Algorithm xxxx: Multicomplex number class for Matlab, with a focus on the accurate calculation of small imaginary terms for multicomplex step sensitivity calculations

JOSE MARIA VARAS CASADO, Department of Aeronautics, Imperial College London, London, UK
ROB HEWSON, Department of Aeronautics, Imperial College London, London, UK

A Matlab class for multicomplex numbers was developed with particular attention paid to the robust and accurate handling of small imaginary components. This is primarily to allow the class to be used to obtain n-order derivative information using the multicomplex step method for, amongst other applications, gradient-based optimization and optimum control problems. The algebra of multicomplex numbers is described as is its accurate computational implementation, considering small term approximations and the identification of principle values. The implementation of the method in Matlab is studied, and a class definition is constructed. This new class definition enables Matlab to handle n-order multicomplex numbers, and perform arithmetic functions. It was found that with this method, the step size could be arbitrarily decreased toward machine precision. Use of the method to obtain up to the 7th derivative of functions is presented, as is timing data to demonstrate the efficiency of the class implementation.

ACM Reference Format:
Jose Maria Varas Casado and Rob Hewson . 2020. Algorithm xxxx: Multicomplex number class for Matlab, with a focus on the accurate calculation of small imaginary terms for multicomplex step sensitivity calculations. *ACM Trans. Math. Softw.* 1, 1 (January 2020), 26 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

This paper describes the development of a matlab class for multicomplex numbers, including the derivation of a method to deal with the instabilities of the inverse trigonometric, fractional power and logarithm functions. The construction of the multicomplex class and associated arithmetic functions allows the development of different applications of multicomplex numbers, with this paper focusing on the use of multicomplex numbers to obtain derivatives for gradient based optimisation problems. Sensitivities are key requirements of gradient based optimizers and typically comprise first and often second derivative information. There are a number of ways to obtain sensitivities for optimisation, including the finite difference method, adjoint's or automatic differentiation and the complex step method. The calculation of higher order derivatives has proven to be a difficult task for current methods, especially when high accuracy and low computation cost is required along with easy implementation. The complex step method, which consists of adding a small imaginary component to the system input and taking the imaginary component of the output is one means of obtaining first derivative information. However, higher order derivatives can not be easily calculated as a complex number only has a real part and one imaginary component. The adaptation of this method for higher order derivatives

Authors' addresses: Jose Maria Varas Casado Department of Aeronautics, Imperial College London, London, UK, cvcgf7@gmail.com; Rob Hewson Department of Aeronautics, Imperial College London, London, UK, r.hewson@imperial.ac.uk.

Manuscript submitted to ACM

was explored before in several papers, such as in [19], but they came across limitations in the accuracy of the approach due to subtraction cancellation errors (round off errors), which could potentially negate the advantages of using the complex step method.

The application of multicomplex numbers allows the complex step method to be extended, and for n-order partial derivatives to be determined by adding multiple imaginary components in the $\mathbb{C}_n$ complex space to the system inputs. In this work a new Matlab class allowing multicomplex numbers to be used is introduced, with particular attention paid to its application for the multicomplex step method, when small imaginary terms are present. The method will be applied to obtain sensitivities when accurate derivatives of a scalar valued functions $f : \mathbb{R}^n \rightarrow \mathbb{R}$, with respect to many inputs are required. There are other potential areas of application of multicomplex numbers which will be facilitated by the development of the class described here such as in quantum mechanics [31].

1.1 Finite Difference Schemes

Finite difference schemes are one of the most common discretization methods for numerically calculating gradients by approximating them with difference equations. The derivation is a truncation of the Taylor expansion series about a point. However, the accurate approximation of derivatives using the finite difference approximation is a compromise between the truncation error arising from the Taylor series expansion and the round-off error due to the floating point approximation of the functions.

As described in [26], an approximation of the optimum value for the step size h is the one that balances the two sources of error, i.e., when the truncation and round off error are equal. This value of h^* can be approximated by;

$$h^* \approx 2\frac{\sqrt{\epsilon^*|f(x_0)|}}{\sqrt{|f''(x_0)|}} \tag{1}$$

where ϵ^* is an estimate of the maximum relative error that occurs when real numbers are represented by floating-point numbers. The function being evaluated needs to be known as well as an appropriate value of ϵ^*, which changes with the function being evaluated but also with x_0. A similar derivation is available in [26] where another approximation is provided for the cases for which the function is not known;

$$h^* \approx \sqrt{\epsilon_f} x_c, \quad x_c = \sqrt{\frac{f(x_0)}{f''(x_0)}} \approx x_0 \tag{2}$$

Where x_c denotes the 'curvature scale' and ϵ_f is the fractional accuracy with which f is computed. Again, the exact value of ϵ_f is estimated as machine accuracy, but varies with the function being evaluated and also with x_0. For calculations with additional sources of inaccuracy it may be larger, and the exact values to choose are difficult to obtain.

When applying the finite difference schemes it is of vital importance to find an optimum step size, which is function dependent and therefore not trivial to determine. Even though approximations have been developed their range of validity is limited and are generally only applicable for simple functions. When many derivative evaluations are needed this task could be tedious and add another layer of complexity and computational cost. This issue is of special concern when partial derivatives of multi-variable functions are needed, as the susceptibility to round off error for each input variable might vary considerably.

1.2 Higher Order Derivatives

One method to calculate n-order derivatives is derived in [23]. An extension of Cauchy's integral formula is used which states that;

$$\oint_C \frac{f(z)}{z - z_0} dz = 2\pi i f(z_0) \tag{3}$$

This integral representation will now be used of $f(z_0)$ to find $f'(z_0)$;

$$f'(z_0) = \frac{d}{dz} f(z_0) = \frac{d}{dz}[\frac{1}{2\pi i} \oint_C \frac{f(z)}{z - z_0} dz] = \frac{1}{2\pi i} \oint_C \frac{d}{dz}[\frac{f(z)}{z - z_0}]dz = \frac{1}{2\pi i} \oint_C \frac{f(z)}{(z - z_0)^2} dz \tag{4}$$

if this process is repeated it can be shown that the formula for any higher derivative is given by,

$$f^{(n)}(z_0) = \frac{n!}{2\pi i} \oint_C \frac{f(z)}{(z - z_0)^{n+1}} dz \tag{5}$$

This contour integral can be approximated using the trapezoidal rule. The detailed derivation is provided in [23] which ultimately arrives at;

$$f^{(n)} \approx \frac{n!}{m\varepsilon} \sum_{j=0}^{m-1} \frac{f(z_0 + \varepsilon e^{i\frac{2\pi j}{m}})}{e^{i\frac{2\pi jn}{m}}} \tag{6}$$

Where the parameter ε is associated with the step size, and m is the number of function evaluations to approximate the contour integral. This method provides a way of calculating n-order derivatives of smooth functions, without the increase in complexity associated with finite difference. This method also suffers from round-off error, arising from the summation term in Eq (6). But unlike the finite difference method, a simple method to estimate and control the roundoff error is available, again, derived in the original paper by [23]. However [20] states that this method is method is 'of little practical use'. This is attributed to the high number of function evaluations required to achieve an acceptable accuracy, which could outweigh the potential benefits of this method.

1.3 Automatic Differentiation

A further method of calculating derivatives commonly used is the Automatic Differentiation (AD) or Adjoint methods. AD is based on the chain rule and exploits the fact that all functions or processes, no matter how complicated, are composed of a combination of basic mathematical operators and functions with known derivatives. The method applies the chain rule repeatedly to these basic mathematical functions until the actual function to be evaluated is "built up". This approach has the advantage of calculating the exact derivatives of the functions (ie, it is not a numerical approximation to the function like finite differences), and hence has no truncation error. In addition, there is no round off error as there is no subtraction of almost equally valued terms, like in finite differences. It also poses an advantage over the Cauchy-trapezoidal method described above as it does not need a large number of function evaluations to achieve acceptable accuracy. Machine precision can be achieved and the computational time can potentially be reduced by several orders of magnitude compared to finite differencing [18]. Tools to implement this method usually consist of either source code transformation or operation overloading [24]. Source code transformation changes the source code of a function in the programming language by another source code to add the new instructions that compute these derivatives. This effectively extends all numbers by adding a second component (much like the imaginary part of a

complex numbers). This second component carries the value of the derivative of the function at that point. Some AD tools that have done this are ADIFOR [7], TAPENADE [17], SIMDIFF and NAGWare. The drawback of this method is that most of these tools are restricted to first and second order derivatives [18], [24]. The toolboxes employing AD that have been developed for Matlab include ADiGator [25], ADiMat [6], ADMAT [34], ADMIT-1 [10], CasADi [3, 4] and MAD [14]. These toolboxes allow for the fast computation of gradients, efficient Jacobian determination (in the case the function we are optimizing is $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$), and 2^{nd} derivative calculation of Hessian matrices [5]. The use of multicomplex numbers provides a means of obtaining sensitivities of all terms which are dependent on the multicomplex input, this is especially useful, for example, when a very large number of constraints are present, as might be the case for open-loop trajectory optimisation, where the path of an object is defined by a number of constraints. The application of multicomplex numbers for such a case is simple, as all sensitivities of numerous constraints to the input are automatically determined.

1.4 Complex Step

A procedure for calculating the derivative of a holomorphic function [16] was derived by Squire and Trapps [30], which extends the finite difference method into the complex space in order to remove the round-off error and hence the necessity to choose an appropriate step size. Starting from the Taylor expansion of an infinitely differentiable function, but evaluating the function at $x = x_0 + ih$;

$$f(x_0 + ih) = f(x_0) + ihf'(x_0) - \frac{h^2}{2!}f''(x_0) - \frac{ih^3}{3!}f'''(x_0) + \frac{h^4}{4!}f''''(x_0) + \ldots \tag{7}$$

Taking imaginary parts of both sides of the equation the following is obtained,

$$\Im(f(x_0 + ih)) = hf'(x_0) - \frac{h^3}{3!}f'''(x_0) + \ldots = hf'(x_0) + R1(h) \tag{8}$$

where the imaginary part of the Taylor series is truncated after the first term, and $R1(h) = O(h^3)$. If this is then solved for $f'(x_0)$ the following is obtained

$$f'(x_0) = \frac{\Im(f(x_0 + ih))}{h} + O(h^2) \tag{9}$$

This method gives a way of calculating the first derivative of a function without the risk of roundoff error, as there is no subtraction term. Therefore, the step size can be made arbitrarily small to achieve machine precision and avoid the errors associated with truncating the Taylor series. In programming languages that accept complex numbers as function inputs such as Matlab the implementation of this method is straightforward.

1.5 Complex Step Higher Derivatives and the Motivation for the Multicomplex Step Method

A similar derivation for the calculation of second derivatives is provided in [19]. Here the real component of the Taylor series is taken;

$$\Re(f(x_0 + ih)) = f(x_0) - \frac{h^2}{2!}f''(x_0) + \frac{h^4}{4!}f''''(x_0) + \ldots = f(x_0) - \frac{h^2}{2!}f''(x_0) + R1(h) \tag{10}$$

Following the same approach as before, $f''(x_0)$ is solved for,

$$f''(x_0) = \frac{2(f(x_0) - \Re(f(x_0 + ih)))}{h^2} + O(h^2) \tag{11}$$

As can be seen, this method has the potential to suffer from round-off errors similar to those encountered when using the finite different method. In [19], higher accuracy formulas for second derivatives are derived by repetitively applying Richardson extrapolations. Instead of trying to reduce the step size arbitrarily, the truncation error is reduced by evaluating the function at different parts of the complex plane ($x_0 + i^{0.5}h$ for example). This gives a more stable method where a larger step size can be used, as the method produces very small truncation errors, several orders smaller than standard finite difference.

It is perhaps intuitive to ask, could the complex step method be applied recursively to calculate higher order derivatives? Following the definition for $f'(x_0)$,

$$f''(x_0) \approx \frac{\Im(f'(x_0 + ih))}{h} \approx \frac{\Im(\frac{\Im[f(x_0+ih+ih)]}{h})}{h} = \frac{\Im[\Im(f(x_0 + 2ih))}{h^2} \tag{12}$$

When trying to do this, the motivation for using multicomplex numbers becomes clear. The operation $\Im[\Im(f(x_0 + 2ih))]$ will always yield 0 as a result, because the two imaginary components that are added to the input variable are in the same complex space. However, if this method can be extended by adding imaginary components of n-different complex spaces a method to calculate n-order derivatives is realised. The derivation for such method was first explored by Lantoine in [20], and will be further described below with particular attention to the derivation of complex number algebra for the numerical calculation of derivatives, i.e. in formulations which account for small imaginary terms.

In order to implement the multicomplex step method in Matlab, a class definition was required as Matlab only recognizes complex numbers up to $\mathbb{C}_1$. A bicomplex class was been created by Verheyleweghen [33], this class defines the 'bicomplex object', which is of the form $z = z_1 + i_2 z_2$ but not for higher numbers of imaginary terms.

2 THE ALGEBRA OF MULTICOMPLEX NUMBERS

In order to introduce the generalization of the theory of multicomplex numbers, the underlying theory described in [21, 22, 27] is applied and particular attention is paid to small imaginary terms. This will allow the algebra of multicomplex numbers to be introduced and understand why it is a generalization of the field of complex numbers.

Two real-valued functions u, v from $\mathbb{R}^2 = \{(x_1, x_2) : x1 \in \mathbb{R}, x_2 \in \mathbb{R}\}$ to $\mathbb{R}$ will now be considered. It is known that if this pair of functions satisfy the Cauchy-Riemann equations, then $\mathbf{f}(x_1 + ix_2) := u(x_1, x_2) + iv(x_1, x_2)$ is a *holomorphic* function as it admits a complex derivative. Instead of considering (u, v) as a point in $\mathbb{R}^2$, a new space $\mathbb{C}$ is considered, where the elements are composed of a real and an imaginary part, i.e. $z = u + iv$. In other words, the complex function $f : \mathbb{C} \to \mathbb{C}$ can be decomposed into $u : \mathbb{R}^2 \to \mathbb{R}$ and $v : \mathbb{R}^2 \to \mathbb{R}$.

Considering two complex functions u, v from $\mathbb{C}^2 = \{(z_1, z_2) : z1 \in \mathbb{C}, z_2 \in \mathbb{C}\}$ to $\mathbb{C}$. In this case the pair (u, v) can be interpreted as a map of $\mathbb{C}^2$ onto $\mathbb{C}$. The implications for (u, v) satisfying the following set of equations can then be considered,

$$\frac{\partial u}{\partial z_1} = \frac{\partial v}{\partial z_2} \quad \& \quad \frac{\partial u}{\partial z_2} = -\frac{\partial v}{\partial z_1} \tag{13}$$

These could be thought of the analogue of the Cauchy-Riemann equations but for a $\mathbb{C}^2$ onto $\mathbb{C}$ mapping, and replacing differentiability by holomorphicity. Instead of considering (u, v) as a point in $\mathbb{C}^2$, a new space with elements of the form $z = u + i_2 v$, where $i_2 = \sqrt{-1}$ is another imaginary unit which commutes with the original i is built up. Therefore $\mathbf{g}(z_1 + i_2 z_2) := u(z_1, z_2) + i_2 v(z_1, z_2)$, and this new space can be denoted $\mathbb{C}_2$. Here, the bicomplex function $g : \mathbb{C}_2 \to \mathbb{C}_2$ can be decomposed into $u : \mathbb{C}^2 \to \mathbb{C}$ and $v : \mathbb{C}^2 \to \mathbb{C}$.

This process can be repeated to build up multicomplex numbers, which are defined in terms of two variables in the underlying complex space. The definition for the set of multicomplex numbers of order n is arrived at [27]

$$\mathbb{C}_n = \{\zeta_n = \zeta_{n-1,1} + i_n\zeta_{n-1,2} : \zeta_{n-1,1}, \zeta_{n-1,2} \in \mathbb{C}_{n-1}\} \tag{14}$$

Where all imaginary components i_n commute between each other and have the property ${i_n}^2 = -1$. From this definition, it is clear that any multicomplex number in $\mathbb{C}_n$ can be represented with 2^n coefficients in $\mathbb{R}$. An example of this is given below for a multicomplex number in $\mathbb{C}_2$, this is termed a bicomplex number [11, 12, 28, 29]

$$\mathbb{C}_2 = \{\zeta_2 = x_{1,1} + i_1x_{1,2} + i_2(x_{2,1} + i_1x_{2,2}) : x_{1,1}, x_{1,2}, x_{2,1}, x_{2,2} \in \mathbb{C}_0 = \mathbb{R}\} \tag{15}$$

It is also important to note that multicomplex numbers can be represented as matrices of real coefficients. The proof for this was formulated by Price [27]. This result is of importance as it provides an easy way of representing these numbers and to do arithmetic operations easily, using matrix algebra. It will be demonstrated that this representation is the most efficient way of computing multiplication and divisions of multicomplex numbers. It is based on the fact that a complex number can be represented as a 2×2 matrix and on the original proposition by Price [27]. The extension for this, in multicomplex space is therefore:

$$\zeta_n = \begin{bmatrix} \zeta_{n-1,1} & -\zeta_{n-1,2} \\ \zeta_{n-1,2} & \zeta_{n-1,1} \end{bmatrix} \tag{16}$$

It is worth noting that these ζ_{n-1} coefficients can be themselves represented with a 2x2 matrix of coefficients of the underlying complex space. Hence, the $\mathbb{C}_n$ multicomplex number can be represented by a $2^n \times 2^n$ matrix of $\mathbb{R}$ coefficients.

3 DERIVATION OF THE MULTICOMPLEX STEP METHOD

Before presenting the derivation for the multicomplex step method, it is important to consider which functions can be defined in the multicomplex space, as the method requires functions which can accept multicomplex inputs. The definitions given below are originally derived by Price in [27]. As we can see from the *complexified* Cauchy-Riemann system, the functions (u, v) are required to be holomorphic in order to build up $\mathbb{C}_2$. Hence, for the build up of $\mathbb{C}_n$, the functions are required to be holomorphic in $\mathbb{C}_{n-1}$. A function $f : \mathbb{C}_n \to \mathbb{C}_n$ is complex differentiable at $z_0 \in \mathbb{C}_n$ if the limit $f'(z_0)$ exists.

$$f'(z_0) = \lim_{z \to z_0} \frac{f(z) - f(z_0)}{z - z_0} \tag{17}$$

A function f is holomorphic in an open set $U \subset \mathbb{C}_n$ if $f'(z)$ exists for all $z \subset U$. According to [20], this definition is not very restrictive and most functions are holomorphic in $\mathbb{C}_n$. The following derivation of the multicomplex step method is based on that of [20] and [33]. In the following derivations it is assumed that k and n are positive integers. Assuming f is a holomorphic function in $\mathbb{C}_n$, we can write the Taylor series of f around the real point x_0 as:

$$f(x_0 + i_1h + \ldots + i_nh) = f(x_0) + (i_1h + \ldots + i_nh)f'(x_0) + \ldots = \sum_{k=0}^{\infty} \left(\left(\sum_{l=1}^{n} i_l h \right)^k \frac{f^k}{k!} \right) \tag{18}$$

Truncating this Taylor series after $k = n + 1$ results in,

$$\sum_{k=0}^{\infty}\left(\left(\sum_{l=1}^{n} i_l h\right)^k \frac{f^k}{k!}\right) = \sum_{k=0}^{n}\left(\left(\sum_{l=1}^{n} i_l h\right)^k \frac{f^k}{k!}\right) + (i_1 + ... + i_n)^{n+1} h^{n+1} \frac{f^{n+1}(x_0)}{(n+1)!} + O(h^{n+2}) \tag{19}$$

If the multinomial theorem to re-express $(\sum_{l=1}^{n} i_l h)^k$ is used, it can be proven that the only term in the Taylor expansion between $k = 0$ and $k = n$ that contains the product $i_1 i_2 ... i_n$ (which corresponds to the 'last' term of the multicomplex number in $\mathbb{C}_n$, ie, the coefficient that contains all the imaginary parts), is the term containing $f^{(n)}$, ie, when $k = n$. From the multinomial theory,

$$\left(\sum_{l=1}^{n} i_l h\right)^k = h^k \sum_{k_1+...+k_n=k} \frac{k!}{k_1!...k_n!} i_1^{k_1} ... i_n^{k_n} \tag{20}$$

This is because for the term to contain the product $i_1 i_2 ... i_n$, each $k_1, k_2 ... k_n$ must be equal to any odd integer. For example, for $k = n$, $\{k_1 = 1, k_2 = 1 ... k_n = 1\}$. For any $k < n$, at least one of the $k's$ is equal to 0, which is enforced by the fact that $k_1 + ... + k_n = k$. If say, $k_2 = 0$, then $i_2^{k_2} = i_2^0 = 1$ and the term will not contain the $\mathbb{C}_2$ imaginary unit. Following the derivation for the $k = n$ case, $k_1!...k_n! = 1$ and hence

$$\left(\sum_{l=1}^{n} i_l h\right)^n = h^n n! \prod_{l=1}^{n} i_l \tag{21}$$

Considering the case for $k = n + 1$, it can be shown that for any n, $(i_1 + ... + i_n)^{n+1}$ will *not* contain the product $i_1 i_2 ... i_n$. Again enforced by $k_1 + ... + k_n = k$, if $k = n + 1$ there must be at least one k which equals 2 (assuming k and n are positive integers). This, as explained before, means that the product $i_1 i_2 ... i_n$ will not appear in the $n + 1$ term of the Taylor series, as $i^2 = -1$ and one of the imaginary terms vanishes. If this is extended to any $k > n + 1$, the assumption of no $i_1 i_2 ... i_n$ product breaks down, which is why the Taylor series is truncated before $k = n + 2$. For example, for $k = n + 2$ and $n = 1$, the only value of k is $k_1 = 3$. Hence $i_1^3 = -i_1$.

If the Taylor expansion is given with only the terms that contain the product $i_1 i_2 ... i_n$ the following is obtained

$$f(x_0 + i_1 h + ... + i_n h) = ... + f^{(n)}(x_0) h^n \prod_{l=1}^{n} i_l + ... + O(h^{n+2}) \tag{22}$$

A new operator $\mathfrak{I}_{1..n}$ can now be defined, which extracts the coefficient of the $\mathbb{C}_n$ multicomplex number that contains all the imaginary parts, which is essentially $\mathfrak{I}_1(\mathfrak{I}_2(...\mathfrak{I}_n(\zeta_n)))$, and apply it to both sides of the equation to obtain the final definition of the derivative which we will use:

$$f^{(n)}(x_0) = \frac{\mathfrak{I}_{1..n}[f(x_0 + i_1 h + ... + i_n h)]}{h^n} + O(h^2) \tag{23}$$

As can be seen the rest of the terms in the Taylor series vanish when the operator $\mathfrak{I}_{1..n}$ is applied, as they do not contain all the $\mathbb{C}_n$ imaginary components. This result can also applied to any n-th order partial derivative of a holomorphic multi-variable function. The procedure is to add a new imaginary component to each variable you want to calculate the derivative with respect to. An example is given below to give an example of the implementation;

$$\frac{\partial^3 f(x,y,z)}{\partial x^2 \partial z}\Big|_{x_0,y_0,z_0} \approx \frac{\mathfrak{I}_{1..3}[f(x_0 + i_1 h + i_2 h, y_0, z_0 + i_3 h)]}{h^3} \tag{24}$$

4 MULTICOMPLEX CLASS DEFINITION

The multicomplex object in Matlab was described as an array of real coefficients of the real term and all the imaginary terms. An example of this is shown below for a $\mathbb{C}_3$ number

$$z_3 = a + i_1 b + i_2(c + i_1 d) + i_3(e + i_1 f + (i_2(g + i_1 h))) \quad \rightarrow \quad [a, b, c, d, e, f, g, h] \tag{25}$$

4.1 Utility Functions

Before performing any operation between two multicomplex numbers they are passed through a function *consimulti*, which converts the two numbers into the same format. The need for this will become clear in the next subsections. For example, in order to multiply a $\mathbb{C}_3$ and a $\mathbb{C}_2$ multicomplex number, the $\mathbb{C}_2$ number must be put into the same 1×8 array form as the $\mathbb{C}_3$ number. This translates into inputing a 0 coefficient in the i_3 terms. This will ensure that you are multiplying two same-sized square matrices. Below is an example of what the function does.

$$z1 = [a_1, b_1, c_1, d_1, e_1, f_1, g_1, h_1], \quad z2 = [a_2, b_2, c_2, d_2] \tag{26}$$

$$consimulti(z1, z2) \quad \rightarrow \quad z1 = [a_1, b_1, c_1, d_1, e_1, f_1, g_1, h_1], \quad z2 = [a_2, b_2, c_2, d_2, 0, 0, 0, 0] \tag{27}$$

consimulti also converts any real number into the multicomplex array form, and is the initial function in the algorithm for any operators or functions that involve a pair of numbers, such as addition, subtraction, multiplication, division, and *atan*2. The function also throws an error for arrays/matrix inputs and non-double/multicomplex inputs.

The matrix representation of multicomplex numbers (as shown in Eq (16)) makes it trivial to extract the multicomplex array object from the matrix form, as it is just the result of the first column, ie, $transpose(M(z_2)(:, 1))$ using standard Matlab notation. From this, the 'last' imaginary part, the one that contains the product $i_1 i_2 ... i_n$ is just the last entry in the input array. The function that exracts this last imaginary term will be called *imgn*.

The specific entry of the array representation of the multicomplex number that is needed depends on the order of the derivative calculation, whether it is a partial derivative or not, so the extraction of this coefficient will be left to be user inputted. However, a function $CX2$ was added to the class definition in order to aid the user in finding the correct coefficient of the array to extract when calculating second derivatives of functions with many input variables. This function inputs a $\mathbb{C}_n$ multicomplex number and extracts the $i_m i_n$ coefficient.

Another function, $inputconverter$, was included in order to make it easier for the user to create a multicomplex number with the step sizes in the correct coefficients. This function inputs the real part, the step size value, and an array with the imaginary component number where you want the step sizes to be in. For example, $inputconverter(20, [1, 2, 3], 10^{-10})$ creates the multicomplex number $20 + 10^{-10} i_1 + 10^{-10} i_2 + 10^{-10} i_3$. This function is especially useful when higher order derivatives are required. This function is not part of the class definition as it does not have a multicomplex type input and is therefore a separate function.

A function called *matrep* was developed which inputs the multicomplex number in the array form and converts it into a $2^n \times 2^n$ matrix of real coefficients. The function works by recursively inputing the two $(n-1)$ order constituents of the n-order multicomplex number in the matrix below;

$$z_n = \begin{bmatrix} z_{n-1,1} & -z_{n-1,2} \\ z_{n-1,2} & z_{n-1,1} \end{bmatrix} \tag{28}$$

For example with $z2 = [a, b, c, d]$, *matrep* undertakes the following

$$z_2 = [a, b, c, d] \quad \rightarrow \quad \begin{bmatrix} a & -b \\ b & a \end{bmatrix}, \begin{bmatrix} c & -d \\ d & c \end{bmatrix} \quad \rightarrow \quad matrep(z_2) = \begin{bmatrix} a & -b & -c & d \\ b & a & -d & -c \\ c & -d & a & -b \\ d & c & b & a \end{bmatrix} \tag{29}$$

Finally, the complex conjugate function *conj* was added, which simply performs the following operation:

$$z_n = z_{1,n-1} + i_n z_{2,n-1} \quad \rightarrow \quad \overline{z_n} = z_{1,n-1} - i_n z_{2,n-1} \tag{30}$$

4.2 Addition and Subtraction

It is extremely easy to describe the methods for the basic operators of addition and subtraction. The elements-wise addition and subtraction of the input arrays will suffice. For example,

$$z_{2,1} \pm z_{2,2} = [a_1, b_1, c_1, d_1] \pm [a_2, b_2, c_2, d_2] = [a_1 \pm a_2, b_1 \pm b_2, c_1 \pm c_2, d_1 \pm d_2] \tag{31}$$

4.3 Multiplication

For the multiplication and division of multicomplex numbers, the matrix representation of a multicomplex number is used. The process consists of first finding the matrix representation, perform the operation on the matrices using the built-in Matlab matrix operators, and converting back to the multicomplex array object form. This method takes advantage of Matlab's efficiency in vectorized operations.

4.4 Division

To perform the operation

$$z_{n,1}/z_{n,2} = z_{n,1} * {z_{n,2}}^{-1}, \tag{32}$$

$z_{n,2}$ needs to be invertible. Unlike for $\mathbb{C}_1$ complex numbers, higher dimensional complex numbers do not always satisfy this condition. The multicomplex number $z_n = z_{n-1,1} + i_n z_{n-1,2}$ is invertible if and only if $z_n * z_n^{\dagger} \neq 0$, where $z^{\dagger} = z_{n-1,1} - i_n z_{n-1,2}$ is the complex conjugate [2] of z_n. This is shown below,

$$z_n * z_n^{\dagger} = {z_{n-1,1}}^2 + {z_{n-1,2}}^2 \tag{33}$$

which upon rearranging the following is obtained,

$${z_n}^{-1} = \frac{{z_n}^{\dagger}}{{z_{n-1,1}}^2 + {z_{n-1,2}}^2} \tag{34}$$

Hence, it is not possible to divide by multicomplex numbers z_n, for which ${z_{n-1,1}}^2 + {z_{n-1,2}}^2 = 0$. These zero divisors are characterized by the equations,

$${z_{n-1,1}}^2 = -{z_{n-1,2}}^2 \quad \rightarrow \quad z_{n-1,1} = \pm i_{m<n} \; z_{n-1,2} \tag{35}$$

Noting that $\frac{1}{i_n} = -i_n$, it is clear that $\pm z_{n-1,1} i_{m<n} = z_{n-1,2}$. Substituting this characteristic equation in the definition of a multicomplex number $z_n = z_{n-1,1} + i_n z_{n-1,2}$, the general form of all the n-dimensional zero divisors $Z0_n$ is:

$$Z0_n = \lambda(1 \pm i_{m<n} \;\; i_n), \quad \lambda \in \mathbb{C}_{n-1} \tag{36}$$

Where λ is an arbitrary number in the set $\mathbb{C}_{n-1}$. What $i_{m<n}$ represents is any imaginary part with $1 \leq m < n$. If $m = n$ the trivial result $Z0_n = \lambda(0) = 0$ is obtained. It is important to note that λ can actually be in any $\mathbb{C}_m$, as long as $\mathbb{C}_m$ is a subset of $\mathbb{C}_n$. The general form of all the $\mathbb{C}_4$ zero divisors $Z0_4$, are,

$$Z0_4 = \lambda(1 \pm i_1 i_4), \quad \lambda(1 \pm i_2 i_4), \quad \lambda(1 \pm i_3 i_4), \quad \lambda \in \mathbb{C}_{0 \leq m < 4} \tag{37}$$

When the multicomplex number is in the matrix representation, the multicomplex numbers which can not be divided are those which have a singular matrix representation. A zero divisor checker was implemented in the division algorithm which outputs *multicomplex number in denominator is a zero divisor* if the determinant of the matrix representation of the denominator multicomplex number is 0. A function $Z0_n$ is included , which inputs n and m and outputs the corresponding zero divisor with $\lambda = 1$, ie, $1 \pm i_{m<n}$. As this function does not have a multicomplex input it is a separate function outside of the class definition.

Fortunately, when employing the multicomplex step method it is extremely unlikely to encounter these zero divisors. This is because in the multicomplex step method the dimension of the real part (ie where the function is being evaluated) is usually larger than the dimension of the coefficients of the imaginary components. For the number to be a zero divisor these dimensions must match (if $\lambda \in \mathbb{R}$, see λ in equation 52). Even when $\lambda \in \mathbb{C}_m$ contains different combinations of very small imaginary terms, it is very difficult for the product $\lambda(1 \pm i_{m<n} \;\; i_n)$ to be of the form of the multicomplex step method. It is therefore possible in most cases to use the flag in the code to remove the check for zero divisors as this will only slow down the computational speed.

A similar approach to the multiplication operator was employed. The multicomplex number was converted into its matrix representation and then the built in Matlab division of matrices was used. It was also observed that the division code could be based on the standard division of complex numbers method. The extension developed is:

$$\frac{z_{n,1}}{z_{n,2}} = \frac{a + i_n b}{c + i_n d} = \frac{(a + i_n b)(c - i_n d)}{(c + i_n d)(c - i_n d)} = \frac{ac + bd}{c^2 + d^2} + i_n \frac{bc - ad}{c^2 + d^2}, \quad a, b, c, d \in \mathbb{C}_{n-1} \tag{38}$$

As can be seen a division of a $\mathbb{C}_n$ multicomplex number can be expressed in terms of divisions of $\mathbb{C}_{n-1}$ multicomplex numbers. This is done iteratively until $\mathbb{R}$ is reached. This method was compared to the matrix representation method in terms of speed of computation.

As it can be seen in Figure 1, the matrix division method was faster for all of the dimensions of complex numbers tested, hence this is the method that is used. This could be attributed to the fact that for the conjugate multiplication division method you need to go through $n - 1$ iterations, and create and store another $2^n - 2$ multicomplex numbers. Furthermore, Matlab is optimized for operations involving matrices and vectors, so vectorized code often runs much faster than the corresponding code containing loops. The significant increase in computation time for the matrix method is a consequence of the need to calculate the inverse of a $2^n \times 2^n$ matrix.

4.5 Relational Logic Operators

It must be noted that there is no formal definition of the relational logic operators for $\mathbb{C}_{n>0}$. However, it is still necessary to overload these operators, as they are very common inside conditional if statements which lead to different execution branches of the code. Clearly the execution branch followed needs to be independent on whether the variable is the

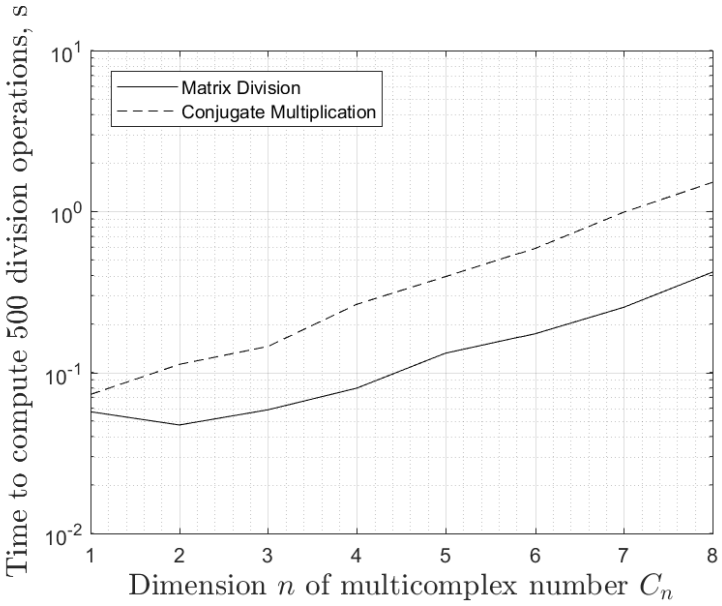

Fig. 1. Graph comparing computation time for 500 division operations for the method that uses the Matlab built in matrix division versus the one that uses the denominator conjugate multiplication.

real number, or a multicomplex number. Therefore, the relational operators were defined and use only the real parts of the multicomplex numbers so that the class definition can be used with codes which contain conditional statements.

4.6 Powers - z^p, Arctangent and Logarithm

The first consideration given to the power operator is whether the power term is an integer or a fractional power, if is the former, a simple application of the multiplication or division operator p times suffices (even if p is negative). If it is the latter however, the solution is a bit more complicated.

Initially, the built-in Matlab power of a matrix operator (*mpower*) on the matrix representation of the multicomplex number was used to calculate all types of powers. However, when the functions involved a fractional power, the error behavior was unstable for small imaginary terms. It is believed that the reason for this problem is that the method used by Matlab to calculate fractional powers of a matrix is not well suited for very small matrix entries. Clearly this is a problem if the multicomplex number class is to be used to obtain numerical derivatives. It is believed by the authors that the method that Matlab employs the 'Blocked Schur Algorithms' [8, 13]. The Schur decomposition of a matrix is computed by the QR algorithm, which is essentially an eigenvalue algorithm [15]. Small perturbations in the matrix, such as roundoff errors, can lead to large perturbations in the eigenvalues which could explain the error increase in the partial derivatives for very values of imaginary terms.

For a $\mathbb{C}_1$ complex number, Matlab returns the principal value of the square root, which is equivalent to returning only the positive root of a real number. Here De Moivre's theorem is the basis for the fractional power of a multicomplex algorithm .

$$z^p = r^p cos(p\theta) + ir^p sin(p\theta), \quad r = |z| = \sqrt{x^2 + y^2}, \quad \theta = Arg(z) = atan2(y, x) \tag{39}$$

$$z_n^p = r_n^p cos(p\theta_n) + ir_n^n sin(p\theta_n), \quad r = \sqrt{{z_{n-1,1}}^2 + {z_{n-1,2}}^2}, \quad \theta_n = Arg(z_n) \tag{40}$$

As can be seen this function needs to be of a recursive nature as for the calculation of the fractional power of a $\mathbb{C}_n$ multicomplex number, the square root of a $\mathbb{C}_{n-1}$ multicomplex number is also needed to calculate r. The $arctan$ function is not an injective function, which means again that it can have several outputs. However, the definition of θ is the *principal value* of $atan(y/x)$, which can be calculated using the Matlab function $atan2(y, x)$ for $z \in \mathbb{C}_1$ [1, 32].

This function restricts the output of $atan$ depending on the signs of the two inputs of the function. For $\mathbb{C}_{n>1}$ this becomes increasingly difficult, as you have to choose between 2^{n+1} possible outputs, as there are 2^{n+1} possible combinations of the signs of the inputs. It is possible to solve this problem with the tangent half angle formula defined over the complex plane

$$Arg(z_n) = atan2(z_{n-1,2}, z_{n-1,1}) = \begin{cases} 2atan\left(\frac{\sqrt{z_{n-1,1}^2 + z_{n-1,2}^2} - z_{n-1,1}}{z_{n-1,2}}\right) & if \;\; z_{n-1,2} \neq 0 \\ 0 \quad if \;\; z_{n-1,1} > 0 \;\; \& \;\; z_{n-1,2} = 0 \\ \pi \quad if \;\; z_{n-1,1} < 0 \;\; \& \;\; z_{n-1,2} = 0 \\ undefined \quad if \;\; z_{n-1,1} = 0 \;\; \& \;\; z_{n-1,2} = 0 \end{cases} \tag{41}$$

and

$$atan(z_n) = \frac{i_n}{2} Log\left(\frac{i_n + z_n}{i_n - z_n}\right) \tag{42}$$

Where '$if \;\; z_{n-1,2} \neq 0$' is taken to mean 'if any coefficient of the $z_{n-1,2}$ array $\neq 0$'. We can reduce any multicomplex number for which $z_{n-1,2} = 0$ to simply $z_{n-1,1}$. We will therefore always end up in the '$if \;\; z_{n-1,2} \neq 0$' category, and there is no need to formally define the inequality operators.

Considering the *log* function,

$$Log(z_n) = Log|z_n| + i_n Arg(z_n) \tag{43}$$

$$|z_n| = \sqrt{{z_{n-1,1}}^2 + {z_{n-1,2}}^2} \tag{44}$$

It should be noted that in all of the above derivations a function starting with a capital letter denotes the principal value of that function, which is non-injective. It is also important to note that there is not a formal definition of the principal value of a function for $C_{n>1}$ inputs. However, with the recursive approach presented above a formal definition can be avoided as the principal value of these functions can be expressed in terms of principal values of functions of underlying complex spaces. For example;

$$Arg(z_n) = f_1(Arg(z_{n-1})) = f_1(f_2(Arg(z_{n-2}))) = \ldots = g(Arg(z_1)) \tag{45}$$

When $\mathbb{C}_1$ is reached the formal definition of the principal value can be used [1];

$$Arg(z_1) = \{arg(z_1) - 2\pi n \quad | \quad n \in \mathbb{Z} \quad \wedge -\pi < Arg(z_1) \leq \pi\} \tag{46}$$

i.e. the value in the open-closed interval $(-\pi, \pi]$.

It could also be the case that the power term is also a multicomplex number, so the class definition performs multicomplex powers of multicomplex numbers, and of real numbers, ie,

$$x^{z_n} \quad \& \quad (z_n)^{z_m}, \quad x \in \mathfrak{R}, \quad z_n \in \mathbb{C}_n, \quad z_m \in \mathbb{C}_m \tag{47}$$

For this, logarithms of both side of equation are taken,

$$(z_n)^{z_m} = y_m \quad (if \;\; m \geq n) \quad \rightarrow \quad z_m log(z_n) = log(y_m) \tag{48}$$

and rearrange for y;

$$y_m = e^{z_m log(z_n)} \tag{49}$$

The algorithm was checked for accuracy by verifying that $((z_n)^{z_m})^{\frac{1}{z_m}} = z_n$.

4.7 Arg and Log Loss of Accuracy - Extending to n-order Derivatives

For n-order derivatives greater than 3, calculated using the multicomplex step, it was found that errors increased significantly as the imaginary terms decreased below a certain value for the *Arg*, *Log*, and consequently the power functions. The problem with the *Arg* function is the various addition and subtraction terms, which introduce round-off error. For example, when expressing *atan* in terms of *Log*, the operations $i_n + z_n$ and $i_n - z_n$ would introduce round-off error when the imaginary component of z_n, $z_{n-1,2}$ is very small. In cases where this imaginary component $z_{n-1,2} < \epsilon \approx 1.11 \times 10^{-16}$, this would lead to the incorrect result $i_n \pm i_n z_{n-1,2} = i_n(1 \pm z_{n-1,2}) = i_n$, due to the machine precision limit. This led to a loss in the accuracy of the multicomplex step method, as the calculation of the derivatives depends directly on the values of the decimals stored, and hence the error was observed to increase as step size decreased.

This only occurred for $\mathbb{C}_2$ and higher order multicomplex numbers. The multicomplex class definition only employed the recursive approach described in Section 4.6 for $\mathbb{C}_2$ and higher order numbers, and used the built in Matlab function *Arg* for $\mathbb{C}_1$ numbers. For these $\mathbb{C}_1$ numbers there was no loss in accuracy, and the multicomplex step method had the expected error behavior. It was assumed that the built in Matlab algorithm for *Arg* has some type of correction for very small values of the imaginary component (much like a small angle approximation). A small angle approximation was therefore implemented for the complex *Arg*.

The derivation for this was derived from the Taylor expansion of the complex *atan*;

$$atan(z) = z - \frac{1}{3}z^3 + \frac{1}{5}z^5 + \ldots \tag{50}$$

and if $z = a + i_n b$ is substituted (a and b are used here for simplicity), and real and imaginary terms are collected, the expansion becomes

$$atan(z) = a - \frac{1}{3}a^3 + ab^2 + \frac{1}{5}a^5 - 2a^3b^2 + ab^4 + \ldots + i(b - a^2b + \frac{1}{3}b^3 + a^4b - 2a^2b^3 + \frac{1}{5}b^5 + \ldots) \tag{51}$$

which upon removing the very small terms (ie, the powers of b, which would vanish as $b \rightarrow 0$), becomes

$$atan(z) \approx a - \frac{1}{3}a^3 + \frac{1}{5}a^5 - \ldots + ib(1 - a^2 + a^4 - \ldots) \tag{52}$$

As we can see, the left hand side of the summation is exactly the same as the Taylor expansion for $atan(a)$, so further simplifying;

$$atan(z) \approx atan(a) + ib \sum_{n=0}^{\infty} (-1)^n a^{2n} \tag{53}$$

The power series $\sum_{n=0}^{\infty}(-1)^n a^{2n}$ corresponds to the Taylor expansion for the function $\frac{1}{1+a^2}$ and a itself could be a multicomplex number. In complex analysis, the radius of convergence of a power series $f = \sum_{n=0}^{\infty} D_n(z-c)^n$ centered around c is equal to the distance from c to the nearest point where f cannot be defined in a way that makes it holomorphic, [9]. The set of all points whose distance to c (where distance is defined by the modulus $|z-c|$) is strictly less than the radius of convergence is called the 'disc of convergence'. For $\frac{1}{1+z^2}$ ($c=0$), the nearest point to 0 where a singularity occurs, and hence holomorphicity is lost, is at $\pm i$. Hence, the radius of convergence is 1 and the disc of convergence if the circle of radius 1 centered around 0 in the $\mathbb{C}_1$ complex plane, $|z_1| \leq 1$, and $z_1 \neq \pm i_1$. The operation $|z_n|$ is not equal to a real number, in fact, $|z_n| \in \mathbb{C}_{n-1}$. There are several points in which the singularity occurs, $z_n = i_n, i_{n-1}, i_{n-2}...$ Therefore, it is difficult to define an equivalent check for convergence to the $\mathbb{C}_1$ example. However, a pragmatic approach was taken and it was found that if the two conditions shown next were met, it was most likely that the power series would converge,

$$||z_n|...{}_n| < 1 \quad \& \quad |z_n|_n > |(z_n)^2|_n \tag{54}$$

Where two new functions have been required to be defined which are two forms of calculating the modulus of a multicomplex number. The function $||z_n|...{}_n|$ follows a recursive approach, calculating the $\mathbb{C}_{n-1}$ modulus of the $\mathbb{C}_n$ number with the two components, and then calculates the modulus of the result of that to get a $\mathbb{C}_{n-2}$ number and so on. For example,

$$||z_2|...{}_2| = ||(a+i_1 b)+i_2(c+i_1 i_2 d)|| = \left| \sqrt{(a+i_1 b)^2+(c+i_1 d)^2} \right| \tag{55}$$

The $|z_n|_n$ function for a $\mathbb{C}_2$ number would look like;

$$|z_2|_2 = |a+i_1 b+i_2 c+i_1 i_2 d| = \sqrt{a^2+b^2+c^2+d^2} \tag{56}$$

These two functions were added to the class definition in the 'utility functions' section. $||z_n|...{}_n|$ is called '*modc*' and $|z_n|_n$ is called '*modc2*'. Then, the function '*modcheck*', calles these two forms of the modulus and performs the check (53), and outputs either *'converge'* or *'diverge'*. In the case that the number a satisfies the convergence criteria, the final formula is

$$atan(z) = atan(a+ib) \approx atan(a) + i\frac{b}{1+a^2} \tag{57}$$

It should be noted than when employing the class definition for the multicomplex step method the condition for convergence will almost always be satisfied, as the input entering $atan2$ will be of the form y/x, and y carries all the small step size components and x carries the real part, so y/x has very small coefficients. A flag was added to the atan function to enable/disable the check for convergence.

This final expression was validated with the built in Matlab *Arg* for several values of z_1 with very small imaginary components, and it turns out the results obtained exactly match the ones obtained from the built in function. This means essentially that this approximation is likely the one that the Matlab algorithm uses. This approximation was coded into the class definition and extended to higher order multicomplex numbers ($a = z_{n-1,1}$, $b = z_{n-1,2}$) using the same approach as with the rest of the functions.

Clearly a problem arises when choosing the range of values of $b = z_{n-1,2}$ to use the approximation for, as $z_{n-1,2}$ is not a real number but rather a multicomplex number with many coefficients. Originally this method was used to reduce the roundoff error in the addition and subtractions $i_n + z_n$ and $i_n - z_n$ in the logarithm expressions, so it was decided to use the coefficient multiplying only the i_n term to choose when to use the approximation. After several trials for choosing the range, it was seen that a significant loss in accuracy in the derivative calculation was observed for the case when this coefficient was less than 10^{-7} for $atan$ and 10^{-4} for Arg (note that this is the case for the double machine precision used). Clearly this is because the Arg function has the tangent half angle formula which introduces another source of round off, see equation (41). The final algorithm for $atan$ was coded with this structure;

$$if \quad |extract\_i_n(z_n)| \geq 10^{-7} \quad \rightarrow \quad atan(z_n) = \frac{i_n}{2} Log(\frac{i_n + z_n}{i_n - z_n}) \tag{58}$$

$$elseif \quad |extract\_i_n(z_n)| < 10^{-7} \quad \rightarrow \quad atan(z_n) \approx atan(z_{n-1,1}) + i_n \frac{z_{n-1,2}}{1 + (z_{n-1,1})^2} \tag{59}$$

where $extract\_i_n(z_n)$ outputs the coefficient of the i_n term, which is essentially what the $\Im_p$ operator does. The same was done with Arg, but with the cutoff being set to 10^{-4}.

The problem with the log function was in the $log|z_n| = 0.5log({z_{n-1,1}}^2 + {z_{n-1,2}}^2)$ part. When performing this calculation, the results would yield inaccurate results when ${z_{n-1,1}}^2 = a^2 \rightarrow 1$ and ${z_{n-1,2}}^2 = b^2 \rightarrow 0$. This is due to the roundoff error introduced in the addition $a^2 + b^2$. For the correction, the built in $log1p(x)$ Matlab function was employed to bypass the roundoff error associated with $log1p(x + \mathbb{O}1)$ for small x. The range of values of $b^2 = x$ to use the approximation was chosen when $|1 - a^2| < 10^{-7}$.

A comparison of the error behavior of the third derivatives calculated with the multicomplex class definition before and after the corrections is given in Figures 2 and 3.

4.8 Trigonometric and Hyperbolic Functions

The same approach of using the definition of these trigonometric functions for complex numbers and extending the theory to multicomplex numbers was employed. These were the functions defined;

$$sin(z_n) = sin(z_{n-1,1})cosh(z_{n-1,2}) + i_n cos(z_{n-1,1})sinh(z_{n-1,2}) \tag{60}$$

$$cos(z_n) = cos(z_{n-1,1})cosh(z_{n-1,2}) - i_n sin(z_{n-1,1})sinh(z_{n-1,2}) \tag{61}$$

$$sinh(z_n) = sinh(z_{n-1,1})cos(z_{n-1,2}) + i_n cosh(z_{n-1,1})sin(z_{n-1,2}) \tag{62}$$

$$cosh(z_n) = cosh(z_{n-1,1})cos(z_{n-1,2}) + i_n sinh(z_{n-1,1})sin(z_{n-1,2}) \tag{63}$$

Note that for all of the trigonometric identities above, there is no need to construct an approximation for very small imaginary components as all of the operations in the process are multiplications so there is no introduction of roundoff error.

The tan function is defined as,

$$tan(z_n) = sin(z_n)/cos(z_n) \tag{64}$$

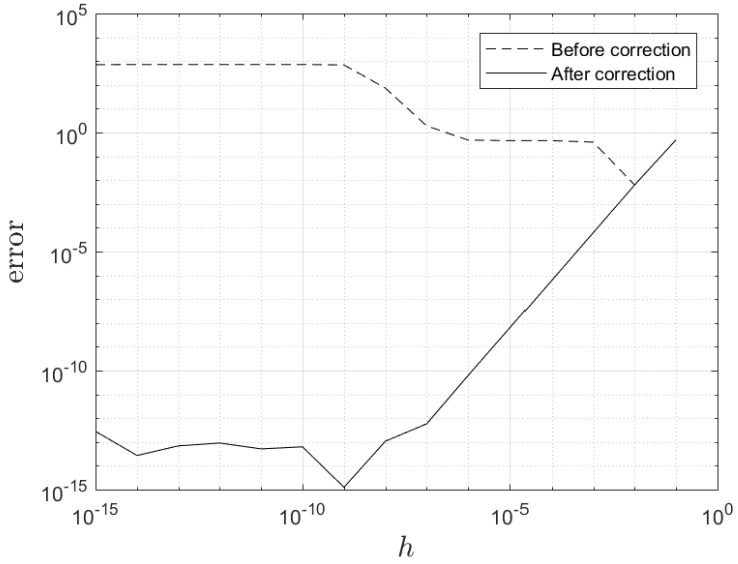

Fig. 2. Third derivative calculation for $f = \sqrt{e^{2x} + x + e^x}$ used as the optimizing function.

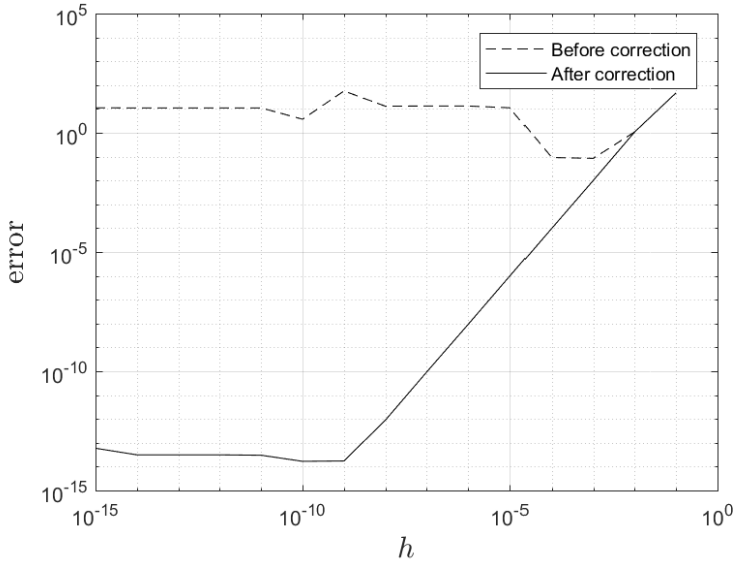

Fig. 3. Third derivative calculation for $f = \sqrt{sin(x) + x^2/cos(x)}$ used as the optimizing function.

However, the following two functions, *asin* and *acos*, (which were defined for $\mathbb{C}_1$ in [1], and here are extended to a multicomplex domain), do contain addition and subtraction terms, as we can be seen below.

$$Asin(z_n) = -i_n Log(i_n z_n + \sqrt{|1 - {z_n}^2|} e^{\frac{i_n}{2} Arg(1-z^2)}) \tag{65}$$

$$Acos(z_n) = -i_n Log(z_n + i_n \sqrt{|1 - {z_n}^2|} e^{\frac{i_n}{2} Arg(1-z^2)}) \tag{66}$$

After employing the class definition to calculate the derivative values it was seen that like with *atan*, when a sufficiently small imaginary term was employed the round off error started increasing as the small imaginary terms decreased further. A similar derivation to that used for *atan* for very small imaginary components was derived therefore for *asin* and *acos*.

The derivation for this was derived from the Taylor expansion of the complex *asin*;

$$asin(z) = z + \frac{1}{2}\frac{z^3}{3} + \frac{1}{2}\frac{3}{4}\frac{z^5}{5} + \frac{1}{2}\frac{3}{4}\frac{5}{6}\frac{z^7}{7} + \ldots \tag{67}$$

with the substitution $z = a + i_n b$ and collecting real and imaginary terms, the expansion becomes

$$asin(z) = a + \frac{1}{2}\{\frac{1}{3}a^3 - ab^2\} + \frac{1}{2}\frac{3}{4}\{\frac{1}{5}a^5 - 2a^3b^2 + ab^4\} + i[b + \frac{1}{2}\{\frac{1}{3}b^3 - ba^2\} + \frac{1}{2}\frac{3}{4}\{\frac{1}{5}b^5 - 2b^3a^2 + ba^4\}] \tag{68}$$

which upon removing the very small terms (ie, the powers of b, which would vanish as $b \to 0$), becomes

$$asin(z) \approx a + \frac{1}{2}\frac{a^3}{3} + \frac{1}{2}\frac{3}{4}\frac{a^5}{5} + ib[1 + \frac{1}{2}a^2 + \frac{1}{2}\frac{3}{4}a^4] \tag{69}$$

As can be seen, the left hand side of the summation is exactly the same as the Taylor expansion for $asin(a)$, so further simplifying;

$$asin(z) \approx asin(a) + ib \sum_{n=0}^{\infty} \frac{(2n)!}{2^{2n}(n!)^2} a^{2n} \tag{70}$$

The power series corresponds to the function $\frac{1}{\sqrt{1-a^2}}$. The procedure as was used for the *atan* function was followed. The resulting disc of convergence is the same as for the *atan* power series, $|z_1| \leq 1$ for $\mathbb{C}_1$. Therefore, for extending to the multicomplex domain, the same two convergence criteria as before, $||z_n|...n| < 1 \quad \& \quad |z_n|_n < |(z_n)^2|_n$.

The cases for which we need to use the small imaginary component approximation is not as clear for *asin* as it was for *atan*. It is essentially when the values of either $i_n z_n$ *or* $\sqrt{|1 - {z_n}^2|} e^{\frac{i_n}{2} Arg(1-z^2)}$ become small enough that they start introducing a significant round-off error. However, having to calculate those two values just to check whether we need to use the approximation was deemed too computationally expensive. Therefore, several tests were carried out to see for which range of values of the imaginary components introduced a significant round-off error. It was found that $|extract\_i_n(z_n)| < 10^{-7}$, following the value chosen for *atan* is a good approximation to use. The final expression is therefore;

$$if \;\; |extract\_i_n(z_n)| < 10^{-7} \;\; \rightarrow \;\; Asin(z_n) \approx Asin(z_{n-1,1}) + i_n \frac{z_{n-1,2}}{\sqrt{1 - {z_{n-1,1}}^2}} \tag{71}$$

For the definition of the small imaginary component *arccos*, the following was used (from [1]),

$$Acos(z_n) = \frac{\pi}{2} - Asin(z_n) \tag{72}$$

These approximations solved the issue of the error shooting rapidly increasing, and this can be seen in the derivative calculations in the performance section.

As with $atan$, $asin$ and $acos$ are non-injective so the principal value is selected by defining them in terms of the principal value of the logarithm, as well as the principal value of the multicomplex square root and argument, [1]. Due to this, care should be taken when implementing these inverse functions, as well as the log or $sqrt$ as they might lead to unexpected results. For example, $sin(arcsin(z_n))$ will always equal z_n, but in some cases $arcsin(sin(z_n)) \neq z_n$. This non-injective nature can be visualized in Figure 4.

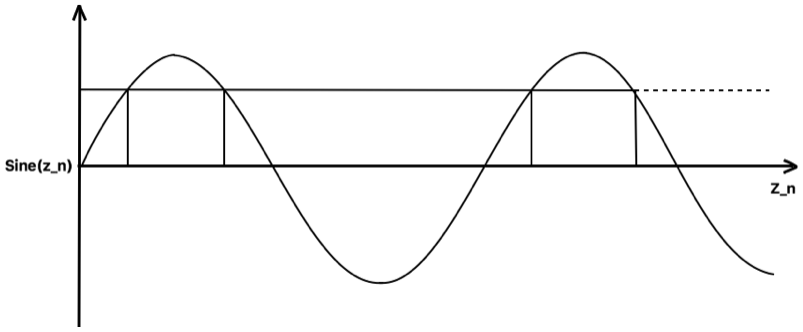

Fig. 4. *One value of z_n will only produce one output $sin(z_n)$, but one value of $sin(z_n)$ could have been produced by an infinite number of $z_n's$.*

4.9 Exponential Function

Defining an exponential function for a multicomplex number is simple due to the relationship between the trig functions and complex exponentials. As with the rest of the functions in the class definition, a recursive approach is employed in which the $\mathbb{C}_n$ multicomplex number is broken down into its two $\mathbb{C}_{n-1}$ components recursively.

$$e^z = e^{z_{n-1,1}+i_n z_{n-1,2}} = e^{z_{n-1,1}} cos(z_{n-1,2}) + i_n e^{z_{n-1,1}} sin(z_{n-1,2}) \tag{73}$$

4.10 Verification of functions with nth order derivatives using the multicomplex step method

It should be noted that the class definition functions were verified with each other, for example, the operation $exp(log(z_n))$ has to equal z_n, as well as $tan(atan(z_n)) = z_n$, $\quad \sqrt{z_n}^2 = z_n$, for any z_n. Any function with a multicomplex input will use many other functions to calculate the output. For example, the simple function$\sqrt{z_n}$ will need the $atan2$, sin and cos functions. The sin and cos will themselves need $sinh$, $cosh$, and the $atan2$ will need the log function, which also needs $atan2$ as well as the square root function, etc...

Another method of ensuring the function outputs were correct was implemented to test the new class definition. Here the multicomplex step method is used to test the accuracy of n-order derivatives.

The derivative values calculated with the class definition were compared to the ones calculated by using the exact derivative. The derivatives up to 7th order were calculated, ensuring the class definition was calculating the accurate results for the functions up to at least $\mathbb{C}_7$. A range of values of h, the size of the small imaginary terms, was used to compare the derivatives, because the algorithms for some of the functions changed depending on the sizes of the imaginary coefficients. This can be clearly visualized in the $exp(asin(z_n))$ example, where a noticeable jump in % error occurred between $h = 10^{-7}$ and $h = 10^{-8}$, this is where the switch between the standard $arcsin$ to the small imaginary

component *arcsin* algorithm occurred. The four test cases shown in Figures 5, 6, 7 and 8 were specifically chosen as together they test all of the underlying functions defined in the multicomplex class.

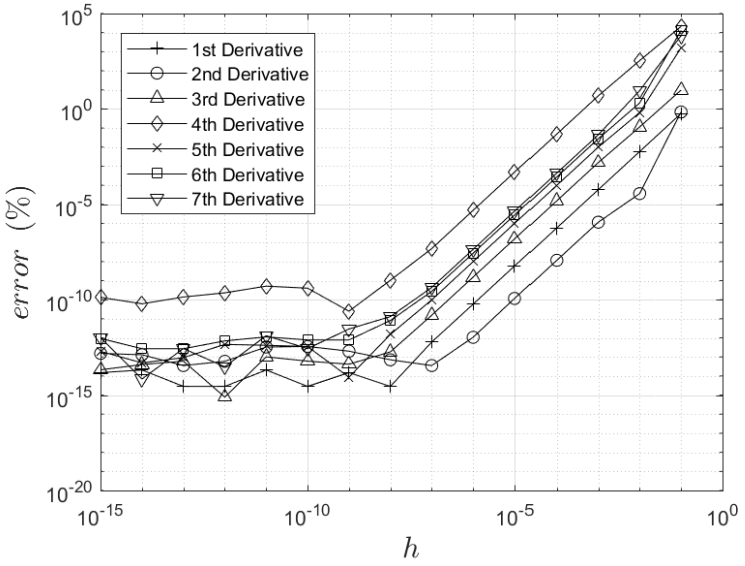

Fig. 5. Percentage error in calculating different order derivatives of the function $f(x) = \sqrt{sin(x) + \frac{x^2}{cos(x)}}$ at $x = 5$ using the multicomplex class definition.

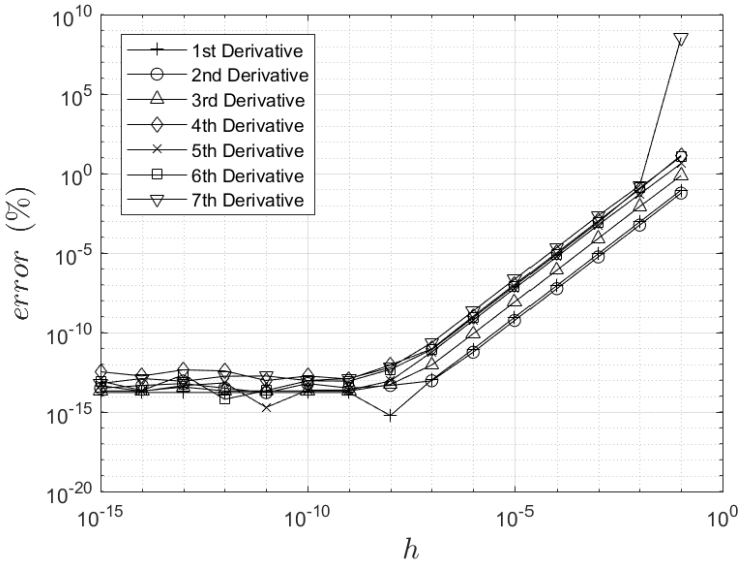

Fig. 6. Percentage error in calculating different order derivatives of the function $f(x) = x^{0.3x} + log(x)$ at $x = 2$ using the multicomplex class definition.

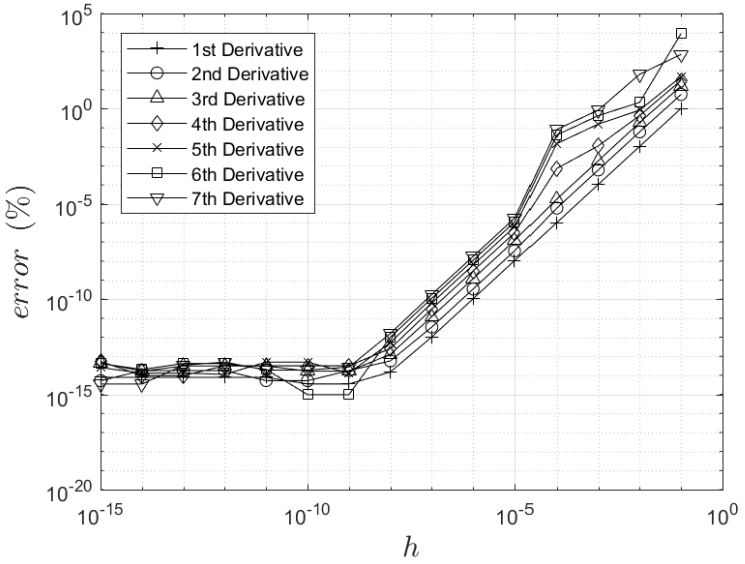

Fig. 7. Percentage error in calculating different order derivatives of the function $f(x) = exp(sin^{-1}(x))$ at $x = 0.5$ using the multicomplex class definition.

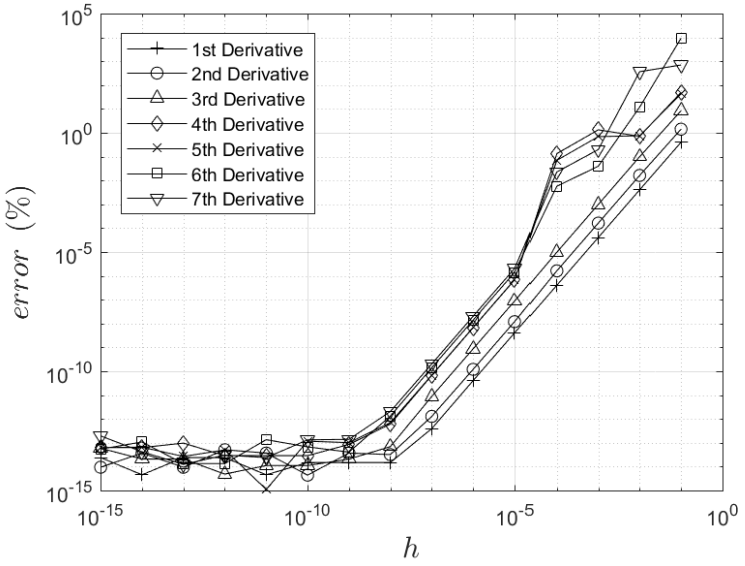

Fig. 8. Percentage error in calculating different order derivatives of the function $f(x) = exp(cos^{-1}(x)) + x$ at $x = 0.5$ using the multicomplex class definition.

5 PERFORMANCE STUDY

For the performance study, each individual function of the class definition was run for a different order multicomplex number. As the run times for the $\mathbb{C}_1$ complex number functions are of the order of $10^{-5}s$, and the resolution of the Matlab timing (tic-toc) function is $10^{-6}s$, it was decided to run 500 iterations of the functions with the exception of the addition, subtraction, multiplication and division functions which were run 10, 000 times in order to obtain an average computation time which is a more accurate value to use for comparisons (all computations were undertaken on a HP EliteDesk 800 SFF, Intel Core i7 quad processor, 8GB RAM, Windows 7 and Matlab 2015a).

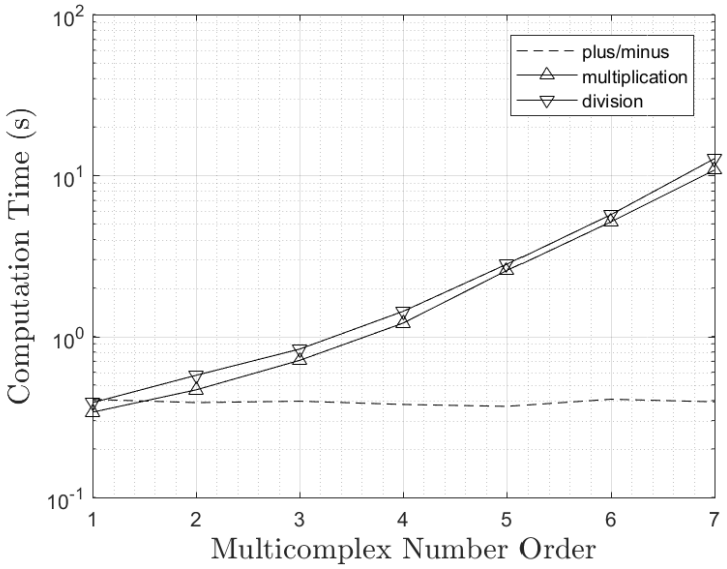

Fig. 9. Computation time for the basic operators of addition/subtraction, multiplication and division for multicomplex numbers of different order - 10000 iterations.

The computational time of the functions as the order of the multicomplex number is increased is typically exponential as can be seen in Figures 9, 10, 11 and 12. This is expected, as the way the class definition represents a multicomplex number of order n is with a 2^n array.

For addition and subtraction there was not a significant increase in computational time with order of multicomplex number, as shown in Figure 9, since what the algorithm does is essentially just an elements-wise addition/subtraction of the multicomplex arrays. In general, the computational times for the simple operators of multiplication and division are much smaller than the ones for the more complex functions of fractional powers, or powers of multicomplex numbers. This is because Matlab is optimized for operations involving matrices and vectors, so vectorized code often runs much faster than the corresponding code containing loops. This is why the algorithm for powers of multicomplex numbers was split into the categories of fractional powers and whole number powers. The method for fractional power calculation involves going through several iterations of $atan2$, $sqrt$ and log (as long as the imaginary components are large enough that small imaginary component approximation is not used), while the whole number powers just uses

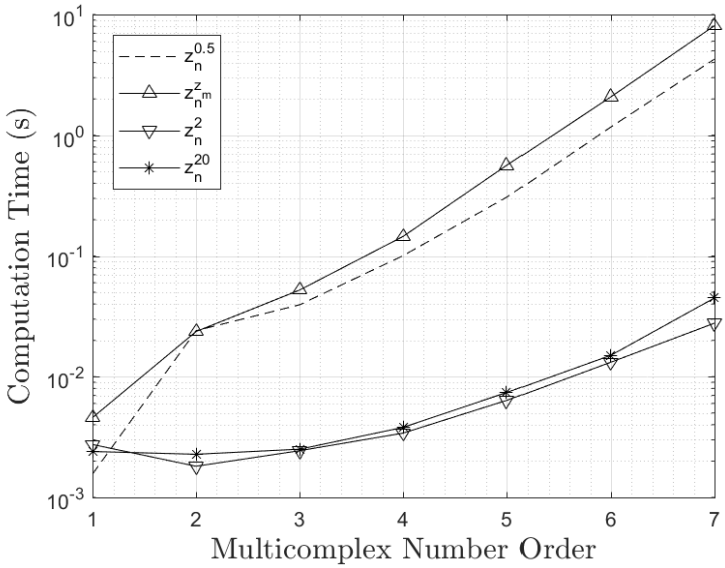

Fig. 10. Computation time for the different versions of powers for multicomplex numbers, for multicomplex numbers of different order - 500 iterations.

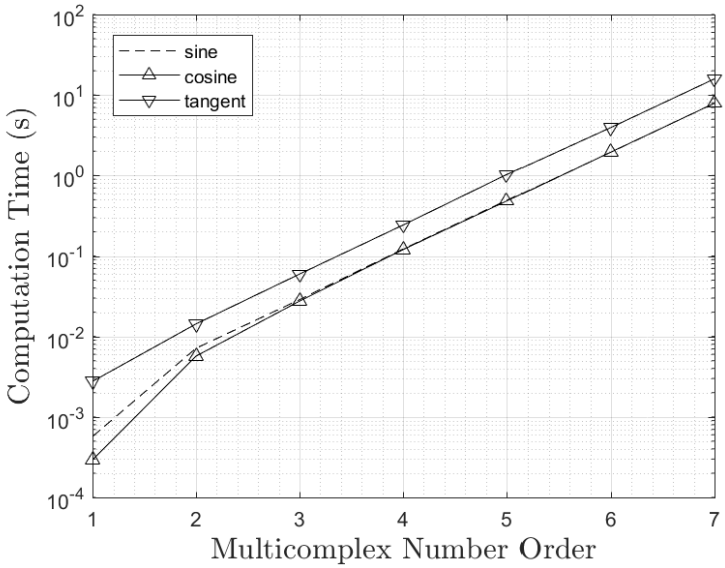

Fig. 11. Computation time for the trigonometric functions for multicomplex numbers of different order - 500 iterations.

a simple matrix multiplication p times, where p is the whole number power. As we can see in Figure 10, even when $p = 20$ (ie, 20 matrix multiplications were performed), it was still faster than the code for calculating$\sqrt{z_n}$.

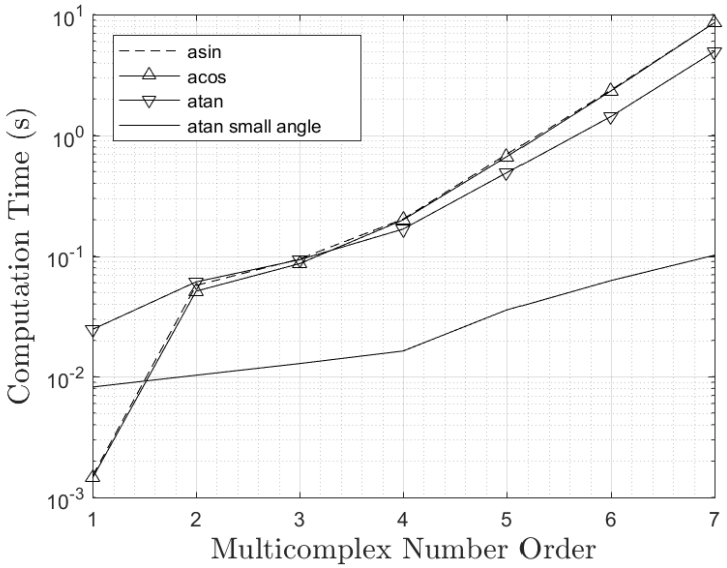

Fig. 12. Computation time for the different versions of the inverse trigonometric functions for multicomplex numbers of different order - 500 iterations.

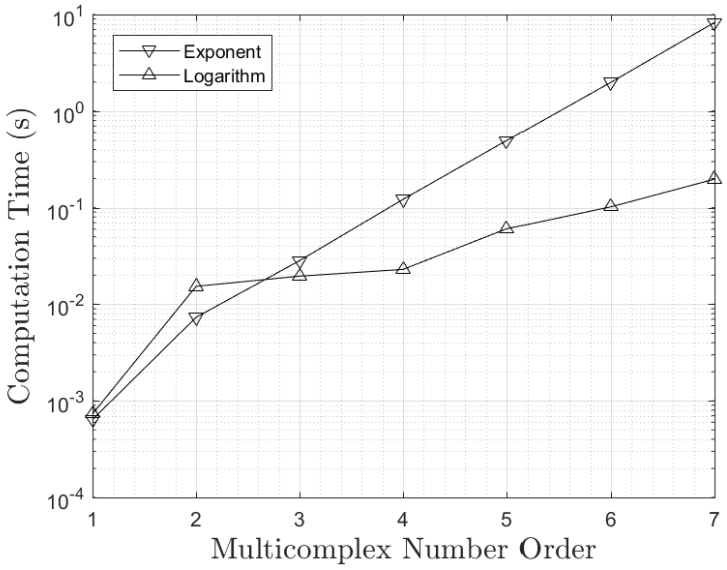

Fig. 13. Computation time for the exponential and logarithm functions for multicomplex numbers of different order - 500 iterations.

For the trigonometric functions *sin* and *cos*, the performance is shown in Figures 11. The computational times for the hyperbolic functions *sinh* and *cosh* are very similar to these results. For calculating, for example, $sin(z_n)$, 4^{n-1}

trigonometric functions of $\mathbb{C}_{m<n}$ are required. That is why Figure 11 shows an exponential behavior with varying n. The computational time for $tan(z_n)$ is more or less double, because the class definition has to calculate both $sin(z_n)$ and $cos(z_n)$ and then perform a division between them.

As the algorithm for $atan2$ changes depending on whether the imaginary component is very small ($< 10^{-7}$), the performance measurement of both algorithm is shown. It should be noted that in Figure 12, the multicomplex number order for $atan2$ corresponds to the order of the inputs of the function, not the order of the multicomplex number of the argument. The computational time for $asin$, $acos$ and $atan2$ are similar to each other, as their algorithm has essentially the same structure. For the small imaginary component $atan2$, it can be observed that there is a significant decrease in computational time. This is due to the simplicity of the algorithm, as it doesn't require any fractional powers or a log representation. From this it can be seen that the advantage of using a small step size will not only increase the accuracy of the derivative calculated, but also increase the computational efficiency of the program.

The last functions that were tested were the exponential and logarithm functions as shown in Figure 13. As with the rest of the functions there was an exponential increase in computational time with increasing n.

The final assessment of the multicomplex class was the efficiency of the class when compared to other means of obtaining sensitivity data. For this a comparison was made between the multicomplex step, finite difference and automatic differentiation as implemented in ADigator. This is not meant to be an exhaustive benchmarking exercise but, rather provide general trends on the performance of the different methods for the test case presented. The test case presented is the sensitivity of the orbital radius of a spacecraft in low Earth orbit to the perpendicular velocity of the spacecraft. The governing equation for the two body problem was

$$\mathbf{r} = \mu \frac{\mathbf{r}}{r^3} \tag{74}$$

where $\mathbf{r}$ is the spacecraft position relative to the centre of the Earth, r is the distance of the spacecraft from the centre of the Earth and μ is the gravitational parameter $398600 km^3/s^2$. The spacecraft's initial position and perpendicular velocity were, $6628 km$ and $7.726 km/s$ respectively. This was undertaken for a range of orbital times with Euler integration and $1s$ timesteps used. It can be seen from Figures 14 and 15 that the fastest approach is the finite difference one. ADigator has two performance measures, one being the length of time required to produce the sensitivity code, the other being how long the code runs. The length of time required to run the ADigator generated code is less than that required for the multicomplex step approach using the class described here, however, the generation of the code takes significantly longer than both of these.

6 CONCLUSION

The derivation and implementation of a multicomplex toolkit was developed with particular attention paid to small imaginary terms to aid in implementing the method in the multicomplex step method. The use of a class definition means that multicomplex numbers can be passed to existing codes without the need to modify algebraic functions being called, allowing the fast, accurate and reliable extraction of derivatives from existing codes. The use of the multicomplex step approach for obtaining derivative data is one of a range of tools available, limited timing information shows that there are benefits and limitations of the different methods explored, and the authors believe the multicomplex step method has a place in the researchers' toolkit.

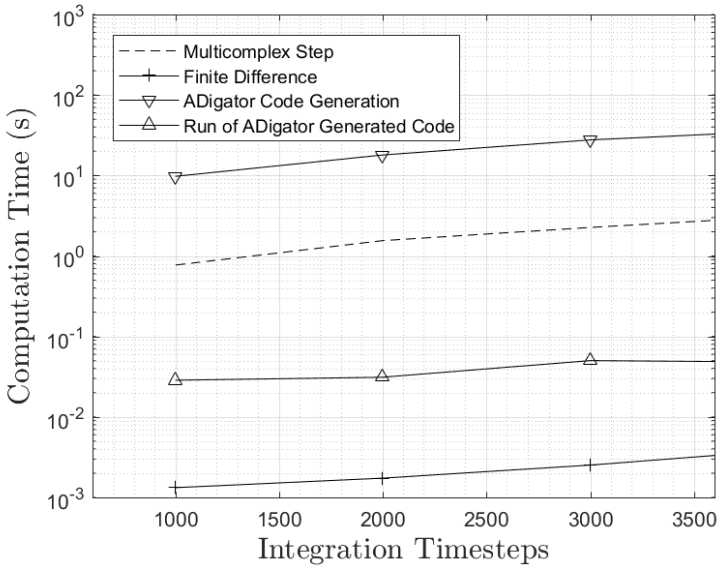

Fig. 14. Timing comparison of different methods of obtaining first derivative sensitivities to initial velocity of low Earth orbiting spacecraft after different orbital durations (timesteps of one second).

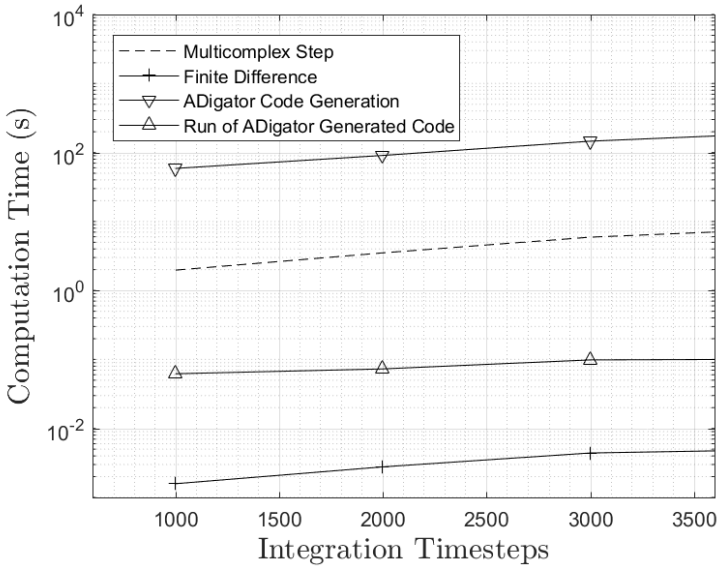

Fig. 15. Timing comparison of different methods of obtaining second derivative sensitivities to initial velocity of low Earth orbiting spacecraft after different orbital durations (timesteps of one second).

REFERENCES

[1] Abramowitz, M. and Stegun, I. A. (1965). *Handbook of mathematical functions, with formulas, graphs, and mathematical tables,*. Dover Publications.

[2] Alpay, D., Luna-Elizarrarás, M. E., Shapiro, M., and Struppa, D. C. (2014). *Basics of Functional Analysis with Bicomplex Scalars, and Bicomplex Schur Analysis*. SpringerBriefs in Mathematics. Springer.

[3] Andersson, J., Åkesson, J., and Diehl, M. (2012). CasADi: A Symbolic Package for Automatic Differentiation and Optimal Control. In *Recent Advances in Algorithmic Differentiation*, pages 297–307. Springer.

[4] Andersson, J. A., Gillis, J., Horn, G., Rawlings, J. B., and Diehl, M. (2019). CasADi: a software framework for nonlinear optimization and optimal control. *Mathematical Programming Computation*, 11(1):1–36.

[5] Avriel, M. (2003). *Nonlinear programming : analysis and methods*. Dover Publications.

[6] Bischof, C., Bucker, H., Lang, B., Rasch, A., and Vehreschild, A. (2002). Combining source transformation and operator overloading techniques to compute derivatives for MATLAB programs. In *Proceedings of the second IEEE international workshop on source code analysis and manipulation*, pages 65–72. IEEE Comput. Soc.

[7] Bischof, C., Khademi, P., Mauer, A., and Carle, A. (1996). Adifor 2.0: automatic differentiation of Fortran 77 programs. *IEEE Computational Science and Engineering*, 3(3):18–32.

[8] Björck, A. and Hammarling, S. (1983). A Schur method for the square root of a matrix. *Linear Algebra and its Applications*, 52-53:127–140.

[9] Brown, J. W. and Churchill, R. V. (2013). *Complex variables and applications*. McGraw-Hill, 8th edition.

[10] Coleman, T. F. and Verma, A. (2000). ADMIT-1: automatic differentiation and MATLAB interface toolbox. *ACM Transactions on Mathematical Software*, 26(1):150–175.

[11] Colombo, F., Sabadini, I., and Struppa, D. C. (2014). Bicomplex holomorphic functional calculus. *Mathematische Nachrichten*, 287(10):1093–1105.

[12] Colombo, F., Sabadini, I., Struppa, D. C., Vajiac, A., and Vajiac, M. (2010). Bicomplex hyperfunctions. *Annali di Matematica Pura ed Applicata*, 190(2):247–261.

[13] Deadman, E., Higham, N. J., and Ralha, R. (2013). Blocked Schur Algorithms for Computing the Matrix Square Root. pages 171–182. Springer.

[14] Forth, S. A. and A., S. (2006). An efficient overloaded implementation of forward mode automatic differentiation in MATLAB. *ACM Transactions on Mathematical Software*, 32(2):195–222.

[15] Golub, G. H. G. H. and Van Loan, C. F. (1996). *Matrix computations*. Johns Hopkins University Press.

[16] Gunning, R. C. R. C. and Rossi, H. (2009). *Analytic functions of several complex variables*. AMS Chelsea Pub.

[17] Hascoet, L. and Pascual, V. (2013). The Tapenade automatic differentiation tool. *ACM Transactions on Mathematical Software*, 39(3):1–43.

[18] Homescu, C. (2011). Adjoints and Automatic (Algorithmic) Differentiation in Computational Finance. *SSRN Electronic Journal*.

[19] Lai, K.-L. and Crassidis, J. (2006). Generalizations of the Complex-Step Derivative Approximation. In *AIAA Guidance, Navigation, and Control Conference and Exhibit*, Reston, Virigina. American Institute of Aeronautics and Astronautics.

[20] Lantoine, G., Russell, R. P., and Dargent, T. (2012). Using Multicomplex Variables for Automatic Computation of High-Order Derivatives. *ACM Transactions on Mathematical Software*, 38(3):1–21.

[21] Luna-Elizarraras, M. E., Shapiro, M. V., Struppa, D. C., and Vajiac, A. (2015). *Bicomplex holomorphic functions - the algebra, geometry and analysis of bicomplex numbers*. Springer.

[22] Luna-Elizzaras, M., Shapiro, M., Struppa, D., and Vajac, A. (2012). Bicomplex Numbers and their Elementary Functions. *Cubo (Temuco)*, 14(2):61–80.

[23] Lyness, J. N. and Moler, C. B. (1967). Numerical Differentiation of Analytic Functions. *SIAM Journal on Numerical Analysis*, 4(2):202–210.

[24] Neidinger, R. D. (2010). Introduction to Automatic Differentiation and MATLAB Object-Oriented Programming. *SIAM Review*, 52(3):545–563.

[25] Patterson, M. A., Weinstein, M., and Rao, A. V. (2013). An efficient overloaded method for computing derivatives of mathematical functions in MATLAB. *ACM Transactions on Mathematical Software*, 39(3):1–36.

[26] Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. (2007). *Numerical recipes : the art of scientific computing*. Cambridge University Press.

[27] Price, G. B. (1991). *An introduction to multicomplex spaces and functions*. M. Dekker.

[28] Ryan, J. (1982). Complexified clifford analysis. *Complex Variables and Elliptic Equations*, 1(1):19–21.

[29] Ryan, J. (2001). C2 extensions of analytic functions defined in the complex plane. *Advances in Applied Clifford Algebras*, 11(S1):137–145.

[30] Squire, W. and Trapp, G. (1998). Using Complex Variables to Estimate Derivatives of Real Functions. *SIAM Review*, 40(1):110–112.

[31] Theaker, K. A. and Van Gorder, R. A. (2017). Multicomplex Wave Functions for Linear And Nonlinear Schrödinger Equations. *Advances in Applied Clifford Aladgebras*, 27(2):1857–1879.

[32] Ukil, A., Shah, V. H., and Deck, B. (2011). Fast computation of arctangent functions for embedded applications: A comparative analysis. In *2011 IEEE International Symposium on Industrial Electronics*, pages 1206–1211. IEEE.

[33] Verheyleweghen, A. (2014). Computation of higher-order derivatives using the multi-complex step method. Technical report, NTNU: Norwegian University of Science and Technology.

[34] Verma, A. (1999). ADMAT: Automatic Differentiation in MATLAB Using Object Oriented Methods. In Henderson, M. E., Anderson, C. R., and Lyons, S. L., editors, *Object Oriented Methods for Interoperable Scientific and Engineering Computing: Proceedings of the 1998 SIAM Workshop*, pages 174–183, Philadelphia. SIAM.