versions were developed, and extensions to other languages are self-evident.

CCS Concepts: • **Mathematics of computing → Numerical differentiation**; **Automatic differentiation**;

Additional Key Words and Phrases: Commutative hypercomplex, multicomplex, multidual, hyperdual, high order derivatives

ACM Reference format:
Andres M. Aguirre-Mesa, Manuel J. Garcia, and Harry Millwater. 2020. MultiZ: A Library for Computation of High-order Derivatives Using Multicomplex or Multidual Numbers. *ACM Trans. Math. Softw.* 46, 3, Article 23 (July 2020), 30 pages.
https://doi.org/10.1145/3378538

1 INTRODUCTION

The advent of the use of hypercomplex algebras, multicomplex and multidual (also called hyperdual), to compute high-accuracy derivatives of arbitrary order has initiated the need for easy-to-use

Andres M. Aguirre-Mesa: Also with University of Texas at San Antonio
Manuel J. Garcia: Also with Universidad EAFIT
This work was funded in part by the United States Department of Defense (DoD grant W911NF-15-1-0456), Universidad EAFIT and the COLCIENCIAS' Scholarship Program No. 6172. This work received computational support from UTSA's HPC cluster SHAMU, operated by the Office of Information Technology.
Authors' addresses: A. M. Aguirre-Mesa, Universidad EAFIT, Medellin, Colombia; emails: aaguirr2@eafit.edu.co, Andres.AguirreMesa@my.utsa.edu; M. J. Garcia, Angelo State University, Texas; email: Manuel.Garcia@angelo.edu; H. Millwater (corresponding author), University of Texas at San Antonio, Texas; email: Harry.Millwater@utsa.edu.

0098-3500/2020/07-ART23 $15.00
https://doi.org/10.1145/3378538

tools to convert existing or new numerical codes to use hypercomplex variables and hypercomplex algebras. Since current computer languages do not support hypercomplex algebras, complex variables being the exception, there is a large effort required by the developer to implement the needed mathematical operations and functions.

To address this need, we have developed a hypercomplex library, MultiZ, that provides the necessary support. The support provided includes: mathematical operations using operator overloading (addition, subtraction, multiplication, division, conjugate) and mathematical functions (sine, cosine, exponential, log, integer, and fractional power), and vector and matrix storage functions.

MultiZ offers two alternatives to perform hypercomplex multiplication. For hypercomplex numbers up to sixth order a hard-coded algorithm is used, which provides high computational efficiency and low memory usage. For higher orders of hypercomplex numbers, MultiZ uses algorithms based on binary bitwise operations. Nevertheless, the user can expand the hard-coded capabilities of MultiZ using code generators provided with the source code. Fortran and Python languages are supported. Extensions to other languages are self-evident.

The use of complex arithmetics for the numerical calculation of derivatives of analytical functions was first introduced by Lyness and Moler [15]. The formulation was later simplified by Squire and Trapp to create the complex Taylor series expansion method (CTSE), also known as complex step method [20]. This method has been applied for computation of first-order derivatives in different areas, such as Computational Fluid Dynamics (CFD) for the computation of derivatives of lift and drag coefficients from airfoil simulations [3]; in structural optimization for the eigenvalue and eigenvector sensitivity analysis [25]; for shape sensitivity of finite element models [24], and for fracture mechanics [18], among others.

Lantoine et al. proposed an extension of the CTSE method for high-order derivatives using multicomplex numbers and applied the method to aerospace trajectory sensitivity analysis [14]. The Multicomplex Taylor Series Expansion method has been applied in other areas such as structural dynamics for the computation of second-order time-dependent derivatives of dynamic systems [9], and for high-order probabilistic sensitivity calculations [10].

Multidual numbers have been applied in multiple fields such as CFD for the computation of first- and second-order derivatives of the lift and drag coefficients of subsonic, transonic and supersonic airfoil simulations [7]; in multibody kinematics to obtain derivatives of the kinematic variables of a body using multidual transformation matrices [4]; and in elasto-plasticity, where fourth-order multidual numbers were used for the automatic computation of stresses, tangent moduli and other internal variables [21].

1.1 Multicomplex and multidual numbers

There are several number systems that expand the concept of complex numbers, adding not one, but multiple imaginary units to the real numbers. These number systems belong to the set of hypercomplex numbers, and some examples are: complex numbers, dual numbers, double numbers, quaternions, octonions, sedenions, and tessarines [13].

The scope of this research is limited to multicomplex and multidual numbers, since their algebras are similar and commutative, in contrast to quaternions and other high-order generalizations, such as Cayley-Dickson algebras. Furthermore, multicomplex and multidual numbers have been extensively applied for the computation of high-order derivatives.

The definition of multicomplex numbers and multidual numbers builds upon the concepts of complex numbers and dual numbers, respectively. Similar to complex numbers, dual numbers are also defined by an expression of the form $z^* = a + b\,\epsilon$, with $a, b \in \mathbb{R}$, and ϵ, the Greek letter epsilon, is the imaginary unit of dual numbers. However, contrary to complex numbers, where $i^2 = -1$, $\epsilon^2 = 0$ for dual numbers [13].

Multicomplex numbers of arbitrary order n can be defined using the following recursive definition [19]:

$$\mathbb{C}_n = \{z^* : z^* = a^* + b^*\, i_n,\ a^*, b^* \in \mathbb{C}_{n-1}\}, \tag{1}$$

where $\mathbb{C}_0 = \mathbb{R}$, and $\mathbb{C}_1$ is the set of regular complex numbers. Multicomplex multiplication is commutative, which implies

$$i_m i_p = \begin{cases} -1 & \forall\, m = p, \\ i_p i_m & \forall\, m \neq p, \end{cases} \qquad p, q \in \mathbb{N}. \tag{2}$$

A bicomplex number, for example, can be expressed as the sum of two complex numbers x_1^* and x_2^*, one of them multiplied by an imaginary unit i_2, resulting in a number that is composed by four real coefficients.

$$x_1^* = a + b\, i_1, \quad x_2^* = c + d\, i_1, \quad x_1^*, x_2^* \in \mathbb{C}_1, \quad a, b, c, d \in \mathbb{R}. \tag{3}$$

$$\begin{aligned} z^* \in \mathbb{C}_2, \quad z^* &= x_1^* + x_2^*\, i_2 = (a + b\, i_1) + (c + d\, i_1) i_2, \\ &= a + b\, i_1 + c\, i_2 + d\, i_1 i_2. \end{aligned} \tag{4}$$

Similarly to multicomplex, multidual numbers of arbitrary order are defined using a recursive rule based on dual numbers.

$$\mathbb{D}_n = \{z^* : z^* = a^* + b^*\, i_n,\ a^*, b^* \in \mathbb{D}_{n-1}\}, \tag{5}$$

$$\epsilon_m \epsilon_p = \begin{cases} 0 & \forall m = p, \\ \epsilon_p \epsilon_m & \forall m \neq p, \end{cases} \qquad p, q \in \mathbb{N}, \tag{6}$$

where $\mathbb{D}_0$ is the set of real numbers, and $\mathbb{D}_1$ the set of dual numbers.

1.2 Matrix representation of hypercomplex numbers

A convenient way to evaluate multicomplex expressions is by the use of matrix functions and operations. This is possible due to a property called algebra isomorphism, which allows one to represent a multicomplex number in a matrix form called the Cauchy-Riemann matrix [19] and to compute multicomplex arithmetic operations using equivalent matrix operations [14].

The matrix form of a multicomplex number is not unique. To comply with the algebra isomorphism property is sufficient to ensure that matrices can reproduce the addition and the multiplication of multicomplex numbers. The following are two matrix representations of the multicomplex number $z^* = a^* + b^*\, i_n$, where $z^* \in \mathbb{C}_n$, and $a^*, b^* \in \mathbb{C}_{n-1}$:

$$\boldsymbol{Z}_1 = \begin{bmatrix} \boldsymbol{A} & -\boldsymbol{B} \\ \boldsymbol{B} & \boldsymbol{A} \end{bmatrix}, \qquad \boldsymbol{Z}_2 = \begin{bmatrix} \boldsymbol{A} & \boldsymbol{B} \\ -\boldsymbol{B} & \boldsymbol{A} \end{bmatrix}, \tag{7}$$

where $\boldsymbol{A}$ and $\boldsymbol{B}$ are the matrix representations of multicomplex numbers a^* and b^*, respectively. The first matrix form is convenient to express multiplication of multicomplex numbers as a matrix-vector multiplication, while the second form allows one to express it as vector-matrix multiplication. In the current study, only the first form will be used.

For a bicomplex number $z^* = x_1^* + x_2^* i_2$, where $x_1^* = a + b\, i_1$ and $x_2^* = c + d\, i_1$, the Cauchy-Riemann matrix is

$$\boldsymbol{Z} = \begin{bmatrix} \boldsymbol{X}_1 & -\boldsymbol{X}_2 \\ \boldsymbol{X}_2 & \boldsymbol{X}_1 \end{bmatrix} = \begin{bmatrix} a & -b & -c & d \\ b & a & -d & -c \\ c & -d & a & -b \\ d & c & b & a \end{bmatrix}. \tag{8}$$

For a bidual number $z^* = x_1^* + x_2^* \epsilon_2$, where $x_1^* = a + b\,\epsilon_1$ and $x_2^* = c + d\,\epsilon_1$, the Cauchy-Riemann matrix is

$$Z = \begin{bmatrix} X_1 & 0 \\ X_2 & X_1 \end{bmatrix} = \begin{bmatrix} a & 0 & 0 & 0 \\ b & a & 0 & 0 \\ c & 0 & a & 0 \\ d & c & b & a \end{bmatrix}. \tag{9}$$

Generalizing for multiduals of arbitrary order n, a multidual number $z^* = a^* + b^*\,\epsilon_n$, $z^* \in \mathbb{D}_n$, $a^*, b^* \in \mathbb{D}_{n-1}$, can be represented by the following block matrix:

$$Z = \begin{bmatrix} \boldsymbol{A} & \boldsymbol{0} \\ \boldsymbol{B} & \boldsymbol{A} \end{bmatrix}, \tag{10}$$

where $\boldsymbol{A}$ and $\boldsymbol{B}$ are the matrix representations of multidual numbers a^* and b^*, respectively.

1.3 Computation of high-order derivatives

Multicomplex and multidual numbers are particularly useful for the computation of highly accurate derivatives of real-valued functions.

Lantoine et al. [14] provided an expression for the computation of the nth-order derivative of a single variable real-valued function, $f^{(n)}(x)$, obtained as the imaginary part of the holomorphic function $f(z^*)$ when evaluated using a multicomplex of the form $z^* = x + h\,i_1 + \cdots + h\,i_n$.

$$f^{(n)}(x) = \frac{1}{h^n}\mathrm{Im}_{1\ldots n}\left[f\left(x + h\,i_1 + \cdots + h\,i_n\right)\right] + O(h^2), \tag{11}$$

where $\mathrm{Im}_{1\ldots n}$ is the imaginary part associated with $i_1 i_2 \cdots i_n$, x is real-valued, and h is a very small positive real number (Garza and Millwater suggest 10^{-20} times the smallest input parameter of the problem [9]). In the case of multidual numbers, proposed by Fike and Alonso under the name hyperduals, the computation of derivatives is performed in an identical manner to multicomplex, using non-real parts of the evaluated function [7]. Although not explicitly stated by the authors, an expression for obtaining nth-order derivatives using multiduals is:

$$f^{(n)}(x) = \frac{1}{h^n}\mathrm{Im}_{1\ldots n}\left[f\left(x + h\,\epsilon_1 + \cdots + h\,\epsilon_n\right)\right]. \tag{12}$$

Notice that, unlike the expression used for multicomplex numbers, Equation (12) is not a second-order approximation, but a mathematically exact expression. Therefore, derivatives computed using multidual numbers are independent of the step size h.

Consider the computation of the second-order derivative of $f(x) = x^3$. Evaluating a bicomplex version of the function at $z^* = x + h\,(i_1 + i_2)$,

$$\begin{aligned} f(z^*) &= \left[x + h\,(i_1 + i_2)\right]^3, \\ &= x^3 + 3x^2 h\,(i_1 + i_2) + 3xh^2\,(i_1 + i_2)^2 + h^3\,(i_1 + i_2)^3, \\ &= x^3 + 3x^2 h\,(i_1 + i_2) + 6xh^2\,(i_1 i_2 - 1) - 4h^3(i_1 + i_2). \end{aligned} \tag{13}$$

Reorganizing in terms of real and imaginary parts, $f(z^*)$ can be written as

$$f(z^*) = (x^3 - 6xh^2) + (3x^2 h - 4h^3)i_1 + (3x^2 h - 4h^3)i_2 + 6xh^2 i_1 i_2. \tag{14}$$

Using Equation (11), the second derivative of x^3 is

$$f''(x) = \frac{1}{h^2}\mathrm{Im}_{1\,2} f(z^*) = 6x. \tag{15}$$

In the multidual case, the bidual version of the function is evaluated at $z^* = x + h\,(\epsilon_1 + \epsilon_2)$

$$\begin{aligned} f(z^*) &= [x + h\,(\epsilon_1 + \epsilon_2)]^3\,, \\ &= x^3 + 3x^2 h\,(\epsilon_1 + \epsilon_2) + 3xh^2\,(\epsilon_1 + \epsilon_2)^2 + h^3 \cancelto{0}{(\epsilon_1 + \epsilon_2)^3}, \\ &= x^3 + 3x^2 h\,(\epsilon_1 + \epsilon_2) + 6xh^2 \epsilon_1 \epsilon_2. \end{aligned} \tag{16}$$

Using Equation (12), the second-order derivative is again $6x$.

2 MULTIZ LIBRARY

The objective of the library is to make hypercomplex numbers easy to use for the computation of high-order derivatives, regardless of the complexity of the function or program. With that objective in mind, the development of the library was based on the following ideals:

(i) Hypercomplex numbers should look and feel as any other built-in numeric variable type. In particular, they should have a similar syntax to complex variables for input and output of the imaginary parts.

(ii) Hypercomplex data structures should adapt dynamically to the number of imaginary parts required by the user. The library should keep track of the order of the hypercomplex number, not the user.

(iii) The hypercomplex library should be able to compute any order of derivative required by the user.

As an introductory example, consider the numerical computation of the third-order derivative of a real-valued function; for instance, the composite function proposed by Martins et al. [16]:

$$f(x) = \frac{e^x}{\sqrt{\sin^3 x + \cos^3 x}}. \tag{17}$$

To compute the third-order derivative of $f(x)$ at $x_0 = 0.5$, a multicomplex version of the function is evaluated at $x^* = x_0 + h\,(i_1 + i_2 + i_3)$, where h is small enough to ensure second-order accuracy of the required derivative ($h < 10^{-10}$).

Using overloading of operators and functions in Fortran, the multicomplex input number x^*, as well as the function $f(x^*)$, can be coded as follows:

```
x = x0 + h*(im(1) + im(2) + im(3))
y = exp(x)/sqrt(sin(x)**3 + cos(x)**3),
```

where `im(j)` stands for the imaginary unit i_j (see Section 2.4).

Finally, the third-order derivative can be computed from the imaginary part associated with the imaginary units $i_1 i_2 i_3$. This is accomplished using the following syntax:

```
d3 = aimag(y, [1,2,3])/h**3.
```

The result of the previous example is −9.3319100381987052, whose relative error with respect to the analytical solution is 1.33×10^{-15}.

Since the value of the step size h is immaterial for computing derivatives using multidual numbers, a value of $h = 1$ can be assumed. Therefore, a multidual version of the previous example would be:

```
x  = x0 + (eps(1) + eps(2) + eps(3))
y  = exp(x)/sqrt(sin(x)**3 + cos(x)**3).
d3 = aimag(y, [1,2,3])
```

The result of the multidual case is −9.3319100381986804, which is equal to the multicomplex case to 13 decimal places and has an identical relative error of 1.33×10^{-15}.

Tables 1 to 6 contain a summary of the functions and operators supported by MultiZ.

Table 1. Data Types of MultiZ

Name	Description
`mcomplex`	Multicomplex number
`mdual`	Multidual number
`mcxvec`	Multicomplex vector
`mduvec`	Multidual vector
`mcxmat`	Multicomplex matrix
`mdumat`	Multidual matrix

Table 2. Functions to Handle Hypercomplex Numbers

Name	Description
mcomplex	Create a multicomplex number by providing the coefficients.
mdual	Create a multidual number by providing the coefficients.
im	Create a unit value imaginary part for multicomplex numbers.
eps	Create a unit value imaginary part for multidual numbers
real	Extract the real part of a number or array.
aimag	Extract an imaginary part of a number or array.
conj_im	Compute the conjugate of a hypercomplex with respect to the specified imaginary unit.

Table 3. Operators and Elementary Functions Overloaded

Op.	Description	Func.	Description
`+`	Addition	`sin`	Sine
`-`	Subtraction and negation	`cos`	Cosine
`*`	Multiplication	`exp`	Exponential
`/`	Division and reciprocal	`log`	Natural log.
`**`	Power (integer and fraction)	`sqrt`	Square root

Table 4. Linear Algebra Operators

Name	Description
`transpose`	Transpose of a hypercomplex matrix
`matmul`	Matrix-matrix and matrix-vector multiplication
`dot_nc`	Dot product without conjugation

2.1 Hypercomplex variable types

MultiZ uses extended derived types in Fortran [17] to define multicomplex and multidual numbers, called `mcomplex` and `mdual`, respectively. Both inherit from a base derived type called `mnumber`, whose components are a one-dimensional allocatable array of real numbers, the number of coefficients, and the order of the hypercomplex number. The base derived type `mnumber` is defined as:

```
TYPE mnumber
  REAL(mrealk), ALLOCATABLE  ::  coeffs(:)
  INTEGER                    ::  order
  INTEGER                    ::  numcoeffs
END TYPE mnumber
```

Table 5. Functions to Handle Hypercomplex Arrays

Name	Description
mallocate	Allocate space for arrays of hypercomplex numbers
mchange_order	Increase or decrease number of imaginary coefficients.
mget	Extract a single hypercomplex number from an array element
mset	Assign a hypercomplex number to an array element
mget_slice	Extract a slice (subarray)
mset_slice	Assign values to a slice (subarray)
mget_row	Extract a row from a matrix
mget_col	Extract a column from a matrix
shape	Get the shape of an array. Number of elements per dimension
size	Get total number of elements of the array

Table 6. Support for Cauchy-Riemann Matrices

Name	Description
mto_cr	Convert hypercomplex data structures (numbers, vectors, matrices) to Cauchy-Riemann compatible arrays.
mcr_to_mcomplex	Convert CR matrix to multicomplex number
mcr_to_mdual	Convert CR matrix to multidual number
mcr_to_mcxvec	Convert CR compatible vector to multicomplex vector
mcr_to_mduvec	Convert CR compatible vector to multidual vector
mcr_to_mcxmat	Convert CR block matrix to multicomplex matrix
mcr_to_mdumat	Convert CR block matrix to multidual matrix
get_cr	Compute a CR matrix element at the specified position from a hypercomplex number

The precision of the floating point numbers used by MultiZ (4, 8, or 16 bytes) is controlled by the module parameter `mrealk`, which can be modified by the user from the source code of the library. Additionally, `mrealk` can be used to enforce floating point precision consistency in the user's own code.

The library also provides its own types for hypercomplex vectors and matrices: the derived types `mcxvec` and `mduvec`, which inherit from the base type `mvector`, and the derived types `mcxmat` and `mdumat`, which inherit from the base type `mmatrix`:

```
TYPE mvector
  REAL(mrealk), ALLOCATABLE ::  coeffs(:,:)
  INTEGER                   ::  order
  INTEGER                   ::  numcoeffs  .
  INTEGER                   ::  mshape(1)
  INTEGER                   ::  msize
END TYPE mvector

TYPE mmatrix
  REAL(mrealk), ALLOCATABLE ::  coeffs(:,:,:)
  INTEGER                   ::  order
  INTEGER                   ::  numcoeffs   .
  INTEGER                   ::  mshape(2)
  INTEGER                   ::  msize
END TYPE mmatrix
```

Similarly to the hypercomplex number types, the array types contain an allocatable array called `coeffs`, which stores real and imaginary coefficients. The coefficients array is 2D for vector types and 3D for matrix types. The first index entry of the array is associated with the imaginary parts.

The use of a single array to store all the coefficients of a vector or matrix type is advantageous to rapidly extract any imaginary part required and to program vectorized procedures to operate on hypercomplex numbers.

The library supports multicomplex algebra and multidual algebra, but not an algebra that combines multicomplex and multidual numbers. Therefore, operations that combine both types of numbers are not supported.

2.2 Creation of hypercomplex numbers

To create a multicomplex or a multidual number, the derived type has to be declared at the beginning of the Fortran code, using one of the following statements:

```
TYPE(mcomplex) :: a  ! Multicomplex number.
TYPE(mdual)    :: b  ! Multidual number.
```

After the type is declared, the coefficients and order of a hypercomplex number can be defined in four different ways:

(i) Similar to complex numbers, by addition of real and imaginary parts.

```
a = x0 + h*im(1) + h*im(2)            ! Bicomplex number.
b = x0 + eps(1) + eps(2)              ! Bidual number.
```

(ii) By direct input of the real valued coefficients using constructors.

```
a = mcomplex([5.d0, 1.d-10, 1.d-10])  ! Bicomplex number.
b = mdual([5.d0, 1.d0, 1.d0])         ! Bidual number.
```

(iii) By allocation of coefficients based on a specified order. All coefficients are initialized to zero.

```
CALL mzero(a, order=2)  ! Works for both types of numbers.
```

(iv) By direct modification of the internal coefficients array `coeffs`. This option is recommended for advanced users only, since `coeffs` is a standard Fortran array and no bounds verification or memory reallocation is performed.

2.3 Creation of hypercomplex arrays

Hypercomplex valued vectors or matrices are declared using the following statements:

```
TYPE(mcxvec) :: u  ! Multicomplex vector.
TYPE(mduvec) :: v  ! Multidual vector.
TYPE(mcxmat) :: m  ! Multicomplex matrix.
TYPE(mdumat) :: p  ! Multidual matrix.
```

The current version of MultiZ only supports allocatable arrays. The memory allocation is defined using the subroutine interface `mallocate`, whose required arguments are the array, the order, and the dimensions. The statement works for all hypercomplex array types:

```
CALL mallocate(u, order=1, d1=2)
CALL mallocate(m, order=1, d1=2, d2=2).
```

MultiZ offers multiple ways to assign values to arrays:

(i) Initialize all elements with a real or hypercomplex number using the equal sign. If the hypercomplex order of the value is greater than the order declared for the array, its memory will be reallocated automatically to store the additional imaginary coefficients:

```
v = 1.d0 + im(1)   ! Init. vector to complex.
p = 1.d0 + eps(1)  ! Init. matrix with dual.
```

In the previous example, all elements of arrays $\boldsymbol{v}$ and $\boldsymbol{p}$ are defined as follows:

$$v_k = 1 + i, \quad \forall k, \tag{18}$$

$$p_{rs} = 1 + \epsilon, \quad \forall r, s. \tag{19}$$

(ii) Assign each element with a real or hypercomplex value using the subroutine `mset`. The required arguments are the array, the position, and the value:

```
CALL mset(v, 1, x)      ! Set v[1]   = x
CALL mset(p, 2, 1, y)   ! Set p[2,1] = y
```

(iii) Assign to a slice (sub-array) of the array using the subroutine `mset_slice`. The value can be either real, hypercomplex, a real array, or a hypercomplex array. The array and the value are required arguments, while the bounds of the array and the strides are optional. The following are the names of the optional arguments.

Arg.	Description	Arg.	Description
`lr`	Lower bound for rows	`lc`	Lower bound for columns
`ur`	Upper bound for rows	`uc`	Upper bound for columns
`sr`	Stride for rows	`sc`	Stride for columns

If any of the bounds of the slice are not provided, they are assumed equal to the bounds of the array. Additionally, if the strides are not provided, they are assumed equal to one. The optional arguments for vectors slices are: `lr`, `ur`, and `sr`.

```
CALL mset_slice(v, ur=4, val=x)  ! Set v[:4] = x
! Set p[2:3,2:3] = y                          .
CALL mset_slice(p, lr=2, ur=3, lc=2, uc=3, val=y)
```

(iv) Make a hypercomplex array equal to a real-valued array of the same dimensions. This operation only requires the use of the equal sign. The right-hand side array can be either a real array or a hypercomplex array of the same type. If the hypercomplex orders of the involved arrays are different, the memory of the left-hand side array will be reallocated if necessary.

(v) By direct modification of the internal coefficients array `coeffs`. This option is recommended for advanced users only: Since `coeffs` is a standard Fortran array, no bounds verification or memory reallocation is performed:

```
v%coeffs[1,3]   = 2.d0      ! Real part of v[3] = 2.
p%coeffs[:,2,1] = y%coeffs  ! p[2,1] = y            .
```

2.4 Indexation of imaginary parts

Functions to extract real and imaginary parts are similar to those used for built-in complex numbers of the Fortran programming language, and they work the same for multicomplex and multidual.

The real part of a hypercomplex number can be extracted using the following function:

```
a = real(x)  ! Works for multicomplex and multiduals.
```

Imaginary parts of a hyperdual number can be extracted using the `aimag` function, but an additional argument is required to specify which of the multiple imaginary parts to extract. For imaginary parts with a single imaginary unit, such as i_1 or ϵ_1, the additional argument is a single integer value. However, to extract the imaginary part with multiple imaginary units, such as $i_1 i_2$ or $\epsilon_1 \epsilon_2$, an array of integers needs to be provided:

Table 7. Relationship between Real Coefficients Index and Presence of Imaginary Units for a Tricomplex Number

Terms of a $\mathbb{C}_3$ number	Index k of the real coeff. x_k	Presence of imag. units i_3	i_2	i_1
x_0	$0 = (000)_2$	0	0	0
$x_1\ i_1$	$1 = (001)_2$	0	0	1
$x_2\ i_2$	$2 = (010)_2$	0	1	0
$x_3\ i_1 i_2$	$3 = (011)_2$	0	1	1
$x_4\ i_3$	$4 = (100)_2$	1	0	0
$x_5\ i_1 i_3$	$5 = (101)_2$	1	0	1
$x_6\ i_2 i_3$	$6 = (110)_2$	1	1	0
$x_7\ i_1 i_2 i_3$	$7 = (111)_2$	1	1	1

```
! Extract imaginary parts. Works for all
! hypercomplex data types.
b = aimag(x,1)      ! From i_1 or e_1.
c = aimag(x,[1,2])  ! From i_1 i_2 or e_1 e_2
```

The mathematical rule that allows multiZ to handle hypercomplex numbers as a one-dimensional array, mapping each real coefficient with a unique imaginary part, is based on the binary number system:

(i) A hypercomplex number of order n contains 2^n real coefficients, which is the same quantity of possible binary numbers that can be formed with n digits. (ii) When hypercomplex numbers are formed using recursive definitions (see Equations (1) and (5)), the presence or absence of imaginary units correlates with different numbers of the binary system.

The relationship between hypercomplex numbers and the binary number system is illustrated using an example. Consider the recursive definition of a multicomplex number, shown in Equation (1), to define a complex z^*, a bicomplex b^*, and a tricomplex t^*, all of them expressed in terms of their real coeficients. The number of real coefficients in each case will be 2^n, where n is the order of the multicomplex number:

$$z^* \in \mathbb{C}_1,\ z^* = x_0 + x_1\, i_1, \tag{20}$$

$$b^* \in \mathbb{C}_2,\ b^* = x_0 + x_1\, i_1 + x_2\, i_2 + x_3\, i_1 i_2, \tag{21}$$

$$t^* \in \mathbb{C}_3,\ t^* = x_0 + x_1\, i_1 + x_2\, i_2 + x_3\, i_1 i_2 + x_4\, i_3 + x_5\, i_1 i_3 + x_6\, i_2 i_3 + x_7\, i_1 i_2 i_3. \tag{22}$$

Using a zero-based numbering for the indices of the real coefficients, and converting them to binary, it can be noticed that their binary representation correlates with the presence or absence of the imaginary units of each term. A detailed comparison for a tricomplex number is shown in Table 7. This correlation holds also for multiduals of arbitrary order.

2.5 Indexation of Cauchy-Riemann matrices

As shown in Section 1.2, the construction of a matrix representation of a hypercomplex number requires multiple copies of its imaginary parts. In general, the matrix representation is necessary for certain hypercomplex operations like multiplication, where at least one of the operands has to be represented by a matrix. However, MultiZ includes hard-coded algorithms, based on a

Table 8. Memory Cost Comparison of Using 1D and 2D Data Structures for Hypercomplex Numbers

Order n	Multidual CR matrix (3^n)	Multicomplex CR matrix (4^n)	Hypercomplex numbers as 1D arrays (2^n)
1	3	4	2
2	9	16	4
3	27	64	8
4	81	256	16
5	243	1,024	32

matrix-vector multiplication, to efficiently compute the multiplication of hypercomplex numbers without storing the matrix (see Section 2.7).

MultiZ includes functions to build the complete Cauchy-Riemann matrix and to extract only the element of the Cauchy-Riemann matrix required by the user. For example, for computing the Cauchy-Riemann matrix of a hypercomplex number x^*, and to extract the element at the position $(2, 3)$ of the Cauchy-Riemann of x^* by indexation, the following commands are used:

```
m    = mto_cr(x)      ! Compute CR matrix of x.
m_23 = x%get_cr(2,3)  ! Compute element at (2,3) only.
```

The library also includes functions to convert a Cauchy-Riemann matrix into a multicomplex or a multidual number:

```
x = mfold_mcomplex(m)  ! CR matrix to multicomplex number.
y = mfold_mdual(m)     ! CR matrix to multidual number.
```

The use of hypercomplex numbers of order n for sensitivity analysis implies a conversion of the variables into hypercomplex variable types, which increases the memory use of each variable by a factor of 2^n, i.e., the memory consumption grows exponentially with respect to the hypercomplex order used if a vector-like data structure is used for hypercomplex numbers. If the data structure is based on the Cauchy-Riemann matrix, the memory used by each hypercomplex variable would increase by a factor 3^n for multidual numbers, assuming the use of sparse matrices, and 4^n for multicomplex numbers. The memory use is therefore drastically reduced when hypercomplex numbers are stored in a vector-like data structure, as shown in Table 8.

To reduce the memory consumption associated with the construction of Cauchy-Riemann matrices, the binary indexation method was used to create map functions between hypercomplex numbers of arbitrary order $\mathbb{C}_n$ and their matrix representation.

Consider for example the bicomplex number $a^* = a_0 + a_1\, i_1 + a_2\, i_2 + a_3\, i_1 i_2$ and its Cauchy-Riemann matrix,

$$\boldsymbol{A} = \begin{bmatrix} a_0 & -a_1 & -a_2 & a_3 \\ a_1 & a_0 & -a_3 & -a_2 \\ a_2 & -a_3 & a_0 & -a_1 \\ a_3 & a_2 & a_1 & a_0 \end{bmatrix}. \tag{23}$$

Let r represent the index associated with the real-valued coefficients of a^*, and p, q the indices associated with the rows and columns of matrix $\boldsymbol{A}$. The element $\boldsymbol{A}_{pq}$ is a multiple of the coefficient a_r, that is

$$\boldsymbol{A}_{pq} = s(p, q)\, a_r, \tag{24}$$

where the multiplier s takes the values $\{-1, 1\}$ for multicomplex numbers and $\{0, 1\}$ for multidual numbers. The relation between the coefficient index r and the matrix indices p, q is given by the XOR operation between p and q,

$$r = p \oplus q, \tag{25}$$

while the definition of the multiplier s depends on the hypercomplex algebra. In the multicomplex case, the multiplier s is the sign, and it is determined by

$$s(p, q) = \begin{cases} -1, & \text{if bit count of } \neg p \wedge q \text{ is an odd number,} \\ 1 & \text{otherwise.} \end{cases} \tag{26}$$

where bit count means to count the number of one bits in the binary representation of a number, $\neg$ is the NOT operator, and $\wedge$ is the AND operator. In the multidual case, the multiplier s is determined by

$$s(p, q) = \begin{cases} 1, & \text{if } \neg p \wedge q = 0, \\ 0 & \text{otherwise.} \end{cases} \tag{27}$$

Equations (25) to (27) work for hypercomplex numbers of arbitrary order.

Consider the particular case of computing the element in the position $(2, 3)$ of the Cauchy-Riemann matrix of the bicomplex number a^*, assuming zero-based numbering. Using Equation (25) to determine the required coefficient of a^* in $\boldsymbol{A}_{pq} = s\, a_r$,

$$r = p \oplus q = 2 \oplus 3 = (10)_2 \oplus (11)_2 = (01)_2 = 1. \tag{28}$$

Hence, $\boldsymbol{A}_{23} = s\, a_1$. Now to determine the multiplier s, using Equation (26),

$$\neg p \wedge q = \neg 2 \wedge 3 = \neg(10)_2 \wedge (11)_2 = (01)_2 \wedge (11)_2 = (01)_2 = 1. \tag{29}$$

The number of set bits in the binary representation of the number 1 is equal to 1, an odd number; therefore, $s = -1$ and $\boldsymbol{A}_{23} = -a_1$. If the multidual case is considered, the result in Equation (29) is different from zero, then $s = 0$ according to Equation (27); therefore $\boldsymbol{A}_{23} = 0$.

2.6 Addition and subtraction

Addition and subtraction of hypercomplex numbers are accomplished by adding or subtracting corresponding imaginary parts. If operands are of the same hypercomplex order, i.e., they have the same number of real coefficients, addition or subtraction works analogously to the addition of vectors.

If the hypercomplex numbers have different hypercomplex orders, the missing imaginary parts in the lowest order terms are set to zero, and the result will have an order equal to the maximum order of the operands.

MultiZ supports the addition or subtraction of hypercomplex numbers of any order, even with real number variable types, since $\mathbb{R} \subset \mathbb{C}_n$. Addition and subtraction are accomplished through operator overloading.

2.7 Multiplication

MultiZ uses a hard-coded algorithm for multiplication of hypercomplex numbers up to sixth order. As shown in Section 3.2, the hard-coded algorithm is the most efficient method to multiply two hypercomplex numbers. Additionally, this method does not require the storage of the Cauchy-Riemann matrix of any of the operands. In case the multiplication of hypercomplex numbers of order higher than six is required, the multiplication is performed using an algorithm based on the indexation of the Cauchy-Riemann matrix of one of the operands. This method sacrifices the performance of the multiplication operation, but keeps an efficient use of the memory resources by avoiding the storage of Cauchy-Riemann matrices.

```
p0 = a0*b0
p1 = a1*b0 + a0*b1
p2 = a2*b0 + a0*b2
p3 = a3*b0 + a2*b1 + a1*b2 + a0*b3
```

Listing 1. Example of hard-code for multiplication of two second-order multidual numbers $p^* = a^* \times b^*$

As discussed in the introduction, the multiplication of two hypercomplex numbers can be performed using matrix-matrix multiplication of their Cauchy-Riemann matrices. However, as shown in Appendix A, the product of two hypercomplex numbers can also be computed using matrix-vector multiplication.

Consider the multiplication of two hypercomplex numbers $x^*, y^* \in \mathbb{C}_n$, with a total number of imaginary parts 2^n, and the Cauchy-Riemann matrix of each one has dimensions $2^n \times 2^n$. Computation through matrix-matrix multiplication requires 2^{3n} floating point operations, while a matrix-vector approach requires 2^{2n} operations. The formulas to index the Cauchy-Riemann matrix were designed to perform hypercomplex multiplication as a matrix-vector multiplication without allocating memory for the matrix. However, the indexation of all elements of a Cauchy-Riemann matrix adds an overhead to the computation time as a tradeoff of the reduction in memory usage.

Since formulas for the indexation of Cauchy-Riemann matrices are general for hypercomplex numbers of arbitrary order, a strategy to perform hypercomplex multiplication efficiently, and with low memory usage, is to use indexation formulas to generate code that directly operates on the real-valued coefficients. An example of multiplication of second-order multidual numbers using hard-code is shown in Listing 1.

2.8 Division

MultiZ computes the division of two hypercomplex numbers using conjugate numbers. Consider a^* and b^* bicomplex numbers and x^* equal to b^* divided by a^*:

$$x^* = \frac{(b_0 + b_1\, i_1) + (b_2 + b_3\, i_1)\, i_2}{(a_0 + a_1\, i_1) + (a_2 + a_3\, i_1)\, i_2}. \tag{30}$$

The i_2-conjugate of a^* is $\text{conj}_2(a^*) = (a_0 + a_1\, i_1) - (a_2 + a_3\, i_1)\, i_2$. Multiplying the numerator and the denominator by $\text{conj}_2(a^*)$, the following is obtained:

$$\begin{aligned} x^* &= \frac{b^* \,\text{conj}_2(a^*)}{a^* \,\text{conj}_2(a^*)} = \frac{b^* \,\text{conj}_2(a^*)}{(a_0 + a_1 i_1)^2 - (a_2 + a_3 i_1)^2\,(-1)}, \\ &= \frac{b^* \,\text{conj}_2(a^*)}{\left(a_0^2 - a_1^2 + a_2^2 - a_3^2\right) + 2\,(a_0 a_1 + a_2 a_3)\, i_1}. \end{aligned} \tag{31}$$

Notice that the denominator of the previous equation is in $\mathbb{C}_1$. Multiplying the numerator and the denominator by the i_1-conjugate of the denominator, the following result is obtained:

$$x^* = \frac{b^* \,\text{conj}_2(a^*)\,\text{conj}_1(a^* \,\text{conj}_2(a^*))}{\left(a_0^2 - a_1^2 + a_2^2 - a_3^2\right)^2 + 4\,(a_0 a_1 + a_2 a_3)^2}. \tag{32}$$

Notice that the division of the two bicomplex numbers was reduced to the product of three multicomplex numbers divided by a scalar. An equivalent approach has been implemented for multidual numbers. The iterative procedure for computing the division of two hypercomplex numbers is summarized in Algorithm 1.

Notice that the hypercomplex multiplications inside the loop of the algorithm are $O(2^{2n})$. Therefore, the computational complexity of this method is $O(2^{2n})$.

ALGORITHM 1: Division of Two Hypercomplex Numbers.

Input:
b^* , a^* Original numerator and denominator.
r_b , r_a Hypercomplex order of the inputs.
Output: Multicomplex $x^* = b^*/a^*$
$n^* \leftarrow \text{allocate}(\max(r_a, r_b))$; # Memory for numerator
$d_c^* \leftarrow \text{allocate}(r_a)$; # Memory for conjugate
$n^* \leftarrow b^*$; $d^* \leftarrow a^*$; # Initialize num. and den.
for $k = r_a$ **to** 1 **step** -1 **do**
  $d_c^* \leftarrow \text{conj}_k(d^*)$; # Compute i_k-conjugate
  $n^* \leftarrow n^* d_c^*$; # Compute new numerator
  $d^* \leftarrow d^* d_c^*$; # Compute new denominator
$n^* \leftarrow n^*/\text{Re}(d^*)$; # Compute final result

Another method for the computation of division is to solve a system of equations $\boldsymbol{Ax} = \boldsymbol{b}$ by Gaussian elimination, whose computational complexity is $O(2^{3n})$. However, since the Cauchy-Riemann matrix form of multidual numbers is a triangular matrix, the complexity for multidual numbers can be reduced to $O(2^{2n})$ by using an LU decomposition.

2.9 Computation of elementary functions

In spite of their significant similarities, the computation of functions for multicomplex and multidual numbers has been traditionally carried out using different approaches: Multidual functions are based on a truncated Taylor series, which takes advantage of the nilpotent rule ($\epsilon_p^2 = 0,\ \forall p \in \mathbb{N}$) to guarantee exact mathematical results [7]. However, multicomplex functions are computed using functions of matrices, given the isomorphism between the multicomplex algebra and the real matrix algebra [14]. Implementations of matrix functions can be found in different numerical analysis platforms, such as MATLAB, or the SciPy library for Python. However, some of these implementations are based on the Schur's decomposition of a matrix, which is not well suited for diagonally dominant matrices with repeated eigenvalues, as is the case with the Cauchy-Riemann matrix form of multicomplex numbers [8].

To overcome these difficulties, and taking into account that the main goal of the library is the computation of derivatives, the approach provided in MultiZ is to use a truncated Taylor series for the computation of both multicomplex and multidual functions. The accuracy of the truncated Taylor series expansion for the computation of elementary hypercomplex functions is verified in Appendix B, where this method is compared against different matrix functions included in the open-source Python library Scipy.

Consider the truncated Taylor series expansion of a holomorphic multicomplex function $f(x^*)$ about x_0, where $x_0 = \text{Re}(x^*)$,

$$f(x^*) \approx f(x_0) + \sum_{k=1}^{n} \frac{f^{(k)}(x_0)}{k!}(x^* - x_0)^k. \tag{33}$$

Since the expansion was made about x_0, $f(x_0)$ and its derivatives are real numbers, and $f(x_0)$ can be evaluated using the built-in functions of any programming language. Notice that the powers of $(x^* - x_0)$ are hypercomplex numbers, which can be computed by multiplication, as explained previously. The derivatives of $f(x_0)$ can be easily computed for many common functions, such as sine, cosine, exponential, logarithm, and square root.

To ensure the computation of machine-precision derivatives of real-valued functions, the variable n in Equation (33) must be equal to the order of derivative required by the user. Using a value of n greater than the order of the required derivative results in the generation of zero-valued terms in the Taylor series. In the multidual case, this is caused by the nilpotent rule of the imaginary units. In the multicomplex case, the extra terms become approximately zero due to the small step size h. By default, MultiZ sets n equal to the order of the hypercomplex number. However, the user can specify a different truncation point by adding an optional argument to the function call.

The truncated Taylor series (Equation (33)) is used for both the multicomplex and the multidual case. In the multicomplex algebra case, the truncated Taylor Series approach only works for small values of the step size h. In the multidual algebra case, the approximate sign is replaced by an equal sign, and the result does not depend on the step size h. An efficient approach for computing hypercomplex functions is shown in Algorithm 2, in which the powers of $(x^* - x_0)$ are obtained by accumulation.

ALGORITHM 2: Taylor Series Approach for the Computation of Hypercomplex Functions.

Input:
x^* Hypercomplex number
n Hypercomplex order, or order of required derivative.
Output: Hypercomplex number $y^* = f(x^*)$
$x_0 \leftarrow \mathrm{Re}(x^*)$; $r \leftarrow 1$; $p^* \leftarrow 1$; $y^* \leftarrow f(x_0)$;
for $k = 1$ **to** n **do**
  $r \leftarrow r \cdot k$; # Update factorial
  $d \leftarrow$ compute $f^{(k)}(x_0)$; # Compute derivative
  $p^* \leftarrow p^* (x^* - x_0)$; # Update $(x^* - x_0)^k$
  $y^* \leftarrow y^* + (d/r)\, p^*$; # $\sum \frac{f^{(k)}(x_0)}{k!}(x^* - x_0)^k$

2.9.1 Exponential Function. Since all derivatives of e^x are equal to e^x, the truncated Taylor series equation is reduced to

$$e^{x^*} \approx e^{x_0} \left[1 + \sum_{k=1}^{n} \frac{1}{k!} (x^* - x_0)^k \right]. \tag{34}$$

2.9.2 Sine and Cosine. An efficient and accurate computation of the sine of a hypercomplex number, based on the Taylor series expansion, is performed using binary bitwise operations. Let the derivative of $\sin(x_0)$ equal to the product of a sign $s(k)$ and a function $g(x_0, k)$ defined as

$$g(x_0, k) = \begin{cases} \cos(x_0), & \text{if } k \text{ is an odd number,} \\ \sin(x_0) & \text{otherwise,} \end{cases} \tag{35}$$

$$s(k) = \begin{cases} -1, & \text{if } k \gg 1 \text{ is an odd number,} \\ 1 & \text{otherwise.} \end{cases} \tag{36}$$

where $\gg$ is the binary bitwise right shift operator. The Taylor series approximation of the sine is then defined as

$$\sin(x^*) \approx \sin(x_0) + \sum_{k=1}^{n} \frac{1}{k!} s(k)\, g(x_0, k)\, (x^* - x_0)^k. \tag{37}$$

For the cosine, the sign $s(k)$ and a function $g(x_0, k)$ are defined as

$$g(x_0, k) = \begin{cases} \sin(x_0), & \text{if } k \text{ is an odd number,} \\ \cos(x_0) & \text{otherwise,} \end{cases} \tag{38}$$

$$s(k) = \begin{cases} -1, & \text{if } k+1 \gg 1 \text{ is an odd number,} \\ 1 & \text{otherwise.} \end{cases} \tag{39}$$

Therefore, the Taylor series approximation of the cosine is defined as

$$\cos(x^*) \approx \cos(x_0) + \sum_{k=1}^{n} \frac{1}{k!} s(k)\, g(x_0, k)\, (x^* - x_0)^k. \tag{40}$$

2.9.3 Power Function. Consider the elementary differentiation rule for the power function $f(x) = x^a$, $a \in \mathbb{R}$, to obtain three derivatives:

$$f'(x) = a\, x^{a-1}, \quad f''(x) = a\,(a-1)\, x^{a-2}, \quad f'''(x) = a\,(a-1)\,(a-2)\, x^{a-3}. \tag{41}$$

Generalizing this rule for an arbitrary order of derivative, and evaluating at $x = x_0$, the following equation is obtained:

$$f^{(k)}(x_0) = \left[\prod_{j=0}^{k-1} (a-j)\right] x_0^{a-k}\,, \qquad f(x_0) = x_0^a\,, \qquad a \in \mathbb{R}. \tag{42}$$

Therefore, the truncated Taylor series approximation can be expressed as

$$(x^*)^a \approx x_0^a + \sum_{k=1}^{n} \frac{1}{k!} \left[\prod_{j=0}^{k-1} (a-j)\right] x_0^{a-k} (x^* - x_0)^k. \tag{43}$$

2.9.4 Logarithm. The derivatives of the logarithm function $f(x) = \log(x)$, evaluated at x_0, are obtained using the following equation:

$$f^{(k)}(x_0) = (-1)^{k-1} \frac{(k-1)!}{x_0^k}. \tag{44}$$

The truncated Taylor series approximation for the logarithm of a hypercomplex number is

$$\log(x^*) \approx \log(x_0) + \sum_{k=1}^{n} \frac{(-1)^{k-1}}{k\, x_0^k} (x^* - x_0)^k. \tag{45}$$

3 NUMERICAL TESTS AND RESULTS

3.1 A simple numerical example

A more complete version of the introductory example shown in Section 2 is presented here. Consider again the composite function used by several authors before [7, 14, 16]:

$$f(x) = \frac{e^x}{\sqrt{\sin^3 x + \cos^3 x}}. \tag{46}$$

Table 9. Computation of Five Derivatives of Equation (46)

Derivative	Result	Relative error
1st	2.4540383344548480	9.04813e-016
2nd	2.3559293755346875	1.31949e-015
3rd	−9.3319100381987052	1.33247e-015
4th	−55.731811928497279	5.09973e-016
5th	70.323499129435177	2.82910e-015

The code for computing all derivatives up to fifth order using multicomplex numbers is shown below:

```
1  PROGRAM main     ! "mrealk" sets precision
2    USE multiz     ! of real number types
3    IMPLICIT NONE  ! x0: eval point, h: step size.
4    REAL(mrealk), PARAMETER ::  x0 = 0.5d0, h = 1d-10
5    TYPE(mcomplex)  ::  x,  y
6    REAL(mrealk)    ::  d1, d2, d3, d4, d5
7    x  = x0 + h*(im(1) + im(2) + im(3) + im(4) + im(5))
8    y  = exp(x)/sqrt(sin(x)**3 + cos(x)**3)
9    d1 = AIMAG(y, 1)/h
10   d2 = AIMAG(y, [1,2])/h**2
11   d3 = AIMAG(y, [1,2,3])/h**3
12   d4 = AIMAG(y, [1,2,3,4])/h**4
13   d5 = AIMAG(y, [1,2,3,4,5])/h**5
14 END PROGRAM main
```

For the multidual case, only two lines of the previous code need to be modified:

```
5    TYPE(mdual)  ::  x,  y

7    x  = x0 + h*(eps(1) + eps(2) + eps(3) + eps(4) + eps(5))
```

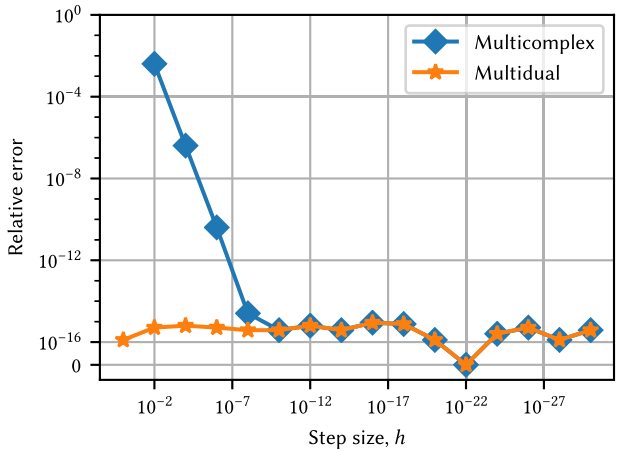
Since the computation of any derivative using multidual numbers is step-size-independent [7], the step size h can be equal to 1 or omitted. Nevertheless, for this numerical test the step size is kept equal in both the multicomplex and the multidual cases to show that when the step size is sufficiently small for the multicomplex case ($h \leq 10^{-9}$), the results obtained in both cases are identical and machine-precision accurate.

The results of the previous code listings and their relative error with respect to analytical derivatives of Equation (46) are shown in Table 9. The relative error is defined as

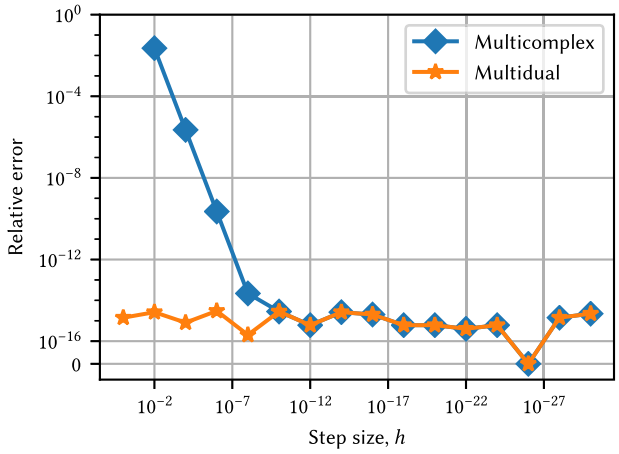
$$\eta = \left| \frac{v_0 - v}{v} \right|, \tag{47}$$

where v is the exact value and v_0 the approximation. It can be observed from the relative error in Table 9 that the derivatives of the test function computed using hypercomplex numbers are exact to machine precision.

In Figure 1, the accuracy of multicomplex and multidual numbers is compared for the computation of fourth- and fifth-order derivatives of the proposed function. It can be noticed that the relative error of the multicomplex algebra converges quadratically for step sizes $h > 10^{-9}$, while the multidual algebra is step-size-independent and exact to machine precision. Notice that the



(a) Fourth-order derivative accuracy



(b) Fifth-order derivative accuracy

Fig. 1. Accuracy of derivatives of the function $f(x) = \exp(x)/\sqrt{\sin^3 x + \cos^3 x}$ computed using multicomplex and multidual numbers for different step sizes.

multicomplex algebra reaches identical and machine-precision-accurate results for a step size h lower or equal to 10^{-9}.

3.2 Efficiency of hypercomplex multiplication methods

This test compares the execution time of multiplying two hypercomplex numbers using different methods. It takes into consideration two possible cases of hypercomplex number data structure: matrix-based and vector-based. Matrix-based refers to implementations of hypercomplex algebras where all computations involve the use of Cauchy-Riemann matrices, while vector-based refers to implementations where hypercomplex numbers are stored as vectors, and the Cauchy-Riemann matrix is computed only as necessary.

Table 10. Execution Times of Different Hypercomplex Multiplication Methods

(a) Multicomplex algebra

Order	MM	MV1	MV2	HC	BB
1	1.00E-07	7.98E-08	1.95E-07	7.53E-08	8.15E-08
2	1.37E-07	8.73E-08	2.26E-07	8.87E-08	1.37E-07
3	3.26E-07	1.25E-07	3.23E-07	1.39E-07	3.57E-07
4	1.42E-06	2.89E-07	9.40E-07	3.47E-07	1.25E-06
5	8.05E-06	1.92E-06	5.34E-06	1.53E-06	4.71E-06
6	4.82E-05	7.72E-06	2.68E-05	6.20E-06	2.13E-05

(b) Multidual algebra

Order	MM	MV1	MV2	HC	BB
1	9.94E-08	7.97E-08	1.99E-07	7.42E-08	7.47E-08
2	1.36E-07	8.83E-08	2.28E-07	8.43E-08	1.02E-07
3	3.27E-07	1.25E-07	3.20E-07	1.10E-07	1.94E-07
4	1.42E-06	2.86E-07	8.00E-07	2.15E-07	5.52E-07
5	8.05E-06	1.90E-06	3.37E-06	3.96E-07	1.76E-06
6	4.76E-05	7.64E-06	1.53E-05	1.42E-06	6.62E-06

All values provided in seconds. (MM) Matrix-Matrix multiplication, (MV1) matrix-based data type Matrix-Vector multiplication, (MV2) vector-based data type Matrix-vector multiplication, (HC) Hard-Coded function, (BB) Binary Bitwise indexation of the CR matrix.

The methods to be compared are: (i) matrix-matrix multiplication for a matrix-based data structure (MM), (ii) matrix-vector multiplication for a matrix-based data structure (MV1), (iii) matrix-vector multiplication for a vector-based data structure (MV2), (iv) a hard-coded function that contains the multiplication operations required for each hypercomplex order (HC), and (v) use of binary bitwise indexation of the Cauchy-Riemann matrix for performing matrix-vector multiplication (BB).

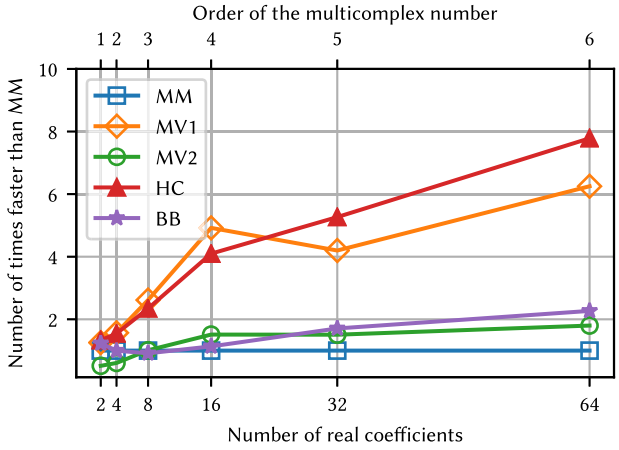
For this test, MM, MV1, and MV2 used dense matrices. However, for the multidual algebra case, HC and BB took into account the sparsity of the Cauchy-Riemann matrix to avoid unnecessary calculations.

Differently from MV1, MV2 requires the allocation and the computation of the Cauchy-Riemann matrix. The matrix was computed using a hard-coded function.

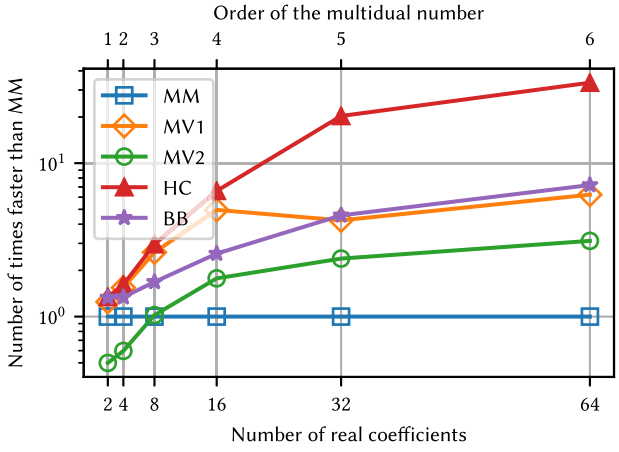
The test was developed in Fortran and compiled with GNU Fortran 7.3. The computations were performed on the UTSA's HPC cluster SHAMU, on a computing node with an Intel Xeon E5-2650 processor running at 2.6 GHz.

Results for multiplication of multicomplex numbers are shown in Table 10(a) and Figure 2(a). It can be observed that methods intended for matrix-based data structure implementations, such as matrix-matrix multiplication (MM) and matrix-vector case 1 (MV1) are the upper and lower bounds of efficiency for all methods of multiplication for multicomplex orders below five, where HC is up to 17% less efficient than MV1. For fifth- and sixth-order multicomplex numbers, HC is approximately 20% more efficient than MV1.

Results for multiplication of multidual numbers are shown in Table 10(b) and Figure 2(b). It can be observed that methods based on matrix-matrix and matrix-vector multiplication have no improvement with respect to the results of the multicomplex algebra case due to the use of dense matrices. In contrast, methods that take advantage of the triangular shape and the sparsity of the



(a) Multicomplex algebra



(b) Multidual algebra

Fig. 2. Ratio of execution times of the Matrix-Matrix multiplication over other methods. (MM) Matrix-Matrix multiplication, (MV1) matrix-based data type Matrix-Vector multiplication, (MV2) vector-based data type Matrix-vector multiplication, (HC) Hard-Coded function, (BB) Binary Bitwise indexation of the CR matrix.

Cauchy-Riemann matrix of the multidual numbers, such as HC and BB, have a significant improvement in performance with respect to the multicomplex case: For multidual numbers of order six, HC runs almost 4.4× faster than HC for multicomplex numbers, and multidual multiplication through BB runs approximately 3.2× faster than BB for multicomplex numbers.

MultiZ uses a vector-based data structure and offers no support for matrix-based data structures. The library supports HC and BB to multiply hypercomplex numbers.

3.3 Design sensitivity example

In this section, MultiZ is used to compute sensitivities of a simple linear elliptic system. Consider the spring system shown in Figure 3, proposed by Tortorelli and Michaleris [23]. The first and

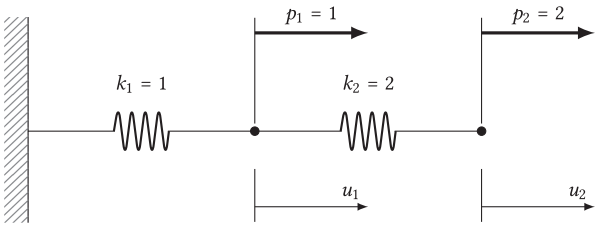


Fig. 3. Spring system of the sensitivity analysis example.

second-order sensitivities of the compliance, c, with respect to the spring constants, k_1 and k_2, are computed and compared to the reference, e.g., $\partial^2 c/(\partial k_1 \partial k_2)$.

The governing equation of the system is $\boldsymbol{K}\boldsymbol{u} = \boldsymbol{p}$, where the $\boldsymbol{K}$ is the stiffness matrix, $\boldsymbol{u}$ contains displacements of nodes 1 and 2, and $\boldsymbol{p}$ contains the loads applied to those nodes. The compliance of the system is given by $c = \boldsymbol{p} \cdot \boldsymbol{u}$. The stiffness matrix of the spring system is defined as

$$\boldsymbol{K} = \begin{bmatrix} k_1 + k_2 & -k_2 \\ -k_2 & k_2 \end{bmatrix}. \tag{48}$$

For this example problem, the set of design parameters is composed of the spring constants, k_1, k_2. The first step to numerically compute sensitivities of the compliance c using hypercomplex numbers is to convert the design parameters to hypercomplex, as well as all of the variables that depend on them. Therefore, the system of equations becomes $\boldsymbol{K}^*\boldsymbol{u}^* = \boldsymbol{p}$, and the hypercomplex version of the compliance is $c^* = \boldsymbol{p} \cdot \boldsymbol{u}^*$. Note that the load vector $\boldsymbol{p}$ does not depend on the spring constants, then its hypercomplex version is equal to the real version ($\boldsymbol{p}^* = \boldsymbol{p}$). The hypercomplex version of the stiffness matrix is defined as

$$\boldsymbol{K}^* = \begin{bmatrix} k_1^* + k_2^* & -k_2^* \\ -k_2^* & k_2^* \end{bmatrix}. \tag{49}$$

The second step is to choose a hypercomplex algebra and the hypercomplex order. For simplicity, multidual algebra is used here, since it does not require the use of a step size parameter. Additionally, second order is used, since it is the minimum hypercomplex order required to compute second-order sensitivities. Consequently, the spring constants k_1^*, k_2^*, the arrays of the system of equations $\boldsymbol{K}^*$ and $\boldsymbol{u}^*$, and the compliance c^* are all bidual variables.

The multidual system of equations needs to be solved three times to compute all second-order sensitivities $\partial^2 c/\partial k_1^2$, $\partial^2 c/\partial k_2^2$, and $\partial^2 c/(\partial k_1 \partial k_2)$. Obtaining all sensitivities in a single run is also possible, but it requires the use of fourth-order hypercomplex numbers, which means a higher number of computations, since the solution of a such system is $O(2^{3n})$, where n is the order of the hypercomplex numbers. To compute $\partial^2 c/(\partial k_1 \partial k_2)$, the multidual spring constants are defined as $k_1^* = k_1 + \epsilon_1$, $k_2^* = k_2 + \epsilon_2$, then the multidual stiffness matrix is defined as

$$\boldsymbol{K}^* = \begin{bmatrix} k_1 + \epsilon_1 + k_2 + \epsilon_2 & -k_2 - \epsilon_2 \\ -k_2 - \epsilon_2 & k_2 + \epsilon_2 \end{bmatrix}, \tag{50}$$

which expressed in terms of real and imaginary parts becomes

$$\boldsymbol{K}^* = \begin{bmatrix} k_1 + k_2 & -k_2 \\ -k_2 & k_2 \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \epsilon_1 + \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \epsilon_2 + \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \epsilon_1 \epsilon_2. \tag{51}$$

Notice that $\boldsymbol{K} = \mathrm{Re}(\boldsymbol{K}^*)$, $\partial \boldsymbol{K}/\partial k_1 = \mathrm{Im}_1(\boldsymbol{K}^*)$, $\partial \boldsymbol{K}/\partial k_2 = \mathrm{Im}_2(\boldsymbol{K}^*)$, and $\partial^2 \boldsymbol{K}/(\partial k_1 \partial k_2) = \mathrm{Im}_{1\,2}(\boldsymbol{K}^*)$. Using comma notation for derivatives, the multidual stiffness matrix for this case can be expressed as

$$\boldsymbol{K}^* = \boldsymbol{K} + \boldsymbol{K}_{,1}\epsilon_1 + \boldsymbol{K}_{,2}\epsilon_2 + \boldsymbol{K}_{,12}\,\epsilon_1\epsilon_2, \tag{52}$$

```
TYPE(mdual)   ::  k1, k2, p1, p2, f
REAL(mrealk)  ::  fr, df1, df2, df11, df22, df12
p1 = 1.d0
p2 = 2.d0
k1 = 1.d0
k2 = 2.d0
! Solve system with perturbations on k1 only.
f    = compliance(k1 + eps(1) + eps(2), k2, p1, p2)
df1  = aimag(f,1)       ! 1st order sens. wrt k1.
df11 = aimag(f,[1,2])   ! 2nd order sens. wrt k1.
! Solve system with perturbations on k2 only.
f = compliance(k1, k2 + eps(1) + eps(2), p1, p2)
df2  = aimag(f,1)       ! 1st order sens. wrt k2.
df22 = aimag(f,[1,2])   ! 2nd order sens. wrt k2.
! Solve system with perturbations on k1 and k2.
f = compliance(k1 + eps(1), k2 + eps(2), p1, p2)
fr   = real(f)          ! Compliance value.
df12 = aimag(f,[1,2])   ! Mixed 2nd order sens. wrt k1 & k2.
```

Listing 2. Computation of first- and second-order sensitivities of the compliance with respect to the spring constants.

where $\boldsymbol{K}_{,i} \coloneqq \partial \boldsymbol{K}/\partial k_i$ and $\boldsymbol{K}_{,ij} \coloneqq \partial^2 c/(\partial k_i \partial k_j)$. The displacement vector $\boldsymbol{u}^*$ and the compliance c^* are expanded as bidual numbers as

$$\boldsymbol{u}^* = \boldsymbol{u} + \boldsymbol{u}_{,1}\epsilon_1 + \boldsymbol{u}_{,2}\epsilon_2 + \boldsymbol{u}_{,12}\,\epsilon_1\epsilon_2, \tag{53}$$

$$c^* = c + c_{,1}\,\epsilon_1 + c_{,2}\,\epsilon_2 + c_{,12}\,\epsilon_1\epsilon_2, \tag{54}$$

where, $\boldsymbol{u}_{,i} \coloneqq \partial \boldsymbol{u}/\partial k_i$, $\boldsymbol{u}_{,ij} \coloneqq \partial^2 \boldsymbol{u}/(\partial k_i \partial k_j)$, $c_{,i} \coloneqq \partial c/\partial k_i$, $c_{,ij} \coloneqq \partial^2 c/(\partial k_i \partial k_j)$.

The computation of $\boldsymbol{u}^*$ is carried out by solving the hypercomplex system of equations $\boldsymbol{K}^*\boldsymbol{u}^* = \boldsymbol{p}$. This system can be solved using a standard linear algebra library, such as LAPACK, by rewriting the system in terms of real-valued arrays, where the stiffness matrix $\boldsymbol{K}^*$ is converted to a Cauchy-Riemann block matrix, and $\boldsymbol{u}^*$ and $\boldsymbol{p}^*$ are expressed as block vectors. For the case where $k_1^* = k_1 + \epsilon_1$ and $k_2^* = k_2 + \epsilon_2$ the system can be rewritten in terms of real-valued arrays as

$$\begin{bmatrix} \boldsymbol{K} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{K}_{,1} & \boldsymbol{K} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{K}_{,2} & \boldsymbol{0} & \boldsymbol{K} & \boldsymbol{0} \\ \boldsymbol{K}_{,12} & \boldsymbol{K}_{,2} & \boldsymbol{K}_{,1} & \boldsymbol{K} \end{bmatrix} \begin{bmatrix} \boldsymbol{u} \\ \boldsymbol{u}_{,1} \\ \boldsymbol{u}_{,2} \\ \boldsymbol{u}_{,12} \end{bmatrix} = \begin{bmatrix} \boldsymbol{p} \\ \boldsymbol{0} \\ \boldsymbol{0} \\ \boldsymbol{0} \end{bmatrix}. \tag{55}$$

MultiZ includes functions to format arrays as shown in Equation (55)

```
k_cr = mto_cr(k)    ! Compute CR block matrix of k.
p_cr = mto_cr(p)    ! Compute block vector of p.
```

and also to convert Cauchy-Riemann compatible arrays to hypercomplex variable types.

```
u = mcr_to_mduvec(u_cr) ! Convert to multidual vector.
```

Following a similar procedure, $\partial^2 c/\partial k_1^2$ is determined by defining the multidual spring constants as $k_1^* = k_1 + \epsilon_1 + \epsilon_2$, $k_2^* = k_2$, and $\partial^2 c/\partial k_2^2$ by defining $k_1^* = k_1$, $k_2^* = k_2 + \epsilon_1 + \epsilon_2$. The code required for computing all sensitivities through second order using multidual variables is included in Listings 2–4. The real data type version of the compliance function is shown in Listing 3, and

```
 1  FUNCTION compliance(k1, k2, p1, p2) RESULT (f)
 2    REAL(8), INTENT(IN)  :: k1, k2, p1, p2
 3    REAL(8)              :: f
 4    REAL(8) :: k(n,n), p(n), u(n)
 5    INTEGER :: ipiv(n), info
 6    k(1,1) = k1 + k2
 7    k(1,2) = -k2
 8    k(2,1) = -k2
 9    k(2,2) =  k2     ! Lapack's subroutine "gesv"
10    p(1) = p1        ! overwrites the RHS with
11    p(2) = p2        ! the solution of the system.
12    u = p
13    CALL dgesv(n, 1, k, n, ipiv, u, n, info)
14    f = DOT_PRODUCT(p, u)
15  END FUNCTION compliance
```

Listing 3. Function to compute the compliance of spring system using real data types. This is a real-valued example code that is modified in Listing 4 using multidual variables.

```
 1  FUNCTION compliance(k1, k2, p1, p2) RESULT (f)
 2    TYPE(mdual), INTENT(IN)  :: k1, k2, p1, p2
 3    TYPE(mdual)              :: f
 4    REAL(mrealk) :: k_cr(n_cr,n_cr), u_cr(n_cr)
 5    INTEGER      :: ipiv(n_cr), info
 6    TYPE(mdumat) :: k
 7    TYPE(mduvec) :: p, u
 8    CALL mallocate(k, order=ord, d1=n, d2=n)
 9    CALL mallocate(p, order=ord, d1=n)
10    CALL mallocate(u, order=ord, d1=n)
11    CALL mset( k, 1, 1, k1 + k2)
12    CALL mset( k, 1, 2, -k2)      ! The suffix "cr" stands
13    CALL mset( k, 2, 1, -k2)      ! for Cauchy-Riemann.
14    CALL mset( k, 2, 2, k2)
15    CALL mset( p, 1, p1)          ! The system is solved
16    CALL mset( p, 2, p2)          ! using a Cauchy-Riemann
17    u = p                         ! block matrix, and a
18    k_cr = mto_cr(k)              ! compatible vector that
19    u_cr = mto_cr(u)              ! contains all imag. parts.
20    CALL dgesv(n_cr, 1, k_cr, n_cr, ipiv, u_cr, n_cr, info)
21    u = mcr_to_mduvec(u_cr, ord)
22    f = mdot_nc(p, u)
23  END FUNCTION compliance
```

Listing 4. Function to compute the compliance of spring system using multidual data types. This function is used in Listing 2 for the sensitivity analysis of the compliance.

the multidual version is shown in Listing 4. Results of the sensitivities computed using multidual numbers are shown in Table 11.

The sensitivity results shown in Table 11 are machine-precision-accurate, or in some cases exact. The sensitivity analysis procedure shown in this section can be easily applied to different finite element models.

Table 11. Sensitivities of the Compliance with Respect to the Spring Constants

Sensitivity	Numerical result	Analytical value	Relative error
$\partial c/\partial k_1$	−8.9999999999999964	−9	3.9475e-16
$\partial c/\partial k_2$	−1.	−1	0.
$\partial^2 c/\partial k_1^2$	17.999999999999993	18	3.9475e-16
$\partial^2 c/\partial k_2^2$	1.	1	0.
$\partial^2 c/(\partial k_1 \partial k_2)$	0.	0	0.

Table 12. Data Types of the Python Version of MultiZ

Name	Description
`mcomplex`	Multicomplex number.
`mdual`	Multidual number.
`marray`	Hypercomplex N-dimensional array.

4 PYTHON VERSION OF THE MULTIZ LIBRARY

Similar to the Fortran version of MultiZ, a Python version was developed to take advantage of object-oriented programming and operator overloading. It was developed using Python classes, and the syntax for the creation of multicomplex and multidual numbers is mostly the same as the Fortran version. The Python version of the simple numerical example (Section 3.1) is shown below.

```
1  from multiZ.mcomplex import *
2  x0 = 0.5; h=1e-10
3  x = x0 + h*(im(1) + im(2) + im(3) + im(4) + im(5))
4  y = exp(x)/sqrt(sin(x)**3 + cos(x)**3)
5  d1 = y.getIm(1)/h
6  d2 = y.getIm([1,2])/h**2
7  d3 = y.getIm([1,2,3])/h**3
8  d4 = y.getIm([1,2,3,4])/h**4
9  d5 = y.getIm([1,2,3,4,5])/h**5
```

For the multidual case, only two lines of the previous code need to be modified.

```
1  from multiZ.mdual import *

3  x  = x0 + h*(eps(1) + eps(2) + eps(3) + eps(4) + eps(5))
```

The data structures of the Python version are based on NumPy arrays to efficiently support vectorized operations, slicing, and linear algebra computations. The use of Numpy allows the array data structures of the Python version to be defined as a single data type called `marray` instead of requiring specific data types for vectors and matrices of multicomplex and multidual numbers. Furthermore, the Python version offers convenient support for assigning and extracting information from hypercomplex arrays, since the syntax is the same used for Numpy arrays and intrinsic Python data types due to the overloading of the indexing operators. Tables 12 to 17 contain a summary of the functions and operators supported by the Python version of MultiZ.

Table 13. Functions and Methods of the Python Version to Handle Hypercomplex Numbers

Name	Description
`mcomplex`	Create a multicomplex number by providing the coefficients.
`mdual`	Create a multidual number by providing the coefficients.
`im`	Create a unit value imaginary part for multicomplex numbers.
`eps`	Create a unit value imaginary part for multidual numbers.
`x.real`	Extract the real part of the number `x`.
`x.imag`	Extract the specified imaginary part of the number or array `x`.
`x.conjugate`	Compute the conjugate of the hypercomplex number `x` with respect to the specified imaginary unit.

Table 14. Operators and Elementary Functions of the Python Version

Op.	Description	Func.	Description
`+`	Addition	`sin`	Sine
−	Subtraction and negation	`cos`	Cosine
`*`	Multiplication	`exp`	Exponential
`/`	Division and reciprocal	`log`	Natural log.
`**`	Power (integer and fraction)	`sqrt`	Square root

Table 15. Linear Algebra Operators of the Python Version

Name	Description
`x.T`	Transpose of the hypercomplex matrix `x`.
`dot`	Matrix-matrix multiplication, matrix-vector multiplication, and dot product. All performed without complex conjugation.

Table 16. Functions of the Python Version to Handle Hypercomplex Arrays

Name	Description
`zeros`	Allocate space for arrays of hypercomplex numbers.
`x.change_order`	Increase or decrease number of imaginary coefficients of the array `x`.
`shape`	Get the shape of an array. Number of elements per dimension.
`size`	Get total number of elements of the array.

Table 17. Support for Cauchy-Riemann Matrices of the Python Version

Name	Description
`x.to_cr`	Convert a hypercomplex number or array `x` to a Cauchy-Riemann compatible array.
`mcr_to_mnumber`	Convert CR matrix to hypercomplex number
`mcr_to_marray`	Convert CR compatible array to hypercomplex array
`x.get_cr`	Compute a CR matrix element at the specified position from a hypercomplex number `x`

5 CONCLUSIONS

In this article, MultiZ is presented as a library primarily intended for the computation of high-order derivatives of real-valued functions using multicomplex or multidual numbers. MultiZ supports the overloading of mathematical operators and functions, allowing users to perform sensitivity analysis with minimum code rewriting. This library can also be used for general purpose multicomplex and multidual calculations. However, it is recommended to use matrix functions for the computation of elementary multicomplex functions, as the functions included in this library are only valid for small imaginary parts.

The hypercomplex algebras supported by the library, multicomplex and multidual, share various similarities such as commutative multiplication, recursive definition, and the capability of computing high-order derivatives with machine-precision accuracy. Moreover, numerical tests show that both methods obtain exactly the same results when the step size h is sufficiently small for multicomplex.

The use of an indexation method based on the binary numbering system, and the use of binary bitwise operations for the indexation of the Cauchy-Riemann matrix, provide a straightforward and low-memory-consuming method for the computation of derivatives of arbitrary order using multicomplex and multidual numbers. Furthermore, binary indexation techniques can be used to generate high-efficiency code for derivative computation.

A truncated Taylor series approach was presented as an alternative to matrix functions for the computation of hypercomplex functions. This approach, which combines hypercomplex multiplication with exact derivatives of real-valued functions, yields machine-precision high-order derivates of composite real-valued functions for both multicomplex and multidual algebras.

Numerical tests using matrix functions showed that, in contrast with Cauchy-Riemann matrices of multicomplex numbers, the Cauchy-Riemann matrices of multidual numbers can be readily used with common matrix function methods for the computation of high-order derivatives, even when very small imaginary parts are involved.

APPENDICES

A MATRIX REPRESENTATION OF BIDUAL NUMBERS

Consider the multiplication of two bidual numbers $a^*, b^* \in \mathbb{D}_2$,

$$a^* = a_0 + a_1\epsilon_1 + a_2\epsilon_2 + a_3\epsilon_1\epsilon_2, \tag{56}$$

$$b^* = b_0 + b_1\epsilon_1 + b_2\epsilon_2 + b_3\epsilon_1\epsilon_2, \tag{57}$$

$$\begin{aligned} c^* = a^*b^* = {} & a_0b_0 + a_1b_1\,\epsilon_1^2 + a_2b_2\,\epsilon_2^2 + a_3b_3\,\epsilon_1^2\epsilon_2^2 \\ & + \left(a_1b_0 + a_0b_1 + a_3b_2\,\epsilon_2^2 + a_2b_3\,\epsilon_2^2 \right)\epsilon_1 \\ & + \left(a_2b_0 + a_3b_1\,\epsilon_1^2 + a_0b_2 + a_1b_3\,\epsilon_1^2 \right)\epsilon_2 \\ & + \left(a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3 \right)\epsilon_1\epsilon_2. \end{aligned} \tag{58}$$

Applying the nilpotent rule of the dual numbers ($\epsilon_p^2 = 0\ \forall p$), the product is equal to

$$\begin{aligned} c^* = {} & a_0b_0 + (a_1b_0 + a_0b_1)\,\epsilon_1 + (a_2b_0 + a_0b_2)\,\epsilon_2 \\ & + (a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3)\,\epsilon_1\epsilon_2. \end{aligned} \tag{59}$$

Extracting the real and the imaginary parts from c^*,

$$c_0 = \mathrm{Re}(c^*) = a_0 b_0, \tag{60}$$
$$c_1 = \mathrm{Im}_1(c^*) = a_1 b_0 + a_0 b_1, \tag{61}$$
$$c_2 = \mathrm{Im}_2(c^*) = a_2 b_0 + a_0 b_2, \tag{62}$$
$$c_3 = \mathrm{Im}_{1,2}(c^*) = a_3 b_0 + a_2 b_1 + a_1 b_2 + a_0 b_3. \tag{63}$$

The product $a^* b^*$ can be also expressed, from Equation (58), as a matrix multiplied by two vectors,

$$a^* b^* = \begin{bmatrix} 1 \\ \epsilon_1 \\ \epsilon_2 \\ \epsilon_1\epsilon_2 \end{bmatrix}^T \begin{bmatrix} a_0 & a_1\epsilon_1^2 & a_2\epsilon_2^2 & a_3\epsilon_1^2\epsilon_2^2 \\ a_1 & a_0 & a_3\epsilon_2^2 & a_2\epsilon_2^2 \\ a_2 & a_3\epsilon_1^2 & a_0 & a_1\epsilon_1^2 \\ a_3 & a_2 & a_1 & a_0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}. \tag{64}$$

Let $\boldsymbol{A}$ be the matrix in the previous expression. Applying the nilpotent rule of the dual numbers, the following coefficients are obtained for $\boldsymbol{A}$:

$$\boldsymbol{A} = \begin{bmatrix} a_0 & 0 & 0 & 0 \\ a_1 & a_0 & 0 & 0 \\ a_2 & 0 & a_0 & 0 \\ a_3 & a_2 & a_1 & a_0 \end{bmatrix}. \tag{65}$$

Also, let M be a mapping function $M : \mathbb{C}_2 \to \mathbb{R}^{4\times4}$, such that $\boldsymbol{A} = M(a^*)$. If $\boldsymbol{A}$ is a valid matrix representation of the bidual number a^*, then $\boldsymbol{C} = M(c^*)$ must be obtained as $\boldsymbol{C} = \boldsymbol{AB}$, where $\boldsymbol{B} = M(b^*)$.

$$\boldsymbol{AB} = \begin{bmatrix} a_0 & 0 & 0 & 0 \\ a_1 & a_0 & 0 & 0 \\ a_2 & 0 & a_0 & 0 \\ a_3 & a_2 & a_1 & a_0 \end{bmatrix} \begin{bmatrix} b_0 & 0 & 0 & 0 \\ b_1 & b_0 & 0 & 0 \\ b_2 & 0 & b_0 & 0 \\ b_3 & b_2 & b_1 & b_0 \end{bmatrix},$$

$$= \begin{bmatrix} a_0 b_0 & 0 & 0 & 0 \\ a_1 b_0 + a_0 b_1 & a_0 b_0 & 0 & 0 \\ a_2 b_0 + a_0 b_2 & 0 & a_0 b_0 & 0 \\ a_3 b_0 + a_2 b_1 + a_1 b_2 + a_0 b_3 & a_2 b_0 + a_0 b_2 & a_1 b_0 + a_0 b_1 & a_0 b_0 \end{bmatrix},$$

$$= \begin{bmatrix} c_0 & 0 & 0 & 0 \\ c_1 & c_0 & 0 & 0 \\ c_2 & 0 & c_0 & 0 \\ c_3 & c_2 & c_1 & c_0 \end{bmatrix} = \boldsymbol{C}. \tag{66}$$

From previous result, it was verified that $M(a^* b^*) = M(a^*)M(b^*)$. It is also straightforward to notice that $M(a^* + b^*) = M(a^*) + M(b^*)$. Therefore, the function $M(x^*)$ maps a valid matrix representation of the bidual number number x^*, and there is an algebra isomorphism between the bidual algebra and the algebra of the matrices generated by M.

B QUALITY OF THE MATRIX FUNCTION IMPLEMENTATIONS FOR COMPUTING DERIVATIVES

As discussed in Section 2.9, functions of matrices have been traditionally used as a method for approximating multicomplex functions, in particular, for the computation of high-order derivatives. However, most widely used methods to approximate matrix functions, such as those included in MATLAB [22] or the open-source Python library SciPy [12], are not designed to work with Cauchy-Riemann matrices of multicomplex numbers, i.e., diagonally dominant matrices produced by very

Table 18. Accuracy Comparison for the Computation of the Second-order Derivative of $\log(x)$ at $x = e^2$, Using a Stepsize $h = 10^{-10}$

Algebra	Method	2nd derivative	Relative error
Bicomplex	TS	−1.8315638888734179e-02	0.
	ISSQ	−3.8302694349567895e+05	2.09125e+07
	GS	−1.8315638888734161e-02	9.47127e-16
Bidual	TS	−1.8315638888734179e-02	0.
	ISSQ	−1.8315638888734224e-02	2.46253e-15
	GS	−1.8315638888734161e-02	9.47127e-16

Methods: (TS) Truncated Taylor Series, (ISSQ) Inverse Scaling and Squaring, (GS) Gregory's Series.

Table 19. Accuracy Comparison for the Computation of the Second-order Derivative of $\sqrt{x}$ at $x = 16$, Using a Stepsize $h = 10^{-10}$

Algebra	Method	2nd derivative	Relative error
Bicomplex	TS	−3.9062500000000000e-03	0.
	Schur	−2.8865798640254070e+05	7.38964e+07
	DB	−3.9062500000000009e-03	2.22045e-16
Bidual	TS	−3.9062500000000000e-03	0.
	Schur	−3.9062500000000000e-03	0.
	DB	−3.9062500000000009e-03	2.22045e-16

Methods: (TS) Truncated Taylor Series, (Schur) Schur's Decomposition, (DB) Denman and Beavers Iterative Method.

Table 20. Suitability of Different Matrix Function Methods Using Multicomplex and Multidual CR Matrix Inputs

Matrix func.	Method	Suitable for a CR matrix	
		Multicomplex	Multidual
Exponential	SSQ	Yes	Yes
Sine, Cosine	SSQ	Yes	Yes
Logarithm	ISSQ	No	Yes
	GS	Yes	Yes
Square root	Schur	No	Yes
	DB	Yes	Yes

Methods: (SSQ) Scaling and Squaring, (ISSQ) Inverse Scaling and Squaring, (GS) Gregory's Series, (Schur) Schur's decomposition, (DB) Denman and Beavers iterative method.

small imaginary parts. Note, first-order derivative calculations are performed using complex functions, which are intrinsic to most programming languages.

However, the availability of matrix function libraries in different programming languages offers a good opportunity to accelerate the adoption of other hypercomplex algebras for the computation of high-order derivatives, e.g., multidual algebra. To the best of the authors' knowledge, matrix functions have not been used with Cauchy-Riemann matrices of multidual numbers.

In this section, the accuracy of the hypercomplex truncated Taylor series expansion for computing derivatives is compared against matrix functions included in Scipy 1.0.0. If the matrix function implementation failed to compute accurate results, an alternative method was tested. Evaluated functions were: exponential, logarithm, sine, square root, and fractional matrix power.

Scipy's implementation of the exponential of a matrix is based on a scaling and squaring algorithm [1], and it provides machine-precision results for computing derivatives using the Cauchy-Riemann matrix of multicomplex and multidual numbers. The sine and the cosine of a matrix implementations in SciPy are based on the exponential function and yield accurate derivative results as well.

The matrix logarithm function included in Scipy is based on an inverse scaling and squaring algorithm [2]. This method produces machine-precision-accurate results when the Cauchy-Riemann matrix of a multidual number is used as input, even when a very small step size h is used, and fails when the matrix comes from a multicomplex number. In contrast, the use of a Gregory's series [11] yields good results in both multicomplex and multidual cases. An accuracy comparison for the computation of the matrix logarithm is shown in Table 18.

The square root of a matrix is implemented in Scipy using a blocked Schur algorithm [5], which fails to compute accurate derivate results for multicomplex Cauchy-Riemann matrices, but yields accurate results for multidual Cauchy-Riemann matrices. The Denman and Beavers iteration method [6] is an alternative that yields accurate derivative results in both the multicomplex and multidual cases. The accuracy comparison is shown in Table 19.

Results in Tables 18 and 19 show that the truncated Taylor series method (TS) provided exact results in the multicomplex and the multidual algebra cases. In the multicomplex algebra case, some methods included in Scipy fail to compute a second-order derivative. However, all methods provide exact or machine-precision results in the multidual algebra case, even when a small step size h is used. The summary of the matrix function methods studied for this test is shown in Table 20. Based on these results, the truncated Taylor series method is used for both multicomplex and multidual algebras.

REFERENCES

[1] A. H. Al-Mohy and N. J. Higham. 2010. A new scaling and squaring algorithm for the matrix exponential. *SIAM J. Matrix Anal. Appl.* 31, 3 (2010), 970–989. DOI: https://doi.org/10.1137/09074721X

[2] A. H. Al-Mohy and N. J. Higham. 2012. Improved inverse scaling and squaring algorithms for the matrix logarithm. *SIAM J. Sci. Comput.* 34, 4 (2012), C153–C169. DOI: https://doi.org/10.1137/110852553

[3] W. K. Anderson, J. C. Newman, D. L. Whitfield, and E. J. Nielsen. 2001. Sensitivity analysis for Navier-Stokes equations on unstructured meshes using complex variables. *AIAA J.* 39, 1 (2001), 56–63. DOI: https://doi.org/10.2514/2.1270

[4] A. Cohen and M. Shoham. 2015. Application of hyper-dual numbers to multibody kinematics. *J. Mechan. Robot.* 8, 1 (08 2015), 011015–011015–4. DOI: https://doi.org/10.1115/1.4030588

[5] E. Deadman, N. J. Higham, and R. Ralha. 2013. Blocked Schur algorithms for computing the matrix square root. In *Applied Parallel and Scientific Computing*, Pekka Manninen and Per Öster (Eds.). Springer Berlin, 171–182. DOI: https://doi.org/10.1007/978-3-642-36803-5_12

[6] E. D. Denman and A. N. Beavers. 1976. The matrix sign function and computations in systems. *Appl. Math. Comput.* 2, 1 (1976), 63–94. DOI: https://doi.org/10.1016/0096-3003(76)90020-5

[7] J. A. Fike and J. J. Alonso. 2011. The development of hyper-dual numbers for exact second-derivative calculations. In *Proceedings of the 49th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition (Aerospace Sciences Meetings)*, Vol. 886. Orlando, FL. DOI: https://doi.org/10.2514/6.2011-886

[8] J. Garza. 2014. *Multicomplex Variable Differentiation in Probabilistic Analysis and Finite Element Models of Structural Dynamic Systems (Accession no. 3670563).* Doctoral Dissertation, The University of Texas at San Antonio. ProQuest Dissertations Publishing.

[9] J. Garza and H. Millwater. 2015. Multicomplex Newmark-beta time integration method for sensitivity analysis in structural dynamics. *AIAA J.* 53, 5 (2015), 1188–1198. DOI: https://doi.org/10.2514/1.J053282

[10] J. Garza and H. Millwater. 2016. Higher-order probabilistic sensitivity calculations using the multicomplex score function method. *Probab. Eng. Mech.* 45 (2016), 1–12. DOI: https://doi.org/10.1016/j.probengmech.2015.12.001

[11] N. J. Higham. 2008. *Functions of Matrices: Theory and Computation.* Society for Industrial and Applied Mathematics.

[12] P. Virtanen, R. Gommers, T. E. Oliphant, et al. 2020. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature Methods* 17 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2.

[13] I. L. Kantor and A. S. Solodovnikov. 1989. *Hypercomplex Numbers: An Elementary Introduction to Algebras*. Springer-Verlag.
[14] G. Lantoine, R. P. Russell, and T. Dargent. 2012. Using multicomplex variables for automatic computation of high-order derivatives. *ACM Trans. Math. Softw.* 38, 3, Article 16 (Apr. 2012), 21 pages. DOI: https://doi.org/10.1145/2168773.2168774
[15] J. N. Lyness and C. B. Moler. 1967. Numerical differentiation of analytic functions. *SIAM J. Numer. Anal.* 4, 2 (1967), 202–210. DOI: https://doi.org/10.1137/0704019
[16] J. R. R. A. Martins, P. Sturdza, and J. J. Alonso. 2003. The complex-step derivative approximation. *ACM Trans. Math. Softw.* 29, 3 (Sept. 2003), 245–262. DOI: https://doi.org/10.1145/838250.838251
[17] M. Metcalf, J. Reid, and M. Cohen. 2011. *Modern Fortran Explained*. OUP Oxford.
[18] H. Millwater, D. Wagner, A. Baines, and A. Montoya. 2016. A virtual crack extension method to compute energy release rates using a complex variable finite element method. *Eng. Fract. Mech.* 162 (2016), 95–111. DOI: https://doi.org/10.1016/j.engfracmech.2016.04.002
[19] G. B. Price. 1990. *An Introduction to Multicomplex Spaces and Functions*. Taylor & Francis.
[20] W. Squire and G. Trapp. 1998. Using complex variables to estimate derivatives of real functions. *SIAM Rev.* 40, 1 (1998), 110–112. DOI: https://doi.org/10.1137/S003614459631241X
[21] M. Tanaka, D. Balzani, and J. Schröder. 2016. Implementation of incremental variational formulations based on the numerical calculation of derivatives using hyper dual numbers. *Comput. Meth. Appl. Mech. Eng.* 301 (2016), 216–241. DOI: https://doi.org/10.1016/j.cma.2015.12.010
[22] The MathWorks Inc. 2017. MATLAB version 9.3.0 (R2017b). Natick, Massachusetts.
[23] D. A. Tortorelli and P. Michaleris. 1994. Design sensitivity analysis: Overview and review. *Inverse Prob. Eng.* 1, 1 (1994), 71–105. DOI: https://doi.org/10.1080/174159794088027573
[24] A. Voorhees, H. Millwater, and R. Bagley. 2011. Complex variable methods for shape sensitivity of finite element models. *Finite Elem. Anal. Des.* 47, 10 (2011), 1146–1156. DOI: https://doi.org/10.1016/j.finel.2011.05.003
[25] B. P. Wang and A. P. Apte. 2006. Complex variable method for eigensolution sensitivity analysis. *AIAA J.* 44, 12 (2006), 2958–2961. DOI: https://doi.org/10.2514/1.19225

Received October 2018; revised December 2019; accepted January 2020