Imperial College London

DEPARTMENT OF AERONAUTICS

2020 - 2021 INDIVIDUAL PROJECT

Optimisation of Multicomplex Step Computational Method with Application in Orbital Mechanics

Academic Supervisor: Dr. Robert Hewson
Second Marker: Dr. Ajit Panesar

Student: Chung Chung To
CID: 01380144
Course: MEng Aeronautical Engineering
Date: 01/06/2021

Imperial College London
South Kensington Campus
SW7 2AZ
United Kingdom

Acknowledgement

I would like to thank Dr. Rob Hewson for his support and guidance throughout this project. He also made this difficult time not so difficult by encouraging me to be confident and give me strength. I would not be able to finish this project without him. I would also like to thank Hakan Serpen for being supportive in the past three years and help with my project selection so that I can have the chance to explore this interesting topic.

Abstract

This report starts with the development history of the multicomplex-step method with its advantages over other numerical methods. It then covers the background theory of the multicomplex step and its mathematical nature. A series of optimisations are done based on the original MATLAB multicomplex class. These optimisations improve the performance of the fractional power function, complex power function and allow matrix operations to be more memory efficient. Corresponding performance analysis shows that the improvements in accuracy and computational time of optimised functions are rather significant. Finally, the class is tested by carrying out a sensitivity analysis on a relative motion problem. That shows the class is full of potential and very convenient as it can solve complex problems under the minimum number of operations.

Contents

List of Figures

List of Tables

1 Introduction

A multicomplex number is a number that consists of multiple imaginary parts. The multiple imaginary parts in a multicomplex number could be utilised as a numerical method, called the multicomplex step method. Comparing to other numerical methods, the multicomplex step method is relatively less explored and less discussed. However, the multicomplex step method is extremely powerful. In mathematical terms, the multicomplex step method is able to compute derivatives of mathematical functions up to high orders and high accuracy, which could even reach machine precision. It can solve equations that involve highly non-linear mathematical function and equations that need to substitute in large values, which are both impossible to solve by the analytical method in programming. The multicomplex step method also has a huge potential in terms of its application. For instance, in orbital mechanics, the calculation of space trajectory requires six states, displacement in x, y, z directions and velocity in x, y, z direction, in many scenarios. The multicomplex step method's high precision and accuracy allow the problem to be solved by assigning all six perturbations at the same time as imaginary parts of a multicomplex number. That allows the evaluation of the corresponding six sensitivities of six components in one integration, which greatly reduce the computational time and complexity. Other numerical methods in contrast, such as the finite difference method, require multiple integrations to achieve the same result.

A MATLAB class definition is developed based on the multicomplex step method by an Imperial College London Aeronautics undergraduate, Jose Varas Casado, as his MEng degree final year project [1] in 2018. This multicomplex class sets a solid foundation to the practicality of this numerical method. Ideally speaking, for a common orbital mechanics problem, the class definition is able to give the sensitivities of all six states of inputs in one integration with high accuracy. However, some limitations hinder the full potential of this class definition. Scaling error starts to arise with the increase in the size of the multicomplex number, which is the increase in the number of imaginary parts in a multicomplex number. That would limit the number of states the multicomplex class can take in many orbital mechanics problems. In report [2] where this multicomplex class definition was used, two integrations were needed by processing three states at each time. It is not a big problem in terms of convenience but in academia, people always aim to improve and to perfect. There are also other rooms for improvement in the existing class, which will be listed in the main part of this report. Solving these limitations would allow the class to maximise its potential and applicability in real-life problems.

Hence, the optimisation of the multicomplex class sets the backbone of this project. The key objectives of this project are: to retain and improve the accuracy of the approximation given by the multicomplex class as well as actively looking for functions and operation that can be optimised in terms of mathematical accuracy, computational time and memory efficiency.

The order of this report starts with the literature review of all the academic references used for the advance of this project. Then it gives an introduction of multicomplex number with a list of their mathematical nature as clear background knowledge for the readers. The background information also includes the derivation of the finite difference and the complex step, as a preface of the multicomplex step numerical method. This section helps with the understanding of how the multicomplex class operates. In the

following section, the multicomplex class definition, implemented in MATLAB, and its other significant operational nature was then introduced alongside with its detailed existing capability and corresponding implementation methods. The report is then followed by the main discussion of several optimisations done for this report. It includes the existing limitation of class, followed by the optimisations and corrections regarding these limitations. Both successful attempts and unsuccessful attempts are carefully explained in this section for any future references in the case when further optimisation needs to be carried out. The optimisation part of the project is then concluded by an overall performance analysis of these changes. This report is finished with a sensitivity analysis of a trajectory motion problem in the rendezvous manoeuvre of a spacecraft moving in relative motion. This sensitivity analysis was done in a derivative-based method so both the multicomplex step method and the finite difference method are used for comparison. That acts as a demonstration of the potential of the multicomplex step method and its class.

2 Literature Review

The use of complex variables to approximate derivatives was first proposed by Lyness and Moler [3] in 1967. The authors stated that the introduction of complex values into a numerical method could avoid some difficulties encountered in numerical differentiation. The complex step method is then first introduced by Squire and Trapps in 1998 [4] and is extended from the finite difference method. The complex step is a numerical method that is used to approximate the value of the first derivative of a real-valued function. Similar to the finite difference method, the complex step introduces the use of finite step size. However, this finite step size used in the complex step method is added as an imaginary part. Contrary to the finite difference, the complex step eliminates the rounding error and truncation error in the finite difference method. It also does not restrict the step size used in the calculation.

The concept of complex number was extended to bicomplex number and even multicomplex number by Price [5] prior to the first development of the complex step method. Even though Price's work was not fully utilised in the area of multicomplex number until more than 20 years after, it laid the ground for achieving higher-order derivatives. The complex step method by Squire and Trapps were extended to the bicomplex step method and multicomplex step method, which allow the second-order of the derivative and even higher order of derivative to be solved, in 2012 by Lantoine, Russell and Dargent [7]. In this paper, it is stated that the multicomplex step method had higher accuracy but a relative slower computational speed compared to the finite difference method for simple mathematical function calculation in its third order of derivative. However, the multicomplex step method demonstrates a higher accuracy as well as a faster computational time when they were used to solve spacecraft trajectory problem. The code used to demonstrate the result above was developed in Fortran 90 but it was not realised with this paper.

The idea of making a multicomplex step method available to the public was first put through by Verheyleweghen [6] in 2014. He developed a bicomplex step method MATLAB class, which allow users to solve the second derivative of the function. To extend this method to solve higher order of derivatives approximation, the multicomplex step method

was then introduced by Varas Casado [1] in 2018, which made even higher orders of derivatives possible. This project is largely based on the theories that were put forward in the papers and books that were mentioned above.

The optimisation of the existing multicomplex step method was largely based on Waugh and Abel [14], which introduced the binomial expansion method of performing fractional power. Equations from Feeman [15] was used as a reference for the computations of the complex number power. Finally, the notes [24] from Dr. Rob Hewson and the book by Curtis [23] were used to understand the concept of a relative motion problem.

3 Multicomplex number

A complex number is a number that consists of a real part and an imaginary part. It can be expressed as $x_1 + x_2 i$, where x_1 is the real part, and $x_2 i$ the imaginary part. For the imaginary part, x_2 is a real number and $i = \sqrt{-1}$. With a similar concept, a multicomplex number is a number that contain multiple imaginary parts, each as i_j and i_k where $i_j i_k = i_k i_j$ [8], as its complex space extend a higher order. For a multicomplex number, it can be expressed as a number with multiple imaginary parts or in a matrix form. This section describes both of the expression in a detailed manner.

3.1 Mathematical Expression

As mentioned above, a complex number has a real part and an imaginary part, which the coefficient of both part are real numbers. To extend the idea to a bicomplex number [5] in this example, it is in the $\mathbb{C}_2$ space and consists of a pair of complex numbers in $\mathbb{C}_1$ space. Then, each of these complex numbers consists of a pair of real numbers in $\mathbb{C}_0$ space. To express them in mathematical equation,

$$\mathbb{C}_2 = a + bi_2 = a_1 + a_2 i_1 + i_2(b_1 + b_2 i_2), \tag{1}$$

where $a, b \in \mathbb{C}_1$ and $a_1, a_2, b_1, b_2 \in \mathbb{C}_0$. Further increasing the complex space dimension to $\mathbb{C}_3$ space, it is a tricomplex number and it consists of a pair of bicomplex numbers in the $\mathbb{C}_2$ space.

$$\mathbb{C}_3 = \alpha + \beta i_3, \tag{2}$$

where $\alpha, \beta \in \mathbb{C}_2$. Following the equation (2), the same process mentioned in equation (1) would repeat until this $\mathbb{C}_3$ is represented in all real numbers $\in \mathbb{C}_0$. In other words, a multicomplex number $\mathbb{C}_n$ can be represented by in different multicomplex number systems $\mathbb{C}_m$ where $m = 1, 2, ...n$.

Figure 1 is a visual representation of the layers of constitution in each complex space dimension of a multicomplex number in the $\mathbb{C}_n$ space, which also shows the recursive nature of a multicomplex number until $\mathbb{C}_0$ space is reached. From this tree diagram, it is clear that the number of elements $\in \mathbb{C}_0$ for a multicomplex number $\in \mathbb{C}_n$ would be 2^n. To follow this tree diagram, the example from equation (2) can be further broken down. For a $\mathbb{C}_3$ to be represented in all real number would be

$$\mathbb{C}_3 = \alpha_{1,1} + \alpha_{1,2} i_1 + (\alpha_{2,1} + \alpha_{2,2} i_1) i_2 + (\beta_{1,1} + \beta_{1,2} i_1 + (\beta_{2,1} + \beta_{2,2} i_1) i_2) i_3, \tag{3}$$

where $\alpha_{1,1}, \alpha_{1,2}, \alpha_{2,1}, \alpha_{2,2}, \beta_{1,1}, \beta_{1,2}, \beta_{2,1}, \beta_{2,2} \in \mathbb{C}_0$. To fully expand equation (3) to the expression that can be taken as input of the class definition and throughout this project,

$$\mathbb{C}_3 = \alpha_{1,1} + \alpha_{1,2} i_1 + \alpha_{2,1} i_2 + \alpha_{2,2} i_1 i_2 + \beta_{1,1} i_3 + \beta_{1,2} i_1 i_3 + \beta_{2,1} i_2 i_3 + \beta_{2,2} i_1 i_2 i_3. \tag{4}$$

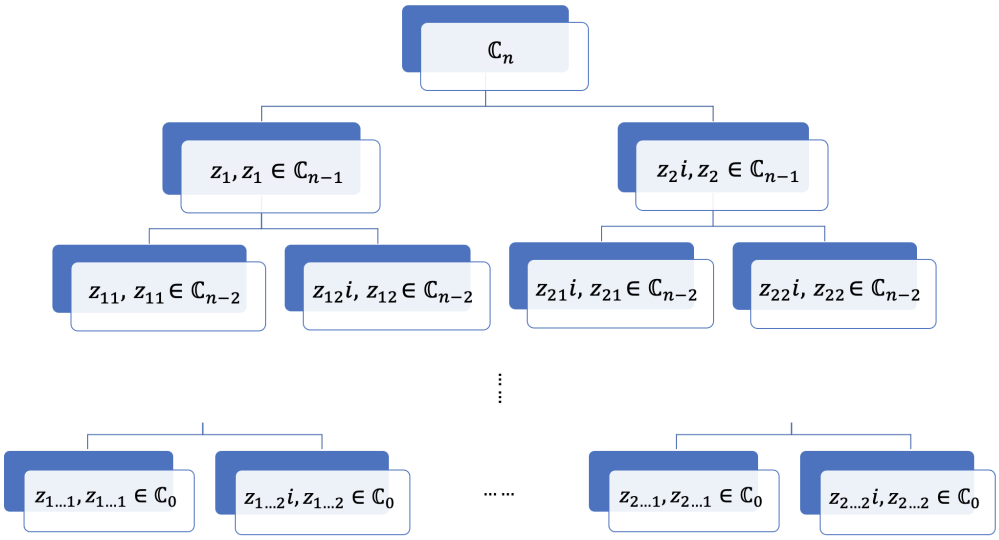


Figure 1: Tree Representation of a multicomplex number $\mathbb{C}_n$ [7]

3.2 Matrix Form Expression

Matrix operation is easy to carry out when using MATLAB as the programming languages. Furthermore, the form of a multicomplex number array would not be always suitable to be used in a calculation so matrices and their operations are widely used in a number of operations and function within the multicomplex class. Hence, it is important to have a full description of the transformation mechanism of multicomplex numbers to a matrix, which elements are all real numbers. Similar to a complex number, a multicomplex number can also be expressed in a matrix form. To begin, a complex number in its matrix form can be expressed as

$$\mathbb{C}_1 = a + bi = \begin{bmatrix} a & -b \\ b & a \end{bmatrix}, \tag{5}$$

where $a, b \in \mathbb{C}_0$. To expend this concept to a multicomplex number, it is important to keep its recursive nature in mind. The matrix form of a multicomplex number, $\mathbb{C}_n$, can be represented in all of the different domains from $\mathbb{C}_{n-1}$ space to $\mathbb{C}_0$ space. Extending this concept of the matrix form of the same bicomplex number mentioned earlier,

$$\mathbb{C}_2 = \begin{bmatrix} a & -b \\ b & a \end{bmatrix} \in \mathbb{C}_1 = \begin{bmatrix} a_1 & -a_2 & -b_1 & b_2 \\ a_2 & a_1 & -b_2 & -b_1 \\ b_1 & -b_2 & a_1 & -a_2 \\ b_2 & b_1 & a_2 & a_1 \end{bmatrix} \in \mathbb{C}_0. \tag{6}$$

The multicomplex number transforms to its matrix following the same steps as illustrated in figure 1. As shown in equation (6), it first transforms into a matrix with elements that are complex numbers $\in \mathbb{C}_1$. Then each of the complex numbers $\mathbb{C}_1$ repeats the same process to transform into elements that are real numbers $\mathbb{C}_0$. This recursive process of transforming into a matrix will be the same for higher order multicomplex number. It would always repeat the same steps until all elements in the matrix are real $\in \mathbb{C}_0$. Hence, the size of the matrix of $\mathbb{C}_n$ would be $2^n \times 2^n$ in size.

A more generalised matrix format introduced by Price [5] can be written as

$$\mathbb{C}_n = \zeta_n = \begin{bmatrix} \zeta_{n-1,1} & -\zeta_{n-1,2} \\ \zeta_{n-1,2} & \zeta_{n-1,1} \end{bmatrix}. \tag{7}$$

Each of the element, ζ_{n-1}, in equation (7) is a representation of a 2×2 matrix which can be represented by ζ_{n-2}. One final important point about multicomplex number matrix is that in the matrix in equation (6), the first column of the matrix is in the order of the MATLAB array format shown in equation (29). This project utilises these features of the multicomplex number when performing optimisation in the later stage.

4 Multicomplex Step Method

As establish in [4], complex step method was derived from finite difference method to calculate gradient. Then it was further developed into the idea of multicomplex step method in [7]. Hence, in order to understand the principle behind multicomplex step method, it is essential to start from the finite difference method and learn about the linkages among these methods and their development step by step.

4.1 Finite Difference Derivation

The finite difference method first starts with Taylor expansion about real point x_0 [9],

$$f(x) = f(x_0) + (x - x_0)f'(x_0) + (x - x_0)^2\frac{f''(x_0)}{2!} + ... + (x - x_0)^n\frac{f^n(x_0)}{n!}. \tag{8}$$

Let $x = x_0 + h$ where h is a sufficiently small step size, then

$$f(x_0 + h) = f(x_0) + hf'(x_0) + h^2\frac{f''(x_0)}{2!} + ... + h^n\frac{f^n(x_0)}{n!} + O(h^{n+1}). \tag{9}$$

For instance, the target is to find the first derivative so the Taylor's expansion is stopped after the first two terms,

$$f(x_0 + h) = f(x_0) + hf'(x_0) + O(h^2). \tag{10}$$

With the rearrangement of the expression in equation (10) in terms of the first derivative, then

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

where $O(h)$ is the truncation error that is in the order of the step size for the first derivative with this method. Hence, the finite differences method can be expressed as

$$f'(x_0) \approx \frac{f(x_0 + h) - f(x_0)}{h} \tag{12}$$

4.2 Complex Step Method Derivation

Similar to the deviation of the finite difference method, the derivation of the complex step method also started with the Taylor expansion about real point x_0 as shown in equation

(8) and a step size of h. However, in this situation the step size was added as a imaginary part such that [10]

$$f(x_0 + ih) = f(x_0) + ihf'(x_0) + i^2h^2\frac{f''(x_0)}{2!} + ... + i^nh^n\frac{f^n(x_0)}{n!} + O(h^{n+1}). \tag{13}$$

In Equation (13), $i^p = -1$ where $p = 2, 4, 6....$Hence, it can also be expressed as

$$f(x_0+ih) = f(x_0)+ihf'(x_0)-h^2\frac{f''(x_0)}{2!} - ih^3\frac{f'''(x_0)}{3!} + h^4\frac{f''''(x_0)}{4!} + ... + O(h^{n+1}). \tag{14}$$

Similarly, in order to find the expression of first derivative approximation, the expansion in equation (14) is truncated again, then take the imaginary part on both side of the equation,

$$\Im(f(x_0 + ih)) = hf'(x_0) - ih^3\frac{f'''(x_0)}{3!} + O(h^4). \tag{15}$$

Rearrange the expression for first derivative approximation,

$$f'(x_0) = \frac{\Im(f(x_0 + ih))}{h} + \frac{O(h^3)}{h} = \frac{\Im(f(x_0 + ih))}{h} + O(h^2). \tag{16}$$

Different from the equation (9), the lack of the subtraction term is the reason of high accuracy of the complex step method as no rounding error was introduced. Since the introduction of a imaginary part into the approximation gave a rounding error free, it was hoped that the same method can be expanded to give a numerical expression for the second order derivative. This time, due to $i^2 = -1$, the coefficient of $f''(x_0)$ is not imaginary. Hence, the real part on both side of equation (14) is

$$\Re(f(x_0 + ih)) = f(x_0) - h^2\frac{f''(x_0)}{2!} + h.o.t. \tag{17}$$

Rearrange the expression,

$$f''(x_0) = \frac{2!(f(x_0) - \Re f(x_0 + ih))}{h^2} + O(h^4). \tag{18}$$

Equation (18) shows that a rounding error was introduced when complex step method is used to approximate higher order of derivative, which means the rounding error can only be eliminated when the coefficient is imaginary so the imaginary parts introduced into the expression needed to be extended into higher order. The idea of applying the complex step method recursively was investigated based on equation (16) [1],

$$f''(x_0) \approx \frac{\Im[\Im(f(x_0 + 2ih))]}{h^2}. \tag{19}$$

However, $\Im[\Im(f(x_0 + 2ih))] = 0$, which means a recursive complex step method is not feasible. This is when multicomplex step method became useful.

4.3 Multicomplex Step Method Derivation

The multicomplex step method also started with equation (8) and it is the further extension of equation (9). It starts with the Taylor expansion about a real point x_0 and

imaginary parts are added up to the n^{th} order,

$$f(x_0 + i_1h + i_2h + ... + i_nh) = f(x_0) + (i_1 + ... + i_n)hf'(x_0) + (i_1 + ... + i_n)^2h^2\frac{f''(x_0)}{2!} + ... + (i_1 + ... + i_n)^nh^n\frac{f^n(x_0)}{n!} + O(h^{n+1}) \quad (20)$$

From the multinomial theorem,

$$(i_1 + ... + i_n)^k = \sum_{\substack{k_1,...,k_n \\ k_1+...+k_n=k}} \frac{n!}{k_1!...k_n!} {i_1}^{k_1}{i_2}^{k_2}...{i_n}^{k_n} \quad (21)$$

The right hand side of equation (21), the product term of ${i_1}^{k_1}{i_2}^{k_2}...{i_n}^{k_n}$ can only be achieved when $k_1 = 1$, $k_2 = 1...$ $k_n = 1$. This is because if k_p, where $p = 1, 2...n$, is even, $i_p = -1$ and the corresponding i term disappear. Similarly, if k_p is any odd number greater than 1, say 3, then $i_p^3 = -i_p$.

From equation (20), the coefficient of the n^{th} order of derivative is $(i_1 + ... + i_n)^n$. Then the equation of the n^{th} order approximation can be find by taking the imaginary term $(i_1 + ... + i_n)^n$ from both side,

$$\Im_{1...n}[f(x_0 + i_1h + ... + i_nh)] = h^n\frac{f^n(x_0)}{n!} + O(h^{n+1}). \quad (22)$$

Rearrange the expression,

$$f^n(x_0) = \frac{\Im_{1...n}[f(x_0 + i_1h + ... + i_nh)]}{h^n} + O(h^2). \quad (23)$$

Finally the multicomplex step method that could be used to look for n^{th} order of derivative can be expressed as

$$\frac{d^nf}{dx^n} \approx \frac{\Im_{1...n}[f(x_0 + i_1h + ... + i_nh)]}{h^n}. \quad (24)$$

From equation (24), $f(x) = f(x_0 + i_1h + i_2h.... + i_nh)$ where h is the step size and is also treated as a small perturbation. If there are more than one variable in the function, the multicomplex step can also be used in the following ways [7].

$$\frac{\partial f^2(x,y)}{\partial x^2} \approx \frac{\Im_{12}[f(x + i_1h + i_2h, y)]}{h^2} \quad (25)$$

$$\frac{\partial f^2(x,y)}{\partial x \partial y} \approx \frac{\Im_{12}[f(x + i_1h, y + i_1h)]}{h^2} \quad (26)$$

$$\frac{\partial f^(x,y)}{\partial x} \approx \frac{\Im_1[f(x + i_1h + i_2h, y)]]}{h} = \frac{\Im_2[f(x + i_1h + i_2h, y)]]}{h} \quad (27)$$

The same theory in equations (25) to (27) works for other multivariables function with mores states and higher derivatives for each of the state.

5 Multicomplex Class

In this section, the capability and operation of the class definition, **multicomplex**, is listed and tabulated in detail. First of all, the input taken by this class is multicomplex number in its expanded form. A typical multicomplex number $z_n \in \mathbb{C}_n$ can be expressed in the from of

$$z_n = z_{n-1,1} + i_n z_{n-1,2}. \tag{28}$$

Note that in equation (28), the expression is recursive until $z_{n-1,1}, z_{n-1,2} \in \mathbb{C}_0$. Take $z_3 \in \mathbb{C}_3$ as an example, as a multicomplex number to be input into the class, it is in the form of

$$z_3 = a + i_1 b + i_2 c + i_1 i_2 d + i_3 e + i_1 i_3 f + i_2 i_3 g + i_1 i_2 i_3 h, \tag{29}$$

where a is the real number input and the rest of the terms are the coefficients of the imaginary parts. The coefficients for the terms i_1, i_2 and i_3 were the small step added to the real number from different multicomplex domain. For instance, with an real number input x, which $x = 0.5$ with the step size h, which $h = 1 \times 10^{-9}$. the function **inputconvertor** helped user to convert the input in terms of the format which acceptable for the class. A visual representation of the input that would be displayed in the array form in MATLAB is tabulated below.

MATLAB array index	1	2	3	4	5	6	7	8
Represented parts	real	i_1	i_2	$i_1 i_2$	i_3	$i_1 i_3$	$i_2 i_3$	$i_1 i_2 i_3$
Equation (29)	a	b	c	d	e	f	g	h
Input value example	0.5	10^{-9}	10^{-9}	0	10^{-9}	0	0	0

Table 1: Array form of multicomplex input in MATLAB

The **multicomplex classdef** was capable of various different mathematical operations and functions. Table 2 tabulates the mathematical operations available within the class definition and their implementation with required steps.

Similarly, equation (30) to equation (41) were the available mathematical functions within MATLAB and their adaptation of different mathematical manipulations in order to be used in the class as a part of multicomplex step method.

The trigonometry functions are available in the class. To be able to use the MATLAB built0in function, which only take input in the $\mathbb{C}_0$ space, it is necessary to decomposed their constituted function into a lower multicomplex number order. The trigonometry decomposition are as following.

$$sin(z_n) = sin(z_{n-1,1})cosh(z_{n-1,2}) + i_n cos(z_{n-1,1})sinh(z_{n-1,2}), \tag{30}$$

$$cos(z_n) = sin(z_{n-1,1})cosh(z_{n-1,2}) + i_n sin(z_{n-1,1})sinh(z_{n-1,2}), \tag{31}$$

$$tan(z_n) = \frac{sin(z_n)}{cos(z_n)}, \tag{32}$$

Hyperbolic functions also operated in the same way in terms of breaking itself down into functions that takes in a lower order as input.

$$sinh(z_n) = sinh(z_{n-1,1})cos(z_{n-1,2}) + i_n cosh(z_{n-1,1})sin(z_{n-1,2}), \tag{33}$$

Operation	Implementation
Addition (**plus**)	1. Convert both z_1 and z_2 into the same length by **consimulti** 2. Element-wise addition of the array
Subtraction (**minus**)	1. Convert both z_1 and z_2 into the same length by **consimulti** Element-wise subtraction of the array
Multiplication (**mtimes**)	1. Transform array into matrix by **matrep** 2. Perform matrices multiplication 3. Convert the matrix back to array by **arrayM**
Division (**mrdivide**)	1. Transform array into matrix by **matrep** 2.invert the divisor matrix 3.perform matrices multiplication 4. Convert the matrix back to array by **arrayM**
Integer Power (**mpower**)	1. Transform array into matrix by **matrep** 2. Raise the matrix to the integer power 3. Convert the matrix back to array by **arrayM**
Fractional Power (**mpower**)	1. Use de Moivre's theorem, $z^p = r^p cosp\theta + ir^p sinp\theta$ 2. find r as $\|z\|$ by **modc** 3. find θ as $arg(z)$ by **atan2**

Table 2: Methods used for the operations in the MATLAB class definition

$$cosh(z_n) = cosh(z_{n-1,1})cos(z_{n-1,2}) + i_n sinh(z_{n-1,1})sin(z_{n-1,2}). \tag{34}$$

The inverse trig functions are also available as

$$asin(z_n) = -i_n log(i_n z_n + \sqrt{|1 - z_n^2|}e^{\frac{in}{2}arg(1-z^2)} \tag{35}$$

$$acos(z_n) = -i_n log(z_n + i_n\sqrt{|1 - z_n^2|}e^{\frac{in}{2}arg(1-z^2)}. \tag{36}$$

$$atan(z_n) = \frac{1}{2}log(\frac{i_n + z_n}{i_n - z_n}) \tag{37}$$

The logarithmic formula

$$log(z_n) = log|z_n| + i_n arg(z_n), \tag{38}$$

where the modulus is

$$|z_n| = \sqrt{z_{n-1,1}^2 + z_{n-1,2}^2} \tag{39}$$

and the argument is

$$arg(z_n) = atan2(z_{n-1,2}, z_{n-1,1}) = 2atan(\frac{\sqrt{z_{n-1,1}^2 + z_{n-1,2}^2} - z_{n-1,1}}{z_{n-1,2}}). \tag{40}$$

Lastly, the exponential formula

$$e^{z_n} = e^{z_{n-1,1}}cos(z_{n-1,2}) + i_n e^{z_{n-1,1}}sin(z_{n-1,2}) \tag{41}$$

is also a part of the class definition.

Note that the calculation involved $z_n \in \mathbb{C}_n$ is a recursive approach until $n = 0$ where z_n is the input in the format displayed in equation (28).

As listed in table 2, fractional power is dependent on many other functions. All the functions within this class is also interdependent on each other. If one of the functions introduces an error in any form, this would results error to be introduced into other functions, hence, the final result. Apart from the function listed in equation (30) to (41), there were some other utility functions, a few of them were mentioned in table 3.

Utility Function	Implementation
matrep	Transform multicomplex number array into its matrix form by using principle listed in section 3.2
arrayM	Transform multicomplex number matrix into its array form by extracting the first column
imgn	Extract the coefficient of the term $i_1...i_n$
CX2	Extract any user defined coefficient of the term i_n
consimulti	convert both multicomplex number array into the same length for operations

Table 3: Utility Functions in the MATLAB class definition

After the user input the multicomplex number alongside with the functions and operators. From equation (23),the last imagery term is taken out by **imgn** then to be divided by the step size h raised to the power of the order of derivative n.

6 Class Optimisation

The existing multicomplex class can achieve many operations and can compute different functions, however, it still has rooms for improvement. The following are the main areas which this project mainly focused on optimising; First of all, the computational of fractional power, which is a part of **mpower** within the **multicomplex** class, computed inaccurate results with the increase in the order of derivatives. Secondly, the lack of complex power within the class restrained its future possibilities. Thirdly, there was a maximum order of multicomplex number the class could handle. In this section, the causes of the above problems are discussed and some potential solutions are suggested. In the end, each problem is given an appropriate solution alongside comparisons to the pre-optimised state.

6.1 Fractional Power operation

The fractional power function could only provide accuracy approximation when the multicomplex number is in the $\mathbb{C}_3$ or lower. Even after the author's improvement, the scaling error still appears with the increase in the order n of multicomplex number system $\mathbb{C}_n$, or the x value in the testing function $f(x)$.

6.1.1 Existing Problems and Inefficiencies

As mentioned in the section 3.2, complex numbers and their matrix form are interchangeable. Hence, the transformation of a multicomplex number into a matrix was widely used in many functions and operations, such as **mpower** in the class. However, unlike the

calculation of integer power, the calculation of fractional power cannot use the MATLAB built-in **mpower** function to get an accurate approximation.

When **mpower** processes matrix with small entries, which are the step size added, the error raises. This is due to the reason that the result generated from **mpower**, when the power is a fractional number, depends on the distribution of the eigenvalues of the matrix as mentioned on MATLAB menu [11]. This condition agrees with the previous suggestion, which MATLAB employs with the Blocked Schur Algorithms [12]. Blocked Schur Algorithms uses the idea of Schur factorization $A = QSQ^H$ [13]. The corresponding Schur factorization algorithm makes use of the eigenvalues of a matrix, which is greatly affected by the small entries contributed by the small step size h. That explained why using the built-in **mpower** in MATLAB for fractional power computation could lead to an unstable result with an error. Hence other computational methods for fractional power were implemented in [1].

The **multicomplex** class in MATLAB used de Moivre's theorem for the calculation of fractional power. The de Moivre's theorem used is in the form of

$$z_n^p = r^p cosp\theta + i_n r^p sinp\theta, \tag{42}$$

where r is the modulus and θ is the argument of the multicomplex number z_n. This is a recursive calculation until the right-hand side of equation (42) reaches the $\mathbb{C}_0$ space. Modulus can be expressed as

$$r = |z| = \sqrt{x^2 + y^2}, \tag{43}$$

where $|z|$ can be calculated from equation (39) and argument θ can be obtained using equation (40). Equation (43) shows that there is an addition term when determining r and equation (40) shows that both of the addition and subtraction terms are involved when determining θ. These subtraction and addition terms introduce rounding error into the final approximation. The higher the order of derivatives the class needs to determine, the larger the rounding error is added up in the recursive process, the further away the approximation is from the real result.

However, the initial method implemented was only valid up to the multicomplex number in $\mathbb{C}_2$ space due to the error introduced to the **log** when the imaginary parts become too small. This error in **log** increases when $i_n \pm z_n \approx i_n$ in equation (37), where $z_n < 10^{-16}$. It was then improved by using a Taylor's expansion around complex $atan(z_n)$ and changed the underlying computational method from equation (37) to

$$atan(z_n) = atan(z_{n-1,1}) + i_n \frac{z_{n-1,2}}{1 + (z_{n-1,1})^2} \tag{44}$$

when $z_n < 10^{-16}$. This method allowed the result to be error-free up to the third to forth order of derivative depending on the value of the real number input. After a range of testing with different values as input, it can be confirmed that the existing class was able to give an accurate result with the input real value greater than 2 as the derivative increased to the sixth order. With the increase in the input value and the order of derivatives, the relative error, between the estimated result from this numerical method and the actual result from the analytical method, increased.

The reason that the Taylor's expansion round the complex $atan(z_n)$ did not fully solve the problem was because it only focused on the rounding error caused by the subtraction and

addition terms. Refer back to the two main equations used for this calculation, equation (43) and equation (40). Since matrices are used for power operation, an inverted matrix would act as the divisor in (40). There is also a square root in (43), which leads to the use of **atan2**, a function involves division operation as well. This indicated that the hike in the error was caused by division operation. To visualise the problem with an inverted matrix in the division operation, take the $z_2 \in \mathbb{C}_2$ for an example, with a large real value, say 8, and step size of 10^{-9},

$$z_2 = 8 + 10^{-9}i_1 + 10^{-9}i_2 = \begin{bmatrix} 8 & 10^{-9} & 10^{-9} & 0 \\ 10^{-9} & 8 & 0 & 10^{-9} \\ 10^{-9} & 0 & 8 & 10^{-9} \\ 0 & 10^{-9} & 10^{-9} & 8 \end{bmatrix}. \tag{45}$$

When the size of the matrix grows with the increase in the order of derivatives, or when the real value, which is the diagonal value, increases, the whole matrix tends to a diagonal matrix. Diagonal matrices are singular and cannot be inverted, which explained the scaling error increased with the increase in the order or in the real value. However, this ensured that the function's accuracy for small real value and low order.

6.1.2 Revised Implementation Methods

To summarise the problems of fractional power in this class, it had rounding errors and the matrix in division operation could not be inverted when its size was large enough to cause the matrix tends to singular. The rounding error problem was caused by **log**, which was used to calculate the modulus of z_n. The singularity problem raised within **atan2**, which was needed when calculating the argument of z_n. These two calculations are unavoidable if de Moivre's theorem is used. Hence, de Moivre's theorem should be avoided and a new method should be implemented.

First revised method was common method used in many other numerical codes is the implementation of exponentiation using

$$z^p = e^{pLog(z)} \tag{46}$$

and it avoids the use of de Moivre's theorem. Despite that, $Log(z)$ means that it takes the principle value of the logarithm for complex values, so

$$Log(z) = In(r) + iarg(z). \tag{47}$$

This circle back to the use of **log** and **atan2**, that caused the initial problem.

Besides, due to the flexibility of transformation between the multicomplex number and its matrix form and MATLAB's fast operation on matrix multiplication as mentioned in section 3.2, the use of matrix for **mpower** should be preserved.

Since the matrix approach was retained, ways of calculating the fractional power of a matrix were explored. The second revised method was a direct numerical procedure to obtain the result of matrix with power in the form of

$$A^p = PD^pP^{-1}, \tag{48}$$

where A is the matrix, p is the fractional power, D is the diagonal matrix of eigenvalues of A and P is the corresponding eigenvector matrix. However, this led to the same problem

as method of directly using MATLAB built-in **mpower**, which also involved the use of eigenvalues, so this is not a feasible method.

Since all these direct approaches circled back to the original problem the class was facing, other indirect method should be explored.

6.1.2.1 Final Implementation - Binomial Expansion of Matrix with Fractional Power

The binomial expansion method [14] of a matrix with a fractional power was originally widely used for the application such as economic statistic. Even though with different purposes, the formulation of the binomial expansion was used because of the same mathematical foundation. It is in the form of

$$A^p = [I + B]^p = I + pB + \frac{p(p-1)}{2!}B^2 + \frac{p(p-1)(p-2)}{3!}B^3 + ..., \tag{49}$$

where I is the identity matrix. The norm of matrix B is computed as

$$N(B) = \sqrt{\sum_i \sum_j b_{ij}^2}, \tag{50}$$

where b are the elements in B. If $N(B) \geq 1$, equation (49) may not converge and it only converges slowly if $N(B)$ is close to 1. To avoid this problem, equation (49) was altered by setting $A = k[I + C]$ and

$$A^p = k^p \left[I + pC + \frac{p(p-1)}{2!}C^2 + \frac{p(p-1)(p-2)}{3!}C^3 + ... \right]. \tag{51}$$

In this case, the norm of matrix C is

$$N(C) = \sqrt{\left(\frac{1}{2}\right)^2 \sum_i \sum_j a_{ij}^2 - \frac{2}{k}\sum_i a_{ii} + n} \tag{52}$$

and it can be minimised by setting k as

$$k = \sum_i \sum_j a_{ij}^2 / \sum_i a_{ii}. \tag{53}$$

A new function **fractionalpow** was built by using the method shown in equation (51) and (53). To ensure the accuracy of binomial approximation, 50 terms, excluding the identity matrix I term, of the expansion in equation (51) were used. This new function allowed the class to compute derivatives all the way up to 13^{th} order, which is the limit of this class. This will be explained in the later session in a more detailed manner.

Figure 2 and 3 are the comparison plots of percentage error of the new function, using **fractionalpow** and pre-optimised fractional power function, using the de Moivre's theorem, in the third order calculation. Values of the error generated by the old function were taken directly from 2018 Multicomplex number Master thesis [1]. $f = \sqrt{e^{2x} + x + e^x}$ and $f = \sqrt{sin(x) + x^2/cos(x)}$ were used also because these are the same functions used in [1]. In Figure 3, there are some discontinuities in plots, which indicates the estimated value from the new function matched exactly with the exact value calculated from the

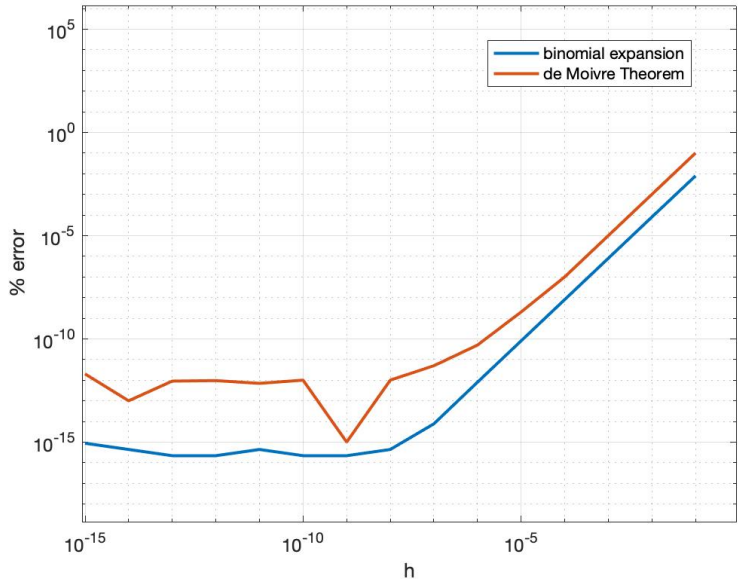


Figure 2: Percentage error of the new function for third derivative calculation for $f = \sqrt{e^{2x} + x + e^x}$ at $x = 0.5$

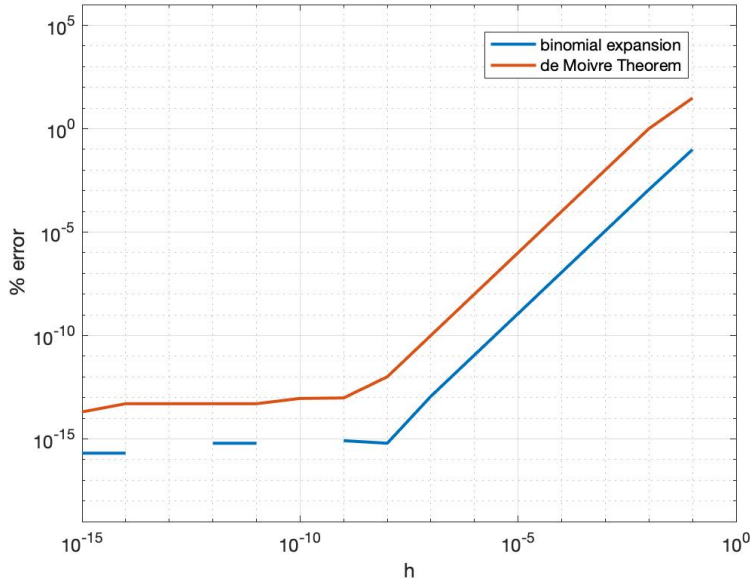


Figure 3: Percentage error of the new functionfor third derivative calculation for $f = \sqrt{sin(x) + x^2/cos(x)}$ at $x = 0.5$

analytical method, hence, zero error and cannot be plotted into this graph. This does not technically means that the estimated values are completely the same as the exact value because the default precision of MATLAB is 16 digits [16]. However, it means that the percentage error is definitely lower than 1×10^{-16}. A more detailed explanation will be included in section 7.1. To compare the old and new function, the binomial expansion gives a more accurate result by at least one or two orders in magnitudes in both cases.

6.2 Complex number power

Multicomplex number with multicomplex number power in [1] was implement by using

$$z_n^{z_m} = e^{z_m \ln(z_n)} \tag{54}$$

where z_n and z_m are both multicomplex numbers. However, this equation is not valid for a complex number power, even when the complex number is being input to the class in the multicomplex number form. Equation (46) returns a real valued result instead of a complex number result for a complex number power calculation. This is because in equation (54), the imaginary part of the complex power is treated as a small step added instead of a actual component of the power. However, for an actual complex number power, its calculation involved an actual imaginary part hence the answer would be a complex number as well. Hence, other methods should be implemented in order to solve this limitation.

6.2.1 Revised Implementation Methods

A few methods were tried to achieve the complex power operation. The first two attempts were to use

$$z_n^c = e^{c \ln(z_n)} \tag{55}$$

where z_n is a multicomplex number and $c = a + ib$ and transform z_n into a matrix, then used equation (55). However, the error is large. This is because from equation (55), the **exp** function in **multicomplex** class always converts the complex number result back to a multicomplex number form. Hence in the later stage of calculation, the imaginary part was again, treated as a step size in i_1 and ignored in the final solution by using the normal procedure which makes use of the $i_1 ... i_n$ term. Using the same equation (55) but with the matrix form of z_n was the same concept as above so it would not give an accurate answer as well.

The third attempt was to use Taylor expansion of exponential,

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots \tag{56}$$

and the log expansion,

$$\ln x = (x-1) - \frac{1}{2}(x-1)^2 + \frac{1}{3}(x-1)^3 - \frac{1}{4}(x-1)^4 + \dots \tag{57}$$

and this did not give any accurate results. All of these attempts failed because they all only focused on the expansion of equation (55) by changing its format or by considering at its approximation approaches. Hence, the mechanism of all these approaches are fundamentally the same and lead to use the same functions in the class. Instead of seeking for an appropriate method by changing the simple format of equation (55), another method, that could change the underlying function used, was more related to solving the complex power problem was discovered and tested.

6.2.2 Final Implementation - de Moivre's theorem expansion on the complex power equation

The successful attempt, interestingly, was a method that makes use of the de Moivre's theorem, which caused problems in the fractional power calculation. This method started

from equation (55), then it further broke the exponential and logarithm terms down by de Moivre's theorem [15] to,

$$e^{ci} = cos(c) + isin(c). \tag{58}$$

Hence,

$$e^{a+bi} = e^a \cdot e^{bi} = e^a(cos(b + isin(b)). \tag{59}$$

On the other hand, Equation (54) can be expressed as

$$z^{a+bi} = e^{(a+bi)\ln(z)} = e^{a\ln(z)} \cdot e^{bi\ln(z)} \tag{60}$$

By substituting equation (59) into Equation (60), the final expression,

$$e^{a\ln(z)} \cdot e^{bi\ln(z)} = e^{a\ln(z)}(cos(b\ln(z)) + isin(b\ln(z)), \tag{61}$$

is obtained. The function built based on equation (61) only worked up to 8^{th} order of derivative due to scaling error reappearing with large matrices in the calculation. This is due to the the use of **log** for $\ln(z)$ involved a square root in the calculation and that leads to the use of fractional power function within the class. Hence, the same type of error arise from the old fractional power function occur in this situation as well. Originally, each function was developed independently. However, this error indicated that **fractionalpow** should be a part of the **mpower** in the multicomplex class instead of a independent function due to the interdependence among functions. After implementing the fraction power function alongside with complex power function, scaling error disappeared and the problem was successfully solved. Complex power was then added as a part of **mpower** within the class.

It was later realised that the formula for complex power function can be further simplified into

$$z^a(cos(b\ln(z)) + isin(b\ln(z)). \tag{62}$$

This is a more direct approach than the exponential approach. The direct approach uses **mpower** for its exponential term and the exponential method involves the use of **log** and **exp**, which is redundant. However, the method described in (61) was already fully implemented and used in testing for the performance. Hence, a comparison was made to check if using direct method in equation (62) and the method in (61) would cause a noticeable difference in terms of time efficiency and accuracy. Figure 4 illustrates the percentage error of both of the real part and the imaginary part of the result. It is plotted in this way because MATLAB ignores the imaginary part in plotting. However, the accuracy of the imaginary part is equally important in this task. From figure 4 we can see that the direct method gives a greater accuracy by an average magnitude of one to two degree. However both methods give error that is sufficiently low hence the results given by the the exponential method in this report are acceptable. Figure 5 gives the computation time needed for both methods used in a range of different order of derivatives calculation. It is not surprising that the direct method needed less computational time but it is not a significant difference between the two methods. It is safe to say that the direct method did not cause a difference huge enough to over throw the conclusion made in the later stage, but this method is the final method implemented in the class for future use.

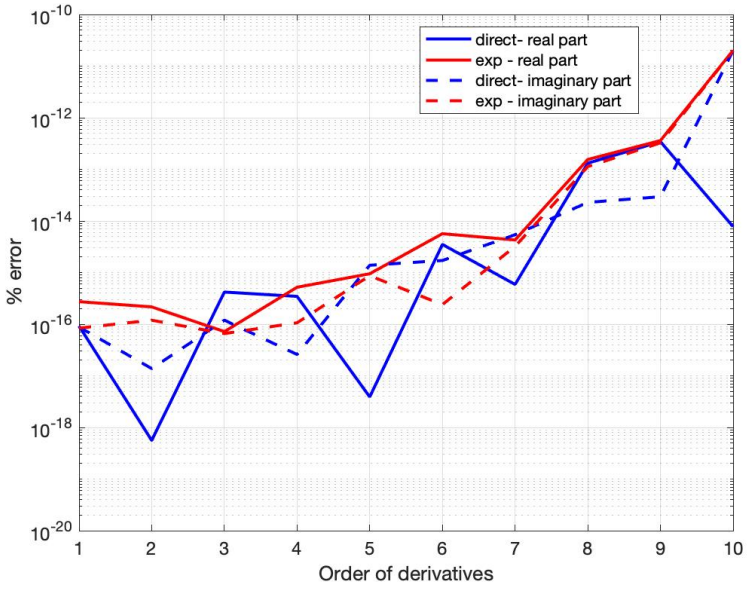


Figure 4: Percentage error of the estimated result of each approach in different order of derivatives for $f = (e^{2x} + x + e^x)^{2+2i}$ at $x = 0.5$.

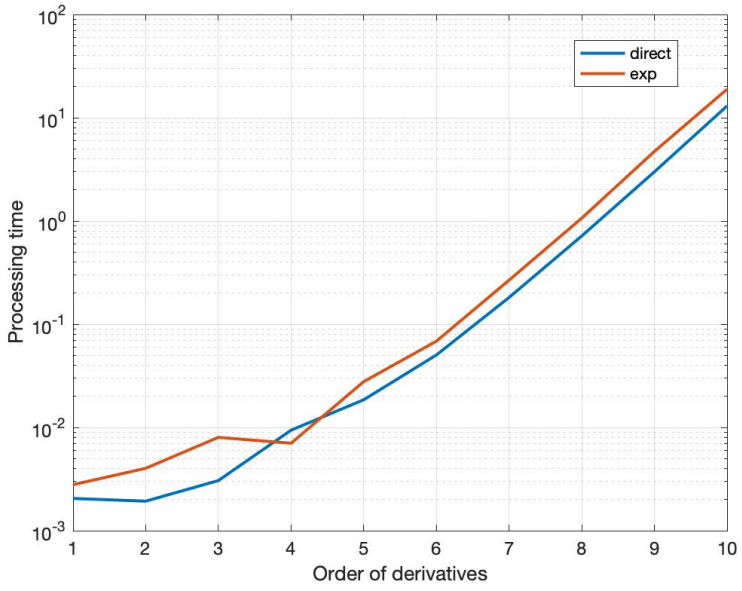


Figure 5: Processing time of each approach in different order of derivatives for $f = (e^{2x} + x + e^x)^{2+2i}$ at $x = 0.5$.

6.3 Matrix Operation

This **multicomplex** class definition had a ceiling of the order of derivatives it can perform, which was also the ceiling of the order of multicomplex number system the class can take. This ceiling was caused by the large matrices in the calculation. Session 3.2 demonstrated the feature of the multicomplex number that it can interchange freely between the number form and matrix form. Due to this flexibility and the widely use of matrices in various calculation, majority of the operations and function in the class requires multicomplex number to transform into matrix before carry out the calculation,

for instance, **mpower**, **log** and **atan2**. However, the matrix size increases exponentially with the increase in the order of multicomplex number system, which increases with the order of derivatives one wishes to calculate. For example, a multicomplex number in $\mathbb{C}_2$ space has a length of $2^2 = 4$ in MATLAB, then its matrix form would have the size of 4×4. When the order of multicomplex number is increased into $\mathbb{C}_{10}$, the length of $z_{10} \in \mathbb{C}_{10}$ is $2^{10} = 1024$ and its matrix would be 1024×1024 in size, After testing, the maximum size of matrix the class can compute is $2^{13} \times 2^{13}$, hence the 13^{th} order of a multicomplex number is the limit the class can process. This is because of the memories in MATLAB can be allocated to matrices is limited [17].

It is important to look for ways that can deal with the size of a matrix, which is the fundamental cause of the limit in the order of multicomplex number. To solve the problem of having an upper limit in the calculation, the nature of a multicomplex number matrix was examined. A feature of a multicomplex number matrix allows the memories used by these huge matrices to be reduced by almost half. According to equation (6), the transpose of the first column of a multicomplex number matrix, where A is the matrix and $j = 1, 2, 3, 4$, is

$$A'_{j1} = \begin{bmatrix} a_{11} \\ a_{21} \\ a_{31} \\ a_{41} \end{bmatrix}' = \begin{bmatrix} a_{11} & a_{21} & a_{31} & a_{41} \end{bmatrix}. \tag{63}$$

It is the multicomplex matrix A in its multicomplex number form, $z_2 \in \mathbb{C}_2 = a_{11} + i_1 a_{21} + i_2 a_{31} + i_1 i_2 a_{41}$. To use this concept in a matrix multiplication of

$$A \times B = C \tag{64}$$

where A, B, C are all matrices, it is equivalent to

$$A \times B_{j1} = \begin{bmatrix} a_{11} & -a_{21} & -a_{31} & a_{41} \\ a_{21} & a_{11} & -a_{41} & -a_{31} \\ a_{31} & -a_{41} & a_{11} & -a_{21} \\ a_{41} & a_{31} & a_{21} & a_{11} \end{bmatrix} \begin{bmatrix} b_{11} \\ b_{21} \\ b_{31} \\ b_{41} \end{bmatrix} = \begin{bmatrix} c_{11} \\ c_{21} \\ c_{31} \\ c_{41} \end{bmatrix} = [z_2]' \in \mathbb{C}_2. \tag{65}$$

Hence, for multiplication, the multiplier does not have to be in matrix form in order to carry out the operation. However, even though utilising this advantage can reduce the memories used for matrices storage, it still did not overcome the fact that the maximum n MATLAB can take in $2^n \times 2^n$ is 13 because a multicomplex matrix still exists in this calcualtion.

6.3.1 Matrix-less Operation

A effective way to overcome this ceiling of having a maximum order of multicomplex number is to eliminate the use of matrix A of equation (65) as well as B matrix while perverse the operation of the matrix, which is required for in the computation process of many functions in the class. It is already known that matrix B can be simply eliminated by changing it to its vector form. However, the elimination of matrix A requires more work. A dot product for each row in A matrix with the B_{j1} column vector can preserve the mechanism of multicomplex number matrix multiplication operation without the actual presence of matrices, Hence, each individual element in each row needs to be determined in order to carry out the dot product calculation. It is worth noticing that all the

elements in the multicomplex number matrix are the same repeated elements from the multicomplex number itself. The pattern and sign of these element followed the nature described in section 3.2 so the each row of the matrix A in the form of equation (6) can be found by understanding the arrangement of the matrix.

Two functions were developed to extract each row from the matrices to perform the operation. The first function is based on the element swapping pattern in each row. The second function is based on the recursive nature of the arrangement of the transformation from multicomplex number to the matrix form introduced in section 3.2.

6.3.1.1 Matrix-less operation by swapping the elements in each row

1	2	3	4	5	6	7	8
2	1	4	3	6	5	8	7
3	4	1	2	7	8	5	6
4	3	2	1	8	7	6	5
5	6	7	8	1	2	3	4
6	5	8	7	2	1	4	3
7	8	5	6	3	4	1	2
8	7	6	5	4	3	2	1

Figure 6: the index of a $z_3 \in \mathbb{C}_3$ element array in matrix form

Figure 6 illustrates the indexes of a $z_3 \in \mathbb{C}_3$ multicomplex number element in the matrix form. The element indexes are the same as described in table 1. From this figure, it is obviously that the elements in the matrix are basically rows of repeated element of a multicomplex number in different sequence arrangement.

The function **matmultiple** makes use of the pattern of each row by swapping the element in a specific order. The colour coded sub-matrix in figure 6 each represent a complex number, which the larger number in the square represents the imaginary part i_1. With the help of the colour coded sub-matrices, the swapping of element in each row can be summarised in table 4.

Row number	1	2	3	4	5	6	7	8
Swapping	0	1	2	2→1	4	4→1	4→2	4→2→1

Table 4: The swapping pattern of the multicomplex number with

Table 4 uses the first row in the matrix as the reference row because its element arrangement is the exact order of an multicomplex number. The number in 'swapping' means that how many elements are swapped in arrangement each time. For example in column 8, it means that the 8^{th} row uses the 1^{st} row as the reference, swap every 4 elements first, then swap every two elements based on that, following by swapping every one of the elements based on the previous rearrangement. This pattern will continue for a larger matrix and the MATLAB code of this function is attached in the Appendix.

This function introduced a tricky challenge which was to determine the sign of each element. The arrangement of element of each row can be determined by the swapping element pattern but the sign can only be determine by a recursive process, which is demonstrated in equation (6). Depending on which complex space the multicomplex number is in, each element in different location of the matrix has its designated sign.

Hence, determining the location of elements in the matrix can determine the sign. The same way can also be used to determine the element itself so the swapping is made redundant.

6.3.1.2 Matrix-less operation by recursive loops

Another method was developed by using the the recursive nature in the pattern of multicomplex number as described in equation (7). Only the elements in the matrix upper left quarter needs their signs to be changed each time. The mechanism of the recursive process of determining the element index and the appropriate sign is demonstrated as below. For example, if 8^{th} element in the third row for $z_3 \in \mathbb{C}_3$ is the element one wishes to determine, first step is to determined its location within the matrix. As this element is in the upper left quarter in the matrix, which is highlighted in bold, the sign is changed to negative.

$$A_{ij} = \left[\begin{array}{cccc|cccc} 1 & 2 & 3 & 4 & \mathbf{5} & \mathbf{6} & \mathbf{7} & \mathbf{8} \\ 2 & 1 & 4 & 3 & \mathbf{6} & \mathbf{5} & \mathbf{8} & \mathbf{7} \\ 3 & 4 & 1 & 2 & \mathbf{7} & \mathbf{8} & \mathbf{5} & \mathbf{6} \\ 4 & 3 & 2 & 1 & \mathbf{8} & \mathbf{7} & \mathbf{6} & \mathbf{5} \\ \hline 5 & 6 & 7 & 8 & 1 & 2 & 3 & 4 \\ 6 & 5 & 8 & 7 & 2 & 1 & 4 & 2 \\ 7 & 8 & 5 & 6 & 3 & 4 & 1 & 2 \\ 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \end{array}\right], \tag{66}$$

where $i = 1, 2, 3, 4, 5, 6, 7, 8$ and $j = 1, 2, 3, 4, 5, 6, 7, 8$. After that, take the corresponding quarter out as a sub-matrix. This time the same element is located in the lower right quarter in this sub-matrix, which is also highlighted in bold. Hence in this round of searching, the sign is unchanged and remains negative.

$$A_{kl} = \left[\begin{array}{cc|cc} 5 & 6 & 7 & 8 \\ 6 & 5 & 8 & 7 \\ \hline 7 & 8 & \mathbf{5} & \mathbf{6} \\ 8 & 7 & \mathbf{6} & \mathbf{5} \end{array}\right], \tag{67}$$

where $k = 1, 2, 3, 4$ and $j = 5, 6, 7, 8$. Lastly, the same process was repeated again. The highlighted sub-matrix was extracted.This time the location of the corresponding element is again located in the upper right quarter. Hence, the sign is changed again and back to positive.

$$A_{mn} = \left[\begin{array}{c|c} 5 & \mathbf{6} \\ \hline 6 & 5 \end{array}\right], \tag{68}$$

where $k = 3, 4$ and $j = 7, 8$. Matrix equation (66) to (68) show the process of how this function works. The sign in the whole process changed twice when the location of the element landed on the upper right hand corner. That means the sign is positive at the end. Equation (68) established that the 8^{th} element in the 3^{rd} row for $z_3 \in \mathbb{C}_3$ is the 6^{th} element of $z_3 \in \mathbb{C}_3$, which is the coefficient of $i_1 i_3$. The rest of the elements in the matrix can also be determined in the same way. Regardless of how big the matrix is with the increase in $\mathbb{C}_n$, this recursive process will always be repeated until there is only one element left in the quarter. This method is implemented to **arr4matmulti**, which is used to carry out matrix-less multiplication operation, and **arr4mat**, which is used to carry out matrix-less power operation.

However, due to the nature of this process, which involves constantly checking which quarter the element belongs and extracting corresponding quarter as a sub-matrix for the next round of checking, the computational process are longer. To investigate if this tedious process would have an effect on the computational time, $f = \sqrt{e^{2x} + x + e^x}$ is used for testing. Another fractional power function, **fracpow**, is developed in order to take **arr4mat** as the matrix-less power operation. The results is shown as below.

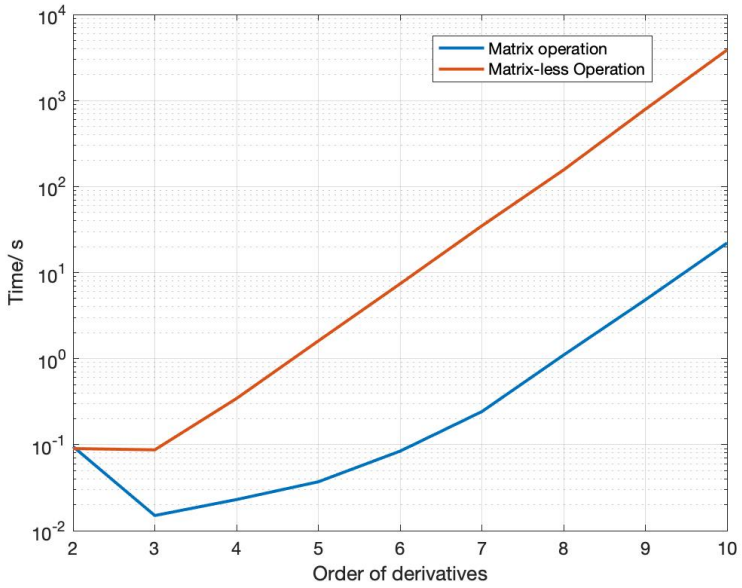


Figure 7: Computational time comparison between matrix and matrix-less operation for $f = \sqrt{e^{2x} + x + e^x}$ at $x = 0.5$

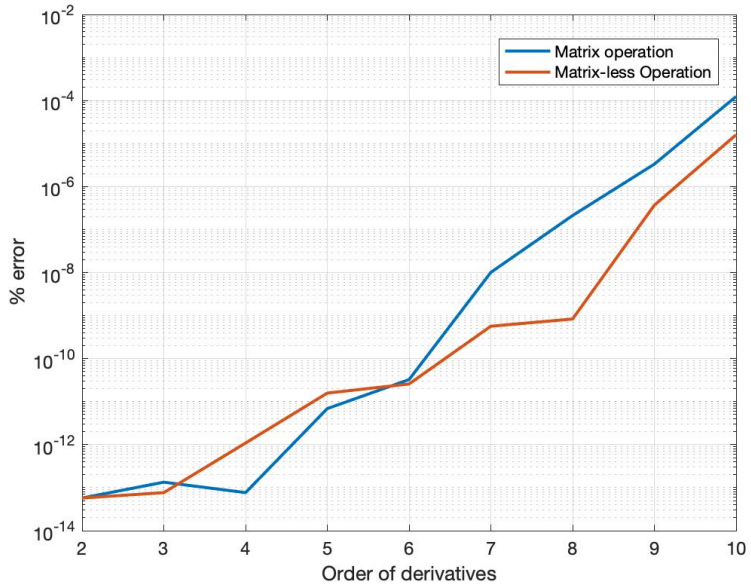


Figure 8: Percentage error comparison between matrix and matrix-less operation for $f = \sqrt{e^{2x} + x + e^x}$ at $x = 0.5$

Figure 7 illustrates the computational time of both of the matrix operation and matrix-less operation. It is obvious that the matrix operation is significantly faster when com-

paring to the matrix-less operation. On the other hand, as shown in figure 8, matrix-less operation gives more accurate results by at least a order in magnitude as the order of derivative reaches to 6^{th} order and above. With this phenomenon being the result, even though the original aim of allowing the class to operate above 13^{th} order was achieved, the computational time became another big problem. Hence, other more elegant solutions were then examined in order to reduce the computation time of matrix-less operation.

6.3.2 Mex Function

One way to reduce the processing time while maintaining all the necessary steps in the matrix to look for faster compiler. Comparing to MATLAB, C++ gives a faster processing speed [18]. The use of MEX, which stands for 'MATALB executable", allows the code to call C++ programme within the MATLAB script by creating an interface between them. First of all, a C++ compiler was needed so Xcode was installed. Then, C++ compiler was set up in MATLAB so the MEX file created can be recognised. The setup is illustrated in the figure below.

```
>> mex -setup C++
MEX configured to use 'Xcode Clang++' for C++ language compilation.
```

Any function can be transformed into a MEX function by the use of MATLAB coder. MATLAB coder allows the user to specify the data types of each input variables within the function, which is corresponding to defining a variable in C++. This step is necessary because the variables have to be defined as a specific data type in C++. The next step for the coder is to check if the format of the MATLAB code is appropriate for C++ condition, such as if the array size is predefined instead of being increased in size in a loop because C++ does not allow re-sizeable array in normal condition. After that, the function is then tested with user-defined input within the coder to check if it can generate the desirable output. After these steps that make sure the code can be adapted in a C++ environment, a MEX file is generated. Alongside with the interface files between MATLAB and C++, and the corresponding MEX function can be called within the script just as any other MATLAB function.

arr4mat_mex is generated as the corresponding MEX file for **arr4mat**. This is because **arr4mat** is the function that introduced the most time lapse by the repetitive sign and element determination process. It was hoped that C++ compiler can speed up the computational time of this function.

However, figure 9 shows that the matrix-less operation in C++ did not improve the computational time by a great amount. It starts with the improvement by at least an order in magnitude faster, but slowly, the amount of time reduced by the mex function is decreased with the increase in the order of derivatives. C++ compiler is faster then MATLAB in most cases but not in matrices operation. MATLAB uses highly-optimised libraries, Intel oneAPI Math Kernel Library(MKL), for its matrix operation, which optimised Basic Linear Algebra Subprograms (BLAS), the external library used to carry out matrix operation in C++, by vectorisation [18]. C++ does not have a built-in matrix operation function so a loop that perform dot product is how C++ compute their matrix operation in the first place. Hence, matrix operation is preserved for the operations that are in 13^{th} order or below, and matrix-less operation is used for 14^{th} order or above, which are constrained by the memories limit in MATLAB.

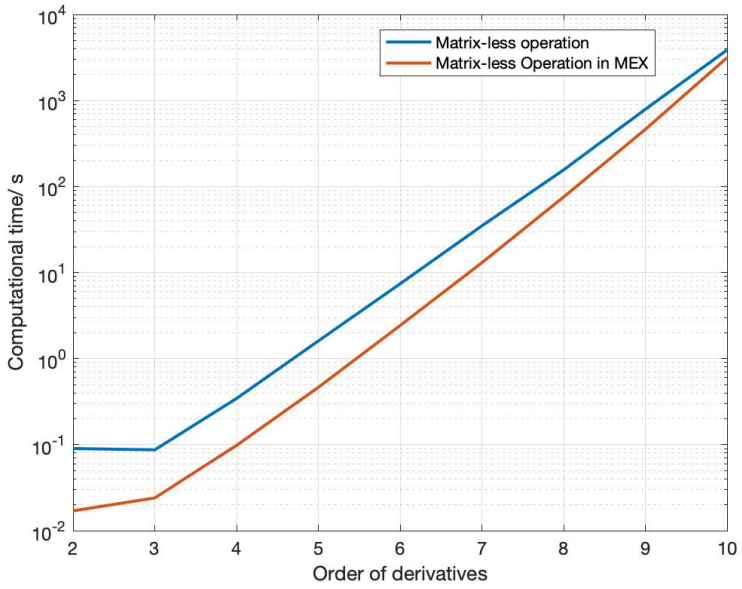


Figure 9: Computational time of the matrix-less operation and MEX version of matrix-less operation

6.4 Further Optimisation

The fractional operation using the binomial expansion method is used by majority of the functions within the class. From equation (51), in each n^{th} term, the power of matrix C is $n-1$. When the binomial expansion method was first implemented as the underlying method of the fractional power operation, 51 terms (I, the identity matrix, is the first term and pC, the product of C matrix and fractional power p, is the second term, etc.) were used to ensure the accuracy of the approximation. However, each C^{n-1} term in the equation caused a huge amount of increase in the computational time. This is because when the matrix-less operation was implemented in the later stage of this project, function **arr4mat** or **arr4mat_mex** was used, These functions are constant repetitions of the same process to determine the appropriate element and corresponding sign in each row for the dot product operation. By using **arr4mat** or **arr4mat_mex**, the calculation of matrix C raise to a big integer power,n, is done by multiplying the matrix together $(n-1)$ times. For example, C^{50}, which appeared in 51^{st} term in the equation, needed to repeat the dot product of $c_{ij} \times c_{ij}$ operation for each row in C matrix for 49 times if **arr4mat** and **arr4mat_mex** were used.

Hence reducing the number of terms used in the binomial expansion would be essential in reducing the computation time of not only the matrix operation, but also the matrix-less operation in fractional power function. A convergence study was carried out to investigate the minimum number of terms in equation (51) needed in order to get a fairly accurate approximation. $f(x) = \sqrt{x^{2x} + x + e^x}$ at $x = 0.5$ was used for testing in the convergence study.

Figure 10 shows results of the convergence study by illustrating the percentage error incurred with 1 to 50 terms of the binomial expansion. In this case however, the number of terms excluded the first, identity matrix term. Hence, when 50 terms were used, the last term of this expansion was the term with C^{50}. In digure 10, the error of the results is very large when only 5 or less than 5 terms were used. Then the error drops almost to its

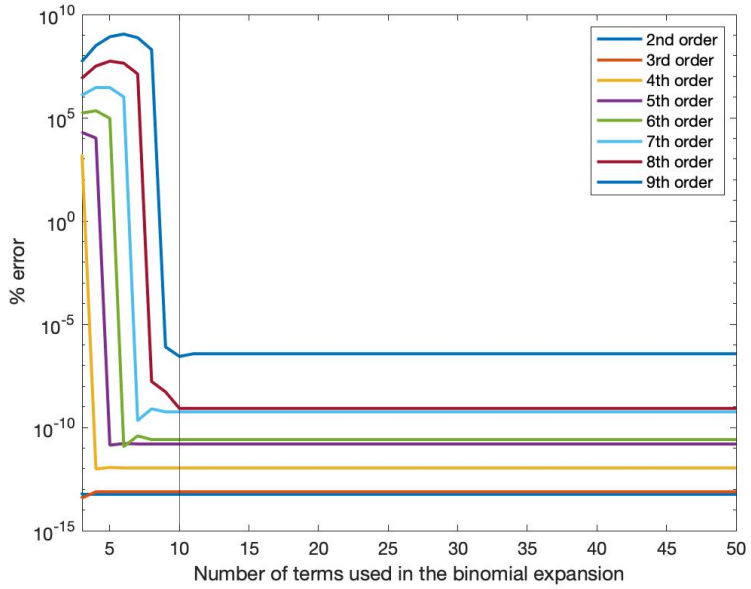


Figure 10: Percentage error of a range of number of terms used in the binomial expansion for 2^{nd} to 9^{th} order

converged level within 1 or 2 extra terms. It is also obvious that with the increase in the order of derivatives, subsequently more terms are needed in order to achieve convergence. At round 10 terms, all orders manage to reach to convergence. A vertical line at $x = 10$ in figure 10 acts as an assistance in determining the point of convergence. Hence, the function **fractionalpow** was amended. Even though the figure shows that the minimum number of terms needed is 10, it was decided 15 terms should be used in the function. 15 is an appropriate number of terms that allows the function to compute an accurate result for higher orders at the end.

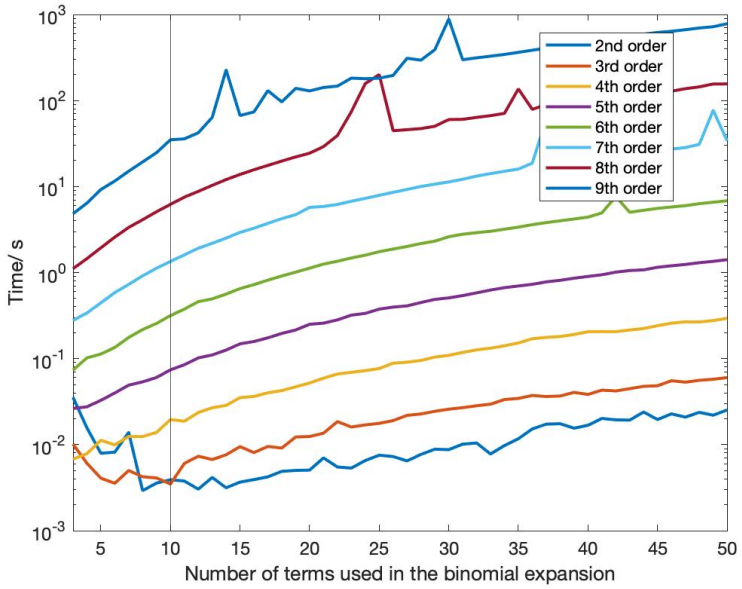


Figure 11: Computational time of a range of number of terms used in the binomial expansion for 2^{nd} to 9^{th} order

In figure 11, the computational time of a range of terms used in the binomial expansion is shown. It can be concluded that, for lower orders calculation, the computational time reduced by using reduced number of terms in the binomial expansion was not significantly noticeable. However, for higher orders calculation, the computational time can be at least reduced by an order in magnitude.

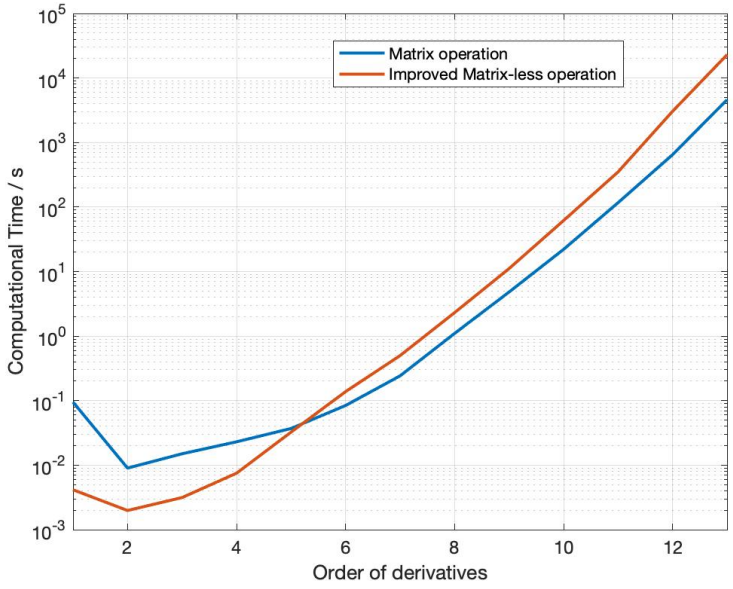


Figure 12: Computational time of Matrix operation and the optimised MEX version of matrix-less operation

Figure 12 shows the time comparison between the matrix operation and the matrix-less operation after all the optimisation mentioned from section 6.3 to section 6.4. When comparing the matrix-less operation performance, it is already much faster than the initial computational time of a matrix-less operation shown in figure 7. This decrease in the computational time allows the programme to complete within a more reasonable time. However, the matrix operation is still faster than the matrix-less. Hence, matrix operation is preserved for lower orders and **arr4mat_mex** would be used for orders higher than 13^{th}.

7 Performance Analysis

After several optimisations was implemented into the class, it is essential to carry out a performance analysis. The objectives of this project, as mentioned in the introduction, are to retain and improve the accuracy of the approximation given by the multicomplex class in terms of computational accuracy, processing time and memories efficiency.

7.1 Approximation Accuracy Analysis

To test the performance in terms of accuracy of the class, several functions were chosen to be tested. The percentage error is the percentage of the relative difference between the approximated result generated from the class function and the analytical result generated from symbolic differentiation, **diff** [19], which is a built-in differentiation tool of

MATLAB, by substituting user-defined x value for $f(x)$. Due to the nature of symbolic differentiation [20], it always gives the differentiated equations in their exact form, even after substituting the x value into the equations. For the convenience of the calculation of the error between the actual value and the estimated value, the exact form of the result of **diff** is converted to number by using **double**. This **double** function gives the value of accuracy in the data type of "double". The floating-point relative accuracy of a double data is 2.2204×10^{-16} [21] so if the relative accuracy calculated is below this value, MALTAB will treat it as an exact solution and will give a relative error of zero. That explained the missing data points in the following figures and figure 3 where the error of the approximated solutions from the class is zero.

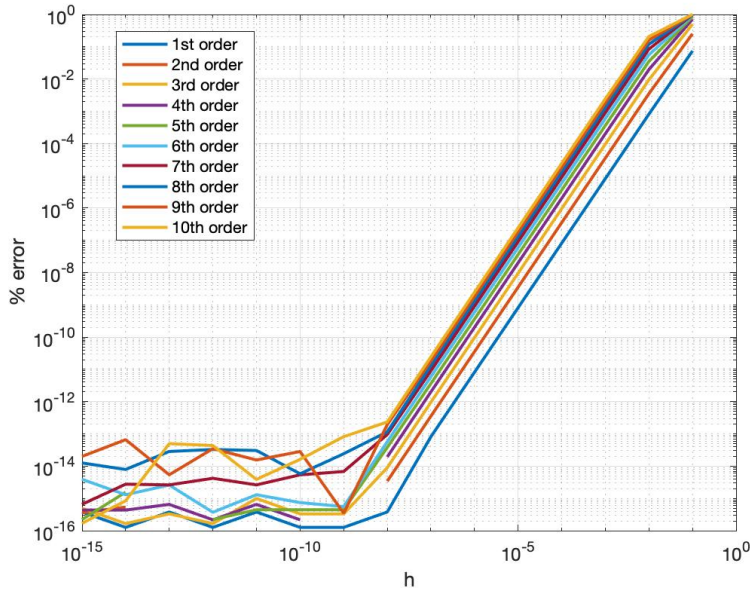


Figure 13: Percentage error of $f(x) = \sqrt{sin(x) + x^2/cos(x)}$ at $x = 5$ at a range of orders

$f(x) = \sqrt{sin(x) + x^2/cos(x)}$, $f(x) = x^{0.3x} + log(x)$ and $f(x) = e^{sin^{-1}(x)}$ were used to test the accuracy of the class. These functions were chosen in [1]. The original intention of using them was because they were able to test all the operations and functions within the class. The additional purpose of using these functions was because the original results from [1] can be used as a reference point for comparison to show that the accuracy of the class was retained or improved.

The class was tested with a range of step sizes h, which were the coefficients of the imaginary parts, from $h = 10^{-15}$ to $h = 10^{-1}$. In figure 13, 14 and 15, a very low percentage error variation shown between $h = 10^{-15}$ to $h = 10^{-8}$ in these three situations. However, the percentage error starts to increase linearly after the step size was increase to $h = 10^{-7}$ and above. This indicates the largest step size the user should use should be around 10^{-8} for an accurate result.

Two more functions were also added into the testing process due to the addition of complex power as a part of **mpower** in the multicomplex class. The first one is $f(x) = (e^{2x} + x + e^x)^{2+2i}$, where the complex power are both integers and the second one is $f(x) = (e^{2x}+x+e^x)^{0.5+0.5i}$, where the complex power are both fractions. Their variation in percentage error against a range of step sizes were plotted in figure 16 and 17 respectively.

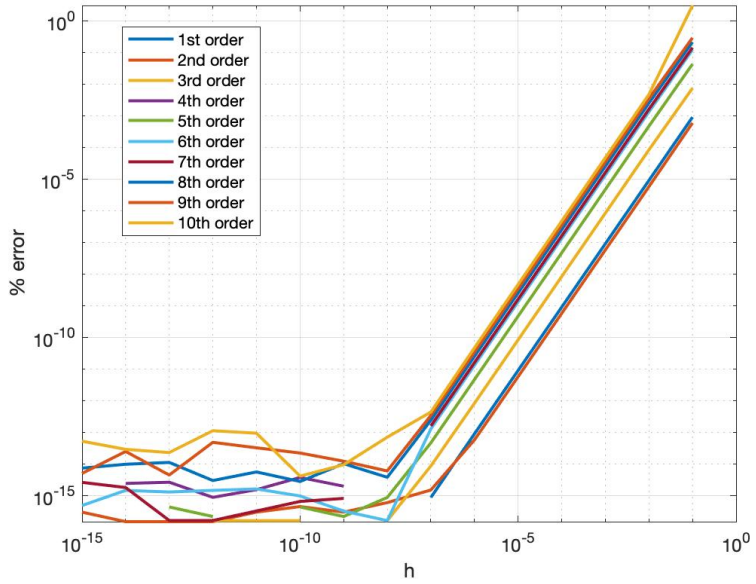


Figure 14: Percentage error of $f(x) = x^{0.3x} + log(x)$ at $x = 2$ at a range of orders

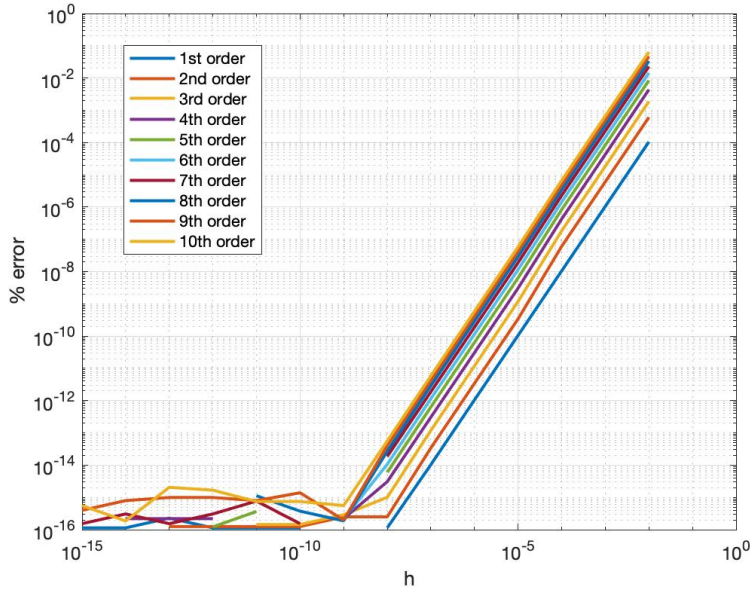


Figure 15: Percentage error of $f(x) = e^{sin^{-1}(x)}$ at $x = 0.5$ at a range of orders

They demonstrate a similar behaviour in terms of the percentage error variation in the smaller h and increase in the error when the step size is around 10^{-7}. Nonetheless, the differences between of the error of the lower orders and the higher orders were much larger comparing to the reset of the function. A possible explanation for this could be that formulation of the complex power involved the use of **log**, **exp**, trigonometry and hyperbolic function. As mentioned in equation (38) in section 5, **log** function in the class introduces a small rounding error. The increase in the order is the increase in the multicomplex domain of the multicomplex number. Also, the function in the class was implemented in a recursive manner, which means the function repeats the same calculation until the input value is in the real number domain. Hence, the higher

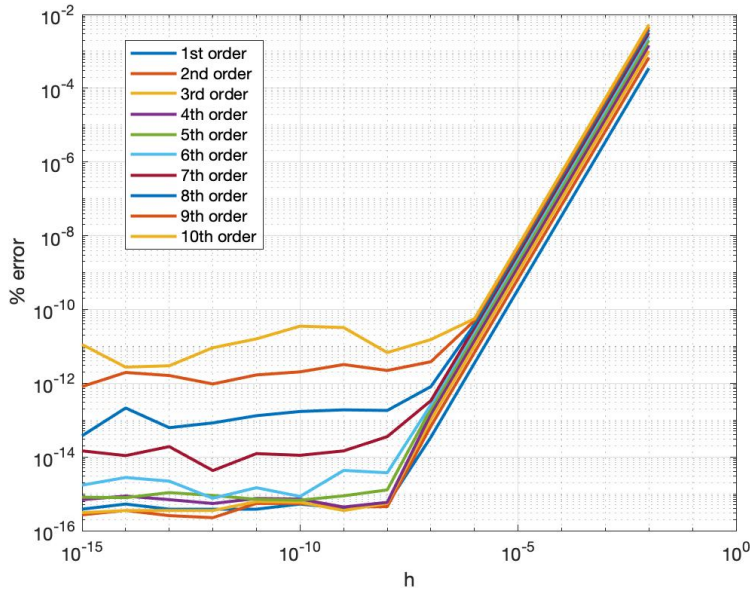


Figure 16: Percentage error of $f(x) = (e^{2x} + x + e^x)^{2+2i}$ at $x = 0.5$ at a range of orders

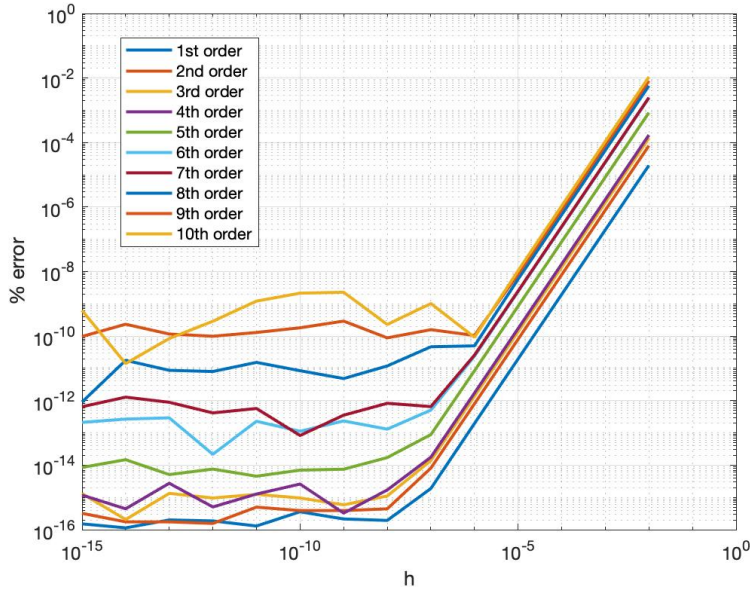


Figure 17: Percentage error of $f(x) = (e^{2x} + x + e^x)^{0.5+0.5i}$ at $x = 0.5$ at a range of orders

the order of the multicomplex number, the greater the number of the recursion in **log** function is needed. That leads to the increase in the relative error for the multicomplex step method in a higher order computation. However, this increase in the relative error for the implementation of complex power did not have any significant effect as the class manage to give a fairly precision result even tough it is not at machine precision.

7.2 Computational Time

The computational time is also a main part of the optimisation. During this analysis, the same basic functions and operation were tested as the ones in [1]. However, the x

value in $f(x)$ used in the testing was unclear so $x = 2$ is used throughout the testing in this section as the real part of z_n and 10^{-9} as the coefficient of the imaginary parts. Each functions were also ran for 500 iterations in order to achieve the same level of average computational time in [1]. In the following graphs, the computational speed of all operators and functions are at least a order or two orders in magnitude faster than they were previously tested. It could be because of different values in the functions were used in this test and the test in [1]. Hence, only the general trend of the computational time and improvement, which use order of magnitude instead of seconds as unit, will be discussed.

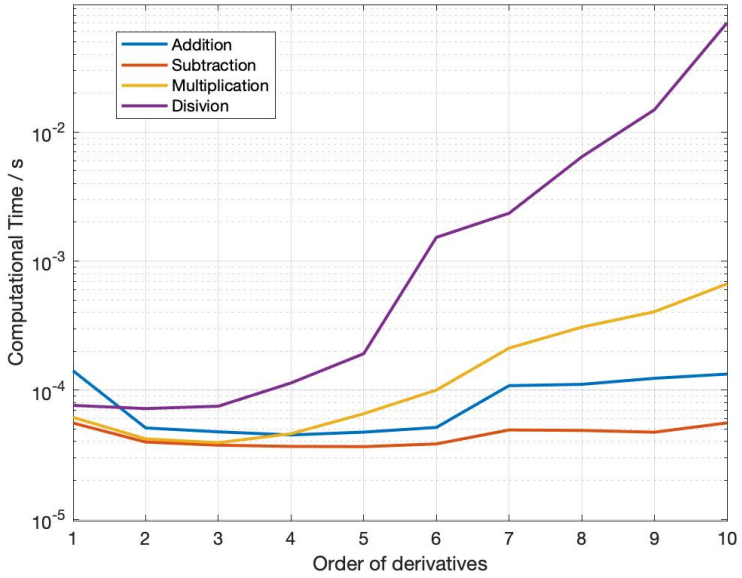


Figure 18: Computational time for basic operation: addition, subtraction, multiplication and division, versus a range of order of derivatives

In figure 18, the computational time of the basic operations demonstrate the same trend as before apart from the multiplication. This is because no optimisation or other changes had been made to these operations due to the fact that their original implementation methods were simple and straightforward. For the multiplication operation, **arr4matmulti**, was developed to carry out matrix-less multiplication operation. This figure shows that the matrix-less operation actually needs less computational time than the matrix operation. The previous tests showed otherwise because the function tested was a combination of fractional power and multiplication operation test, which involved both of **arr4matmulti** and **arr4mat**. For the fractional power, the binomial expansion required the matrix to be raised to a very high power and that repetitive steps were the cause of the long processing time.

Figure 19 shows the performance of the power operations. The integer complex power and fractional complex power were added into the analysis. Both of them needed almost exactly the same amount of time to compute throughout all the orders that were tested. The equation used for complex power is (61), which involve **cos** and **sin** and **log** in the multicomplex class. All of these functions are computed in a recursive manner, which means computational time needed are much longer than other simpler power operation. Take the complex power $a + bi$ as an example, $e^{a\ln(z)}$, $cos(b\ln(z_n)$ and $sin(b\ln(z_n)$ would

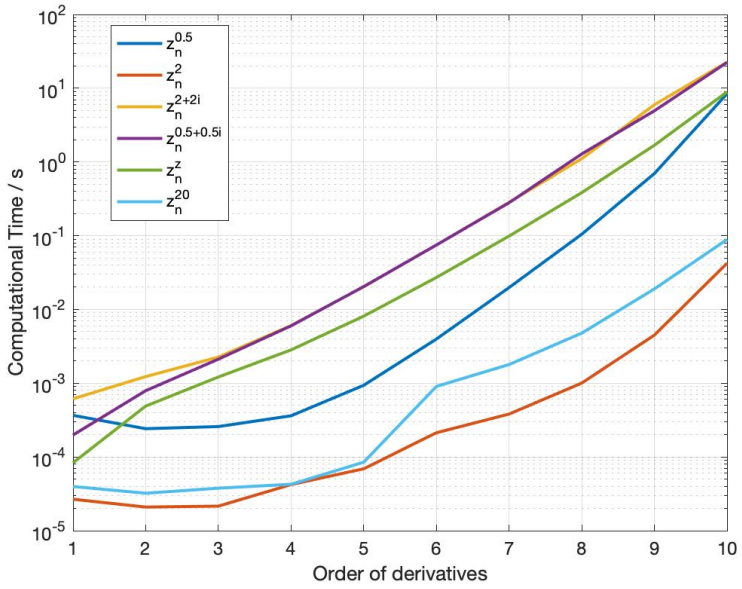


Figure 19: Computational time for power operation: integer, fractional, complex, multicomplex and fractional complex, versus a range of order of derivatives

be compute in the same way regardless the nature of the number a and b. That explained why the computational time of both of integer and fractional complex power were very similar. The rest of the power operation follow a similar trend as before apart from the fractional power. In the previous report, fractional power and integer power had a difference of 3 orders in magnitude of time at the 7^{th} order. After the optimisation, the fractional power manage to decrease the disparity to 2 orders in magnitude different at the 10^{th} order.

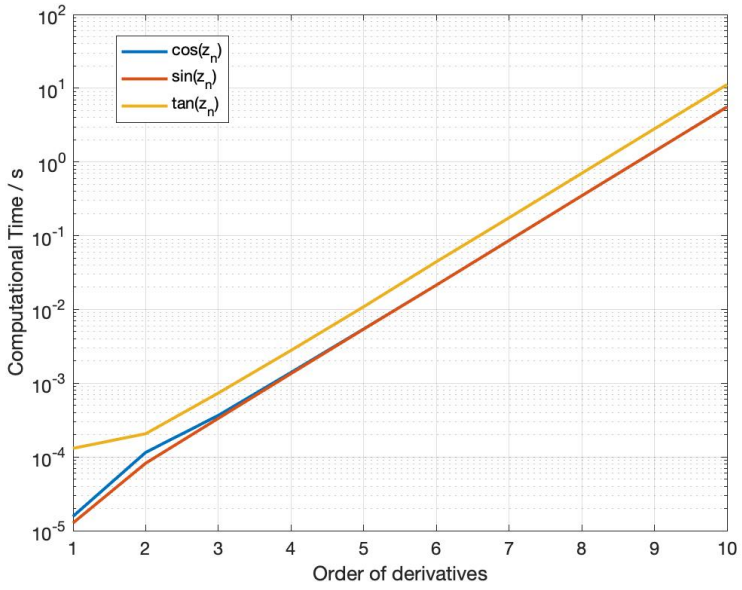


Figure 20: Computational time for trigonometry functions: sin, cos and tan, versus a range of order of derivatives

In figure 20, the computational time of the trigonometry functions also demonstrate the

same trend as before because these functions were not part of the optimisation and their underlying method was not changed by any optimisation implemented in the project.

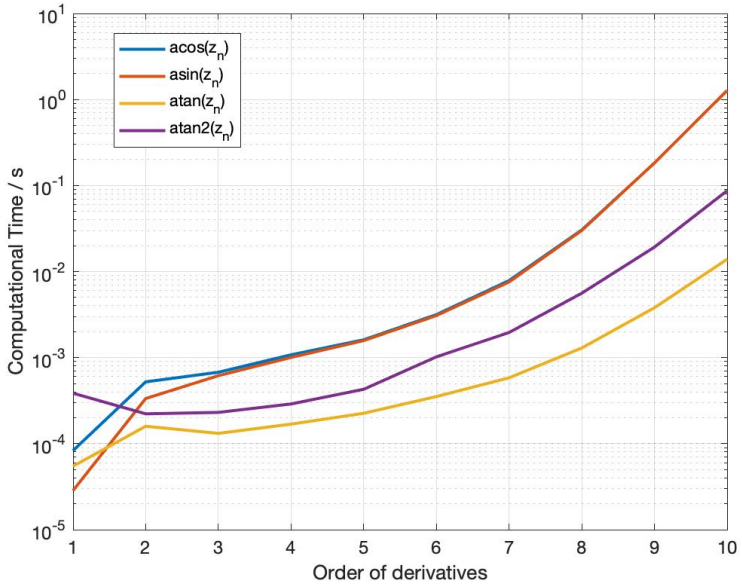


Figure 21: Computational time for inverse trigonometry functions: asin, acos, atan and atan2, versus a range of order of derivatives

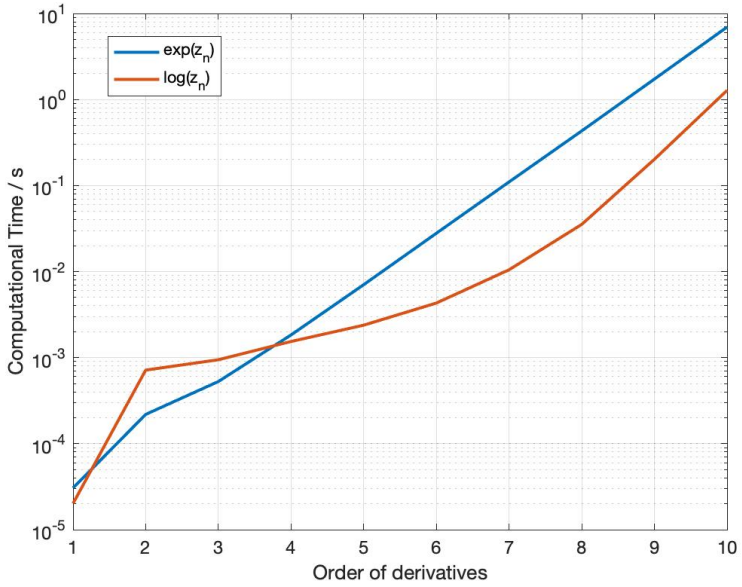


Figure 22: Computational time for exponential functions: exp and log, versus a range of order of derivatives

As shown in figure 21, the computational time for **asin** and **acos** were the same, which is the same as pre-optimised solution. On the other hand, **atan** needed less time to compute than those functions. However, from the result of pre-optimised class definition, computational time of **atan** was the same as **asin** and **acos**. Similarly, in figure 22, the computational time for **log** is lower than the one for **exp** after 4^{th} order. Meanwhile, the old computational time of **log** in the pre-optimised class surpassed the computational

time for **exp** after 2^{nd} order and never dropped below the **exp** level. This phenomenon can be explained by the underlying formulation of **atan**. To compute **atan**, it involves computing a log term according to equation (37). To compute **log**, it is necessary to calculate the modulus of z_n. From equation (43), to calculate the modulus, it involves the use of square root. That means the fractional power is needed for function **log**. The optimisation of the fractional power by using the binomial expansion successfully helped the other functions, which need to use fractional power, to reduced their computational time.

To conclude this section, the accuracy analysis showed that the optimisation helped to increase the accuracy of approximation generated by the class. Moreover, the computational time results showed that the implementation of new function **fractionpow**, helped **mpower** and other related functions to improve their computational time. Meanwhile, the performance of other functions and operations were also retained.

7.3 Limits in the Performance

Despite successfully eliminating the scaling error of the old fractional power function and improve the accuracy and computational time of many other functions, the current fractional power function has its limitations. Functions involving fractional powers have an error that depends on the the order n of multicomplex number system $\mathbb{C}_n$, the x value in the testing function $f(x)$ and the value of fractional power p. To find out the limit of the fractional power function, $f(x) = \sqrt{e^{2x} + x + e^x}$ was used for testing.

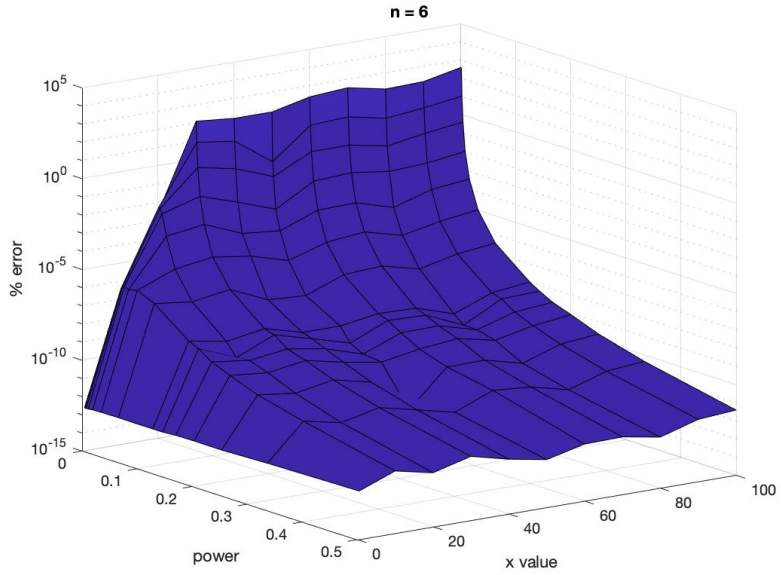


Figure 23: Variation of Error in a range of power and x value in $f(x)$ for $\mathbb{C}_6$

Figure 23 and 24 show that percentage error under each circumstance, which is a range of the fractional power p and the value x in $f(x)$, for multicomplex number in $\mathbb{C}_6$ and $\mathbb{C}_{10}$ space respectively. A range of the fractional power decreasing from 0.5 to 0.001 was tested against a range of x value ranging from 0.5 to 100. In figure 23, it shows that the increase in value x does not change the error by a great amount but the decrease in the value of the fractional power causes a significant different within the same x value. However, in

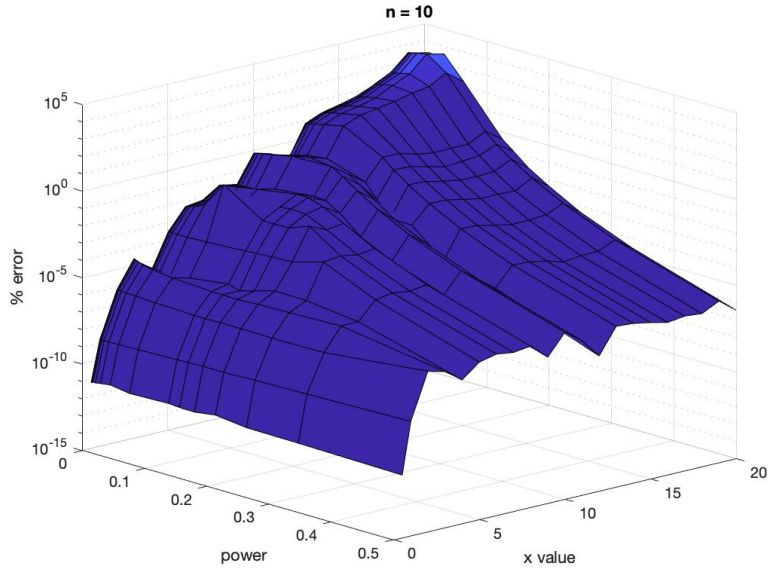


Figure 24: Variation of Error in a range of power and x value in $f(x)$ for $\mathbb{C}_{10}$

figure 24, it is the complete opposite. It is clear that the error is mainly dominated by the value x and the variation in power did not cause significant different within the same x value. This phenomenon can be explained by the multicomplex number in matrix form and the binomial expansion of fractional powered matrix. According to equation (51), the value of the fractional power plays an important part. The smaller the power, the smaller value is contributed by each addition of the terms in the expansion. hence, the sequence converge faster then it should so the approximated value given by equation (51) is significantly smaller than the real value calculated from the symbolic differentiation function. This is a flaw of this implementation for the fractional power. On the other hand, when the multicomplex number is further increased to a higher order, the matrix increase in size exponentially, which cause the matrix, again, tends to a diagonal matrix. This causes the error to increase rapidly with the matrix multiplication operation within the binomial expansion equation. Hence, before the small fractional power can cause an effect on the accuracy of the result, the matrix problem has already contribution more to the error.

The rest of the limitation plots are attached in the appendix.

8 Application of Multicomplex step method in Orbital Mechanic

A lot of the trajectory motion problems in orbital mechanic involve six states, which are the displacement and velocity in the three dimensions. Meanwhile, multicomplex step method can solve functions up to multiple order of derivatives or partial derivative of multi-variables. It is less time consuming to carry out sensitivity analysis by using the multicomplex step method for the multi input variables. A common method for sensitivity analysis is the derivative-based method, which is used in this project. In this section, it first starts the description of the orbital mechanic problem. Then it follows by

the related steps for the carrying out the corresponding sensitive analysis.

8.1 Relative Motion of two bodies

Relative motion describes a spacecraft moving in the position that is relative to another spacecraft, which motion is act as the reference point. Both of them orbit around the same central body. Relative motion is useful in the situation such as to determine one chaser spacecraft relative to one target spacecraft in close proximity [23]. Normally, the situation is the target spacecraft is non-maneuvering and the chaser spacecraft is perturbed in order to perform rendezvous maneuver. Figure 25 illustrates the situation mentioned above where the frame of reference is set as: the target spacecraft A as a reference object of the chaser spacecraft B in the vector direction of $\hat{i}, \hat{j}, \hat{k}$. Hence, even thought spacecraft A is orbiting the center body, its coordinate is fixed to be at (0,0,0) in this frame of reference.

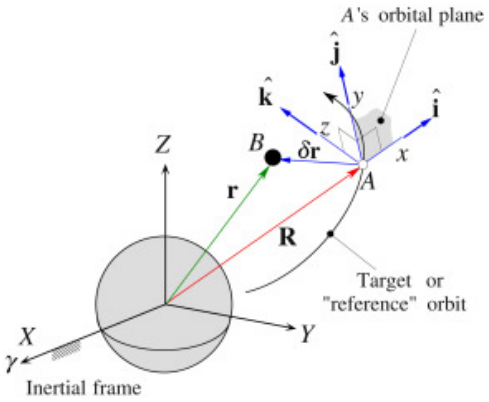


Figure 25: Position of chaser B relative to target A. [23]

In this project, International Space Station (ISS) acts as the reference object and a space shuttle is the chaser object, which aims to catch up with the ISS to deliver supplies. The space shuttle, without any external perturbation, moves in the position that is determined by motion relative to the ISS using Clohessy-Wiltshire equation. Both spacecrafts absolute motion orbiting around the Earth is plotted in figure 26. The Clohessy-Wiltshire equations used for the plotting in figure 26 are [24]:

$$\ddot{x} = 3\omega^2 x + 2\omega\dot{y} \tag{69}$$

$$\ddot{y} = -2\omega\dot{x} \tag{70}$$

$$\ddot{z} = -\omega^2 z \tag{71}$$

where x, y, z are the position relative of the space shutter relative to the ISS, $\dot{x}, \dot{y}, \dot{z}$ are the relative velocity and $\ddot{x}, \ddot{y}, \ddot{z}$ are the acceleration of the space shuttle.

In order to complete the rendezvous manoeuvre, the motion of the space shuttle when perturbation is applied needs to be modelled. The perturbation force acts as an acceleration force, a_x, a_y, a_z is applied to the space shuttle in its x, y and z directions. Hence, the Clohessy-Wiltshire equations can be rewritten as [24],

$$\ddot{x} = 3\omega^2 x + 2\omega\dot{y} + a_x, \tag{72}$$

$$\ddot{y} = -2\omega\dot{x} + a_y, \tag{73}$$

$$\ddot{z} = -\omega^2 z + a_z. \tag{74}$$

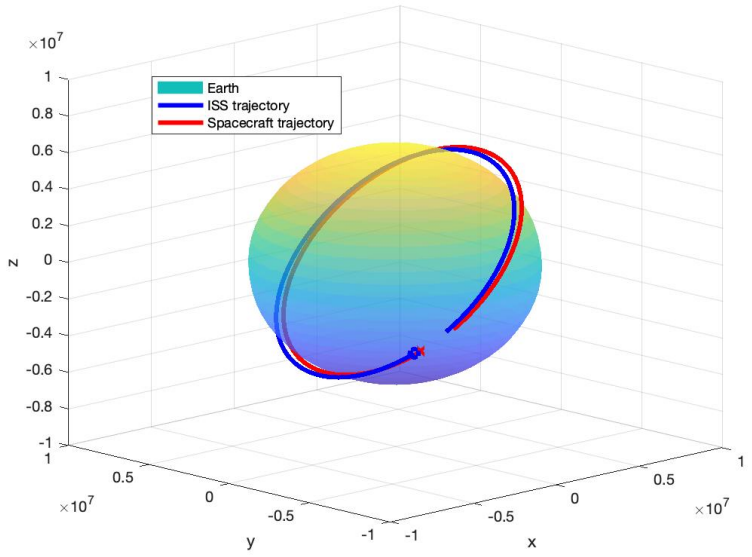


Figure 26: The trajectories of the International Space Station and a spacecraft around the earth

8.2 Numerical Integration

To compute the position of the space shutter relative to the ISS using equations (72) to (74), an numerical integrator needed to be constructed in order to compute the the position in each time step because MALTAB built-in **ODE45** is unable to take multicomplex number as its input. Newmark-beta integrator was implemented due to the fact that it is capable of solving second order differential equations. Newmark-beta method [25] in the x directions is,

$$\begin{aligned} \ddot{x}_{n+1} &= 3\omega^2 x_n + 2\omega \dot{y}_n + a_x, \\ \dot{x}_{n+1} &= \dot{x}_n + (1-\alpha)\Delta t \ddot{x}_n + \alpha \Delta t \ddot{x}_{n+1}, \\ x_{n+1} &= x_n + \Delta t \dot{x}_n + (1-2\beta)\Delta t^2 \ddot{x}_n + \beta \Delta t^2 \ddot{x}_{n+1}. \end{aligned} \tag{75}$$

The same procedure can be repeated for the integration in y and z directions by using equations (73) and (74) for the acceleration step $n+1$. For the use of this class, α and β are chosen to be 0.5 and 0.25 respectively for unconditional stability [25].

8.3 Sensitivity analysis

The sensitivity analysis can study how the uncertainties in the output of a model can be allocated to different sources of uncertainty in the input of the model [22]. This can be determined by the partial derivative of the output with respect to the input. In this case, a space shuttle moving in the position relative to the ISS, a small perturbation can be added to the acceleration, where the acceleration force is constant. Following the Newmark-beta integration, the propagation of the space shutter at each time steps can be calculated. Eventually after a fixed time, the final displacement of the perturbed space shuttle can be found. To find the sensitivity of the displacement with respect to a_x, the partial derivative of displacement in three directions with respect to a_x should be calculated. This is also the same for a_y and a_z.

The concept introduced in equation (27) is helpful in the carrying out of this sensitivity analysis in the **multicomplex** class. The equation also shows its potential and ability in this analysis procedure. Since there are three inputs a_x, a_y and a_z, the partial derivatives with respect to all these three inputs need to be investigated. First of all, the small perturbation was added to the a_x as a i_1 component, which allowed the partial differentiation in the later stage. Similarly, the perturbation of a_y was added as a i_2 component and the perturbation of a_z was added as a i_3 component. Hence, the acceleration in three directions, $\ddot{x}, \ddot{y}, \ddot{z}$, were taken in to the integration as multicomplex numbers. After the Newmark-beta integration, the displacement relative to the ISS was also obtained at each time-step as multicomplex numbers as well. Each of these displacements would be a multicomplex number with the error of a specific acceleration force as the corresponding imaginary component. For instance, the error in acceleration force a_x would in the coefficient of i_1 component from the multicomplex number of x.

To verify the result and error calculated from the **multicomplex** class, finite difference method was also used in the sensitivity analysis. In order to find the partial derivative with respect to a_x, a_y and a_z, the principle introduced in equation (12) was used. The performance of both methods was compared.

Method	Multicomplex step	Finite difference
Number of equation input	3	3
Number of integration	1	2×3
Computational time	10.1344s	0.0551s (×3)

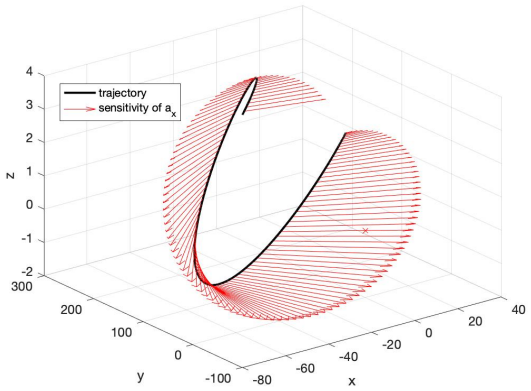
Table 5: Comparison between Multicomplex step and Finite difference

Table 5 shows that both methods required the same equation of motion as input. However, 6 integrations was needed by the finite difference method and only 1 for the multicomplex step method. It is because in this case of having three different input, the perturbation was added separated to a_x, a_y and a_z one by one as the finite difference method is only able to process one variable as the input, such as

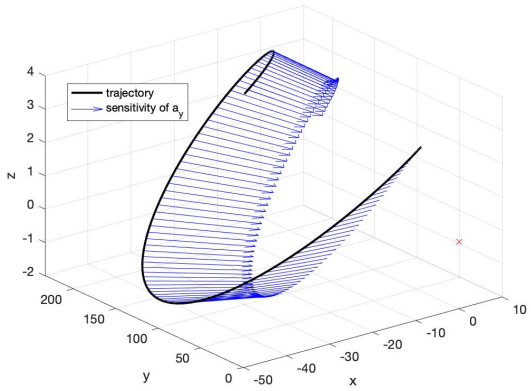
$$\frac{\partial x}{\partial a_x} = \frac{x(a_x + h) - x(a_x)}{h}. \tag{76}$$

Therefore, to compute the partial derivative of each input variables, the small perturbation needed to be added manually one by one for each integration. However, given the convenience of multicomplex step method, it took a much longer time to finish than the finite difference method. Part of the reason was due to the complexity multicomplex number, the code had to call functions within multicomplex class for every steps within the integration. Another reason was the integrator was not built-in, hence, it would not be as optimised as the MATLAB built-in integrator **ODE45**. The sensitivity plots generated by the finite difference method are included in the Appendix and they are coherent with the plot generated by the multicomplex step method.

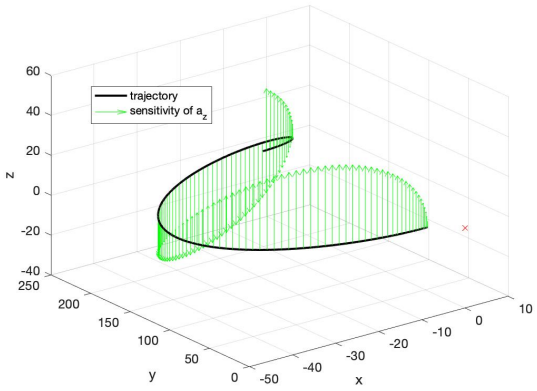
Figure 27a is the sensitivity in position with each x direction acceleration perturbation of 10^{-9} in magnitude. Same for figure 27b and 27c, which are the sensitivity in y and z direction acceleration respectively. These figures were verified by the finite different method. The trajectory motions plotted on each graphs are all about 1 period of the ISS, which is 93 minutes.



(a) Sensitivity in position with each x direction acceleration perturbation



(b) Sensitivity in position with each y direction acceleration perturbation



(c) Sensitivity in position with each z direction acceleration perturbation

Figure 27: Sensitivity in position with each directions acceleration perturbation by multicomplex step

In order to understand the meaning behind this analysis, it is essential to know that the arrows plotted are the sensitivity in the vector form. They point at the directions where the points will move with an extra increase in the acceleration force. Their magnitude represents how sensitive the points are to the changes in the acceleration force. Larger the arrow in its magnitude, the more sensitive it is to the acceleration change.

Figure 27b is the easiest to interpret. It is clear that the increase in the a_y would move the trajectory motion of the space shuttle in both of the x and y directions closer to the ISS, which is represented by the red cross in the figures above. For the y direction, the space shuttle would be less sensitive to the increase in acceleration at the beginning. The motion in y direction with the march of time step becomes more sensitive to the perturbation as the size of the arrows becomes larger in comparison. However, in the x direction, the sensitivity to a_y increases with in the march in time then decreases again once reached half way of one period of orbiting.

Figure 27a shows a more complex behaviour. Similar to the behaviour shown in figure 27b, the trajectory motion in y direction also move closer to the target point with in the increase in a_x. It also follows the same behaviour of having an increasing sensitivity with the march in time then start to decrease once reached half of a period. However, the behaviour of the motion in the x direction is more complex. In the first half of the period

that the space shuttle travels, increase in a_x could move the space shuttle closer to the target in x direction. On the other hand, in the second half of the period, increase in a_x would push the space shuttle further away from the target in x direction. That shows the acceleration force applied in one direction would not monotonically move the space shuttle close to the target in all direction.

Since the motion in z direction is independent of the motion in x and y directions, any changes in a_z would not have an effect on the motion in x and y direction as well. This is also demonstrated in figure 27c. The sensitivity arrows show that increase in would first push the space shuttle in the opposite direction of target in the first half of the period then push it towards the target in the second half. Hence, when carrying out the optimisation the completely opposite behaviour of sensitivity in the each half of one period is worth noting.

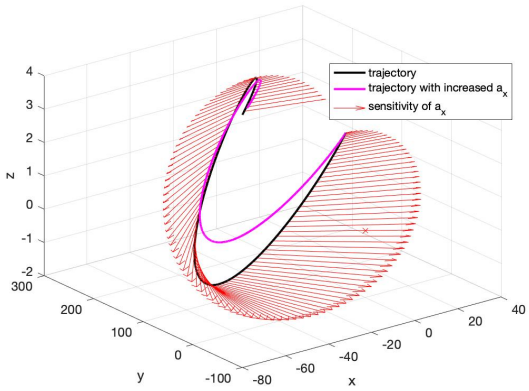
Experiments of increasing the acceleration force by the increment of 10^{-6} in each of the three directions were done. The behaviour was then compared with the sensitivity arrow plot to see if they follow the same trend.

Figure 28b and 28c, as described before, both of the trajectories with increased a_y and a_z follow the trend suggested by the sensitivity arrows. However, in figure 28a, the trajectory with increases acceleration force in x direction is not moving exactly as the arrow suggested. This is because the motion is not as sensitive to the changes in a_x than in a_y. The graphs was plotted in the way for better visualisation and in fact, despite the sensitivity arrows seem to be the same size in all plots, the size of sensitivity with respect to a_y is an order of magnitude larger. Hence, this suggested that the increment in a_x and a_z could be larger then 10^{-6} in order to achieve desirable result in the case of optimising the rendezvous trajectory. Back to figure 28a, on the other hand, the y direction motion of the space shuttle does move closer to the target. This phenomenon is also coherent with the suggestion made earlier, which is the changes in certain acceleration force do not monotonically change the perturbed trajectory towards the target in all direction.

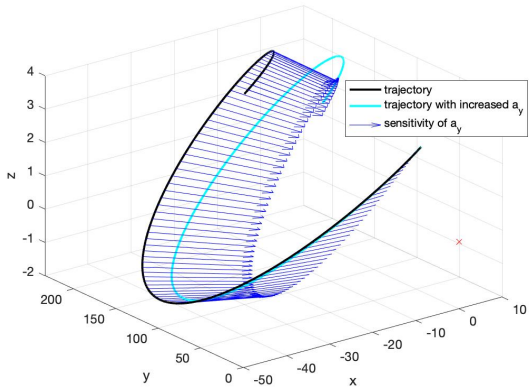
9 Conclusion

As mentioned in the introduction, the main objectives of this project were to maintain and improve the performance of this class in terms of approximation accuracy, computational time and memory efficiency. In this project, the areas that could be optimised was first identified. Possible explanations of the root of these problems were then analysed. Followed by the understanding of the underlying cause of these issues, several solutions were proposed for each of the problems. Several revised implementation methods were tested against the original implementation method then a final implementation that gave the best result or managed to give a result was chosen. With all these steps done, a final performance analysis was carried out to test the overall performance of the class as well as each function. The main purpose of this analysis was to investigate if the main objectives set up at the beginning of the project was achieved.

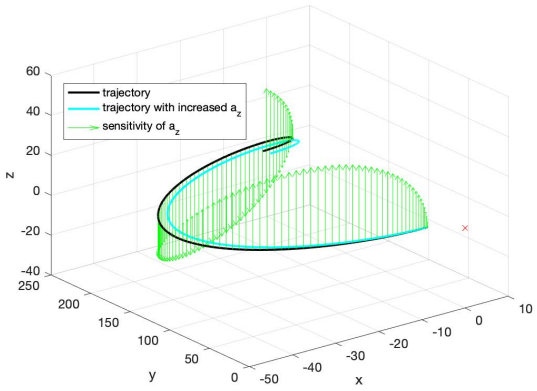
To further elaborated the above context, first of all, three main problems within the class needed to be solved: The scaling error issue of the fractional power function within the class started to increase with the increase in the order of multicomplex number; the lack of complex power; and the limitation of the order of multicomplex number this class was



(a) Sensitivity in position with each x direction acceleration perturbation



(b) Sensitivity in position with each y direction acceleration perturbation



(c) Sensitivity in position with each z direction acceleration perturbation

Figure 28: Sensitivity in position with each directions acceleration perturbation by multicomplex step with an extra trajectory by the increased perturbation force

able to take.

The first problem was also the biggest because many other functions were dependent on fractional power. The underlying causes were the rounding error and the uninvertible matrix with the increase in its size and diagonal value. Both of these problems were unavoidable due to the use of de Moivre's theorem. Hence de Moivre's theorem was replaced by a binomial expansion method of the fractional powered matrix, which successfully eliminated the problems.

The second problem was the lack of complex power. Multicomplex power, which is a similar concept to complex power, was implemented in the class. However, it needed a completely different computational method due to the method of calculating the final result. The basic equation of transforming the complex power function in the exponential and logarithm terms did not give the desired result until it was further expanded by de Moivre's theorem. This implementation of complex power was added as a new member in the power operation of the class.

The third problem was caused by the matrix operation used in different function within the class. It was noticed that the highest order of multicomplex number the class was

able to process was thirteenth. That was caused by the memory constrain in MATLAB. The solution was to remove the matrix from the matrix operation by carrying out dot product row by row. To implement the matrix-less operation, a couple of methods were developed in order to extract the rows from the matrix. The first method was to swap the order of each row by using the first row as a benchmark but the sign of each element in the cannot be determined easily. The second method used the recursive nature of the matrix and determine the elements in each row by using its location index in the matrix. This method is more time-consuming but accurate so it was chosen to carry out the matrix-less operation. To speed up this time-consuming process, MEX files of the corresponding functions were created so this function could be computed in the C++ compiler.

After the implementation, different mathematical functions that consisted of all the functions in the class were tested. The aim of this was to confirm that the new implementations improved the performance of the old functions but did not alter the well-functioning ones. It was clear that the accuracy of the class was maintained. The fractional power function also managed to improved the accuracy of the estimated value calculated from the class. The computational time of each of the functions was compared with the computational time of the pre-optimised functions. It was shown that the new fractional power function successfully helped the operations and the functions that are dependent on it to have a faster computational process. Even though the matrix-less power operation takes significantly more time to compute than the matrix-operation for fractional power, it could still be used for fourteenth order and above, which break the ceiling of the original class. On the other hand, the matrix-less multiplication operation successfully reduced the computation time for simple multiplication calculation.

The class was further tested by carrying out a sensitivity analysis of an orbital mechanic problem. It was to find out the external perturbation needed for a rendezvous manoeuvre of a space shuttle that travels relative to the ISS. The result was interesting as it gave readers an insight into how the increment in an input, which is the acceleration force in a specific direction in this case, would not necessarily monotonically change all the outputs in the same direction. By considering how sensitive the direction is to the additional acceleration force, a further optimisation could be carried out to minimise the time or the fuel needed for a rendezvous manoeuvre.

Overall, it was a successful project in terms of how it achieved the initial objective. However, in the further prospect, the **multicomplex** class can be further optimised to eliminate the limitation of the existing fractional power. Other better implementation of the matrix-less operation is also worth exploring so the class be more memory efficient.

References

[1] J. M. Varas Casado, Multicomplex Numbers for Sensitivity Analysis and Optimization. Master's thesis, Imperial College London, 2018.

[2] Y. Shimane, Multicomplex-Step Method for Space Trajectory Design. Master's thesis, Imperial College London, 2020.

[3] J. N. Lyness and C. B. Moler, Numerical Differentiation of Analytic Functions, SIAM Journal on Numerical Analysis, Vol. 4, No. 2, pp. 202-210. 1967.

[4] W. Squire G. Trapp, Using Complex Variables to Estimate Derivatives of Real Functions, 1998 Society for Industrial and Applied Mathematics. 1998

[5] G. B. Price, An Introduction to Multicomplex Spaces and Functions, Chapman Hall/CRC Pure and Applied Mathematics. Taylor Francis, 1990.

[6] A. Verheyleweghen, Computation of higher-order derivatives using the multi-complex step method, 2014.

[7] G. Lantoine, R. P. Russell T. Dargent, Using multicomplex variables for automatic computation of high-order derivatives. ACM Transactions on Mathematical Software (TOMS), 2012.

[8] J. Cockle, London-Dublin-Edinburgh Philosophical Magazine, Series 3. On a New Imaginary in Algebra. 34:37–47. 1849

[9] A. Quarteroni , F. Saleri. Numerical differentiation and integration. In: Scientific Computing with MATLAB and Octave. Texts in Computational Science and Engineering, vol 2. Springer, Berlin, Heidelberg, 2006.

[10] J. Martins, P. Sturdza, J. Alonso. The complex-step derivative approximation. ACM Transactions on Mathematical Software, Association for Computing Machinery, pp.245 -262, 2003,

[11] Mathworks, mpower. `https://uk.mathworks.com/help/matlab/ref/mpower.html` [Last assessed 12 May 2021]

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

[13] A. Björck S. Hammarling, A Schur method for the square root of a matrix. Numerical Algorithms Group Limited, 1983.

[14] F. Waugh M. Abel, On Fractional Powers of a Matrix. Journal of the American Statistical Association, p. 1018-102, 1967.

[15] Timothy G. Feeman. Complex Numbers. In: The Mathematics of Medical Imaging. Springer Undergraduate Texts in Mathematics and Technology. Springer, New York, NY. 2010

[16] Mathworks, digit. `https://uk.mathworks.com/help/symbolic/digits.html`.[Last assessed 15 May 2021]

[17] Mathworks, memory. `https://uk.mathworks.com/help/matlab/ref/memory.html`[Last assessed 17 May 2021]

[18] T. Andrews, Computation Time Comparison Between Matlab and C++ Using Launch Windows. California Polytechnic State University San Luis Obispo, SLO, CA, 93407, USA. 2012

[19] MATLAB, diff. `https://uk.mathworks.com/help/matlab/ref/diff.html`[Last assessed 20 May 2021]

[20] Watt, Stephen M. Making Computer Algebra More Symbolic (Invited) (PDF). Proc. Transgressive Computing 2006: A conference in honor of Jean Della Dora, (TC 2006). pp. 43–49. 2006.

[21] Mathworks, double. `https://uk.mathworks.com/help/matlab/ref/double.html`.[Last assessed 23 May 2021]

[22] A. Saltelli, Sensitivity Analysis for Importance Assessment. Risk Analysis. 22 (3): 1–12. 2002.

[23] Howard D. Curtis, Orbital Mechanics for Engineering Students (Third Edition), Chapter 7 Relative Motion and Rendezvous, Page 367-404. 2014.

[24] R. Hewson, Advanced Flight Mechanics, Aeronautics Department, Imperial College London. 2020

[25] J. Garza and H. Millwater, Multicomplex Newmark-Beta Time Integration Method for Sensitivity Analysis in Structural Dynamics, University of Texas at San Antonio, San Antonio, Texas 78249. 2015.

A Multicomplex class Definition in MATLAB [1]

```
classdef multicomplex % Of the form zn:[a1,a2,...,an]
    %{
    Copyright 2018, 2019 Jose Maria Varas Casado and Robert Hewson,
    Imperial College London.

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.
    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.
    You should have received a copy of the GNU General Public License
    along with this program.  If not, see <https://www.gnu.org/licenses/>.
    %}
    % edited by Chung Chung To in 2021
    %% Properties %%

    properties
        zn          % The object zn is a matix of the form zn:[a1,a2,...,an]
    end

     % Flag for check of convergence and check for zero divisors -
     % 'F' : False, 'T' = True
     % Recomended to leave False
     properties (Constant = true)
       check_convergence = 'F';
       check_Z0 = 'F';
    end

    %% Initialization %%

    methods
        function self=multicomplex(a)          % Returns error
```

```
            if there are more than one input
36          if nargin ~= 1
37              error('Requires 1 array input')
38          end
39          if mod(log2(length(a)),1) ~= 0    % Multicomplex
               numbers need 2^n coefficients, for C_n
40              error('Requires 2^n input size')
41          end
42
43          self.zn=a;                        %
               Initialization of ClassDef
44
45      end
46  end
47
48
49  methods
50 %% Basic Operator Overloading %%
51
52      function mrep = matrep(self)
                                              % Matrix
           Representation
53
54          % The matrix representation can only go up to a
               certain C_n (in
55          % this case C_10). You could change this easily
               by adding more if
56          % statements inside the while loop. However,
               MATLAB has a limit in
57          % the memory used to store a matrix, so for n>13,
                the number of
58          % elements inside the matrix would more than 2^14
                x 2^14, which is more
59          % than the maximum allowed. This imposes an upper
                limit on the
60          % order of the multicomplex number that MATLAB
               can perform
61          % operations with.
62
63          sz=length(self.zn);
64
65          n=log2(sz);
66
67          if length(self.zn) == 1
68
69              mrep=[self.zn];
70
71          end
```

```
72
73              i=2;
74
75              while i<=sz
76
77                  m1=self.zn(i-1);
78                  m2=self.zn(i);
79
80                  C1=[m1,-m2;m2,m1];
81                  if rem(i,2^2) == 0
82                      C2 = [storeC1,-C1;C1,storeC1];
83                  if rem(i,2^3) == 0
84                      C3 = [storeC2,-C2;C2,storeC2];
85                  if rem(i,2^4) == 0
86                      C4 = [storeC3,-C3;C3,storeC3];
87                  if rem(i,2^5) == 0
88                      C5 = [storeC4,-C4;C4,storeC4];
89                  if rem(i,2^6) == 0
90                      C6 = [storeC5,-C5;C5,storeC5];
91                  if rem(i,2^7) == 0
92                      C7 = [storeC6,-C6;C6,storeC6];
93                  if rem(i,2^8) == 0
94                      C8 = [storeC7,-C7;C7,storeC7];
95                  if rem(i,2^9) == 0
96                      C9 = [storeC8,-C8;C8,storeC8];
97                  if rem(i,2^10) == 0
98                      C10 = [storeC9,-C9;C9,storeC9];
99                  if rem(i,2^11) == 0
100                     C11 = [storeC10,-C10;C10,storeC10];
101                 if rem(i,2^12) == 0
102                     C12 = [storeC11,-C11;C11,storeC11];
103                 if rem(i,2^13) == 0
104                     C13 = [storeC12,-C12;C12,storeC12];
105                 end
106                     storeC12 = C12;
107                 end
108                     storeC11=C11;
109                 end
110                     storeC10=C10;
111                 end
112                     storeC9=C9;
113                 end
114                     storeC8=C8;
115                 end
116                     storeC7=C7;
117                 end
118                     storeC6=C6;
119                 end
```

```
120                     storeC5=C5;
121                 end
122                     storeC4=C4;
123                 end
124                     storeC3=C3;
125                 end
126                     storeC2=C2;
127                 end
128                 storeC1=C1;
129
130                 i=i+2;
131             end
132             if n == 1
133                 mrep=C1;
134             end
135             if n == 2
136                 mrep=C2;
137             end
138             if n == 3
139                 mrep=C3;
140             end
141             if n == 4
142                 mrep=C4;
143             end
144             if n == 5
145                 mrep=C5;
146             end
147             if n == 6
148                 mrep=C6;
149             end
150             if n == 7
151                 mrep=C7;
152             end
153             if n == 8
154                 mrep=C8;
155             end
156             if n == 9
157                 mrep=C9;
158             end
159             if n == 10
160                 mrep=C10;
161             end
162             if n == 11
163                 mrep=C11;
164             end
165             if n == 12
166                 mrep=C12;
167             end
```

```
168             if n == 13
169                 mrep=C13;
170             end
171             if n == 14
172                 mrep=C14;
173             end
174         end
175
176         function out = plus(obj1,obj2)
                                        % Addition
177
178             % MATLAB accepts putting multicomplex numbers
                    inside arrays, so
179             % the for loop inside the addition and
                    substraction operator
180             % overloading allows for the elementswise
                    addition and
181             % substraction of same sized arrays of
                    multicomplex numbers.
182
183             siz=size(obj1);
184             %outfor = zeros(1,siz(2));
185             for i=1:siz(2)
186                 [obj11,obj21] = consimulti(obj1(i),obj2(i));
187                 outfor(i)=multicomplex(obj11.zn+obj21.zn);
188             end
189             out=outfor;
190         end
191
192         function out = minus(obj1,obj2)
                                        % Substraction
193
194             % Same algorithm structure as the addition
                    operator.
195
196             siz=size(obj1);
197             %outfor = zeros(1,siz(2));
198             for i=1:siz(2)
199                 [obj11,obj21] = consimulti(obj1(i),obj2(i));
200                 outfor(i) = multicomplex(obj11.zn-obj21.zn);
201             end
202             out=outfor;
203         end
204
205         function out = uminus(obj1)
                                            % Unary Minus
206             out=multicomplex(-obj1.zn);
207         end
```

```
208
209         function out = uplus(obj1)
                                                % Unary Plus
210             out=multicomplex(+obj1.zn);
211         end
212
213         function out = lt(self,other)               % Less
               than, self < other
214             out = false;
215             [self,other] = consimulti(self,other);
216             if self.zn(1) < other.zn(1)
217                 out = true;
218             end
219         end
220
221         function out = gt(self,other)           % Greater
               than, self > other
222             out = false;
223             [self,other] = consimulti(self,other);
224             if self.zn(1) > other.zn(1)
225                 out = true;
226             end
227         end
228
229         function out = le(self,other)    % Less than or equal
               , self <= other
230             out = false;
231             [self,other] = consimulti(self,other);
232             if self.zn(1) <= other.zn(1)
233                 out = true;
234             end
235         end
236
237         function out = ge(self,other) % Greater than or equal
               , self >= other
238             out = false;
239             [self,other] = consimulti(self,other);
240             if self.zn(1) >= other.zn(1)
241                 out = true;
242             end
243         end
244
245         function out = eq(self,other)               % Equality
               , self == other
246             out = false;
247             [self,other] = consimulti(self,other);
248             if self.zn == other.zn
249                 out = true;
```

```
250             end
251         end
252
253         function out = ne(self,other)              % Not equal
               , self ~= other
254             out = true;
255             [self,other] = consimulti(self,other);
256             if self.zn == other.zn
257                 out = false;
258             end
259         end
260
261         function out = mtimes(self,other)
                                      % Multiplication
262
263             % Multiplication was found to be most efficient
                   wit the built
264             % in matrix multiplication
265
266             [self,other] = consimulti(self,other);
267             %out=multicomplex(arrayM(matrep(self)*matrep(
                   other))); %matrix operation
268             out = multicomplex(arr4matmulti(self,other)); %
                   matrixless operation
269         end
270
271         function out = mrdivide(self,other)
                                    % Division
272
273             % Division was found to be most efficient wit the
                    built
274             % in matrix division
275
276             [self,other] = consimulti(self,other);
277
278             % The following check for zero divisors is only
                   turned on if
279             % specified by the user
280             if other.check_Z0 == 'T'
281                 if det(matrep(other)) == 0
282                     error('Attempting to divide by a zero
                           divizor multicomplex number')
283                 end
284             end
285
286             out=multicomplex(arrayM(matrep(self)/matrep(other
                   )));
287
```

```
288         end
289
290         function out = mpower(input,p)
                                            % Power
291
292             % First we check and see whether the power is
                   itself a multicomplex
293             % number, in which case the solution is of the
                   form e^*log(input)
294
295             l=length(input.zn);
296
297             if isa(p,'multicomplex') == 1
298
299                 [input,p] = consimulti(input,p);
300
301                 out=exp(p*log(input));
302
303             elseif imag(p) ~= 0
304
305                 input = log(input);
306
307                 input1 = cos(input * imag(p)) + sin(input *
                       imag(p))*i;
308
309                 input2 = exp(input*real(p));
310
311                 out = input1*input2;
312
313             elseif rem(p,1) == 0
314
315                 % if the power is an integer, the easiest
                       method of multiplication is to
316                 % use the standard matrix multiplication p
                       times.
317
318                 out=multicomplex(arrayM(matrep(input)^p));
319
320             else
321
322                 % However, if it is a fractional power using
                       an iterative de Moivres theorem is
                       employed.
323                 % This algorithm breakes down the n order
                       multicomplex number into the
324                 % two n-1 components and performs de Moivres.
                        When you reach C_1, the
325                 % built in atan function is employed.
```

```
326                 if l == 1  % This just checks for a real
                       number input
327
328                     out = multicomplex(input.zn^p);
329
330                 else
331
332                     out = multicomplex(arrayM(fractionalpow(
                           matrep(input),p))); % for fractional
                           power
333                     %out = multicomplex(fracpow((input),p))
334                     %out = multicomplex(cppfracpow_mex((input
                           .zn),p));
335                 end
336             end
337         end
338
339     %% Basic Function Overloading %%
340
341
342         function out = atan2(input2,input1)
               % Arctan2
343
344             % Works in a similar way as the other functions,
                   breaking down
345             % the C_n multicomplex number in to C_n-1 until
                   you reach C_1.
346
347             l=length(input1.zn);
348
349             if l == 1   % Again checking for a real number
                   input.
350
351                 out=multicomplex(atan2(input2.zn,input1.zn));
352
353             else
354
355                 % The atan2 version of arctan is coded in
                       here to take into
356                 % account the signs of the inputs.
357
358                 input=input2/input1;
359
360                 % Here we also have the approximation we
                       derived for very
361                 % small imginary components.
362
363                 if  abs(input.zn((l/2)+1)) < 10^-4
```

```
364
365                     if input.check_convergence == 'T' &&
                           modcheck(multicomplex(input.zn(1:l/2))
                           )
366                         % Convergence criteria only turned on
                               if
367                         % specified by the user
368
369                         error('Convergence criteria for atan
                               not met')
370                     end
371
372                     half=l/2;
373
374                     outp1=atan2(multicomplex(input.zn(1:half)
                           ),multicomplex(1));
375                     outp2=multicomplex(input.zn(half+1:end))
                           /(1+(multicomplex(input.zn(1:half)))
                           ^2);
376
377                     out=multicomplex([outp1.zn(1:end),outp2.
                           zn(1:end)]);
378
379                 else
380
381                     input=((((input1^2)+(input2^2))^0.5)-
                           input1)/input2;
382
383                     l=length(input.zn);
384
385                     ar=zeros(1,l);
386                     ar((l/2)+1)=1;
387                     ar=multicomplex(ar);
388
389                     outp=ar*0.5*log(multicomplex(arrayM((
                           matrep(ar+input))/(matrep(ar-input))))
                           );
390                     out=2*outp;
391
392                 end
393
394             end
395
396         end
397
398         function out = atan(input)
               % Arctan
399
```

```
400             l=length(input.zn);
401
402             if l == 1
403
404                 out=multicomplex(atan(input.zn));
405
406             elseif l==2
407
408
409                 outp=atan(input.zn(1)+(1i*input.zn(2)));
410                 out=multicomplex([real(outp),imag(outp)]);
411
412             else
413
414
415                 if  abs(input.zn((l/2)+1)) < 10^-7
416
417                     half=l/2;
418
419                     outp1=atan2(multicomplex(input.zn(1:half)
                            ),multicomplex(1));
420                     outp2=multicomplex(input.zn(half+1:end))
                            /(1+(multicomplex(input.zn(1:half)))
                            ^2);
421
422                     out=multicomplex([outp1.zn(1:end),outp2.
                            zn(1:end)]);
423
424                 else
425
426                     ar=zeros(1,l);
427                     ar((l/2)+1)=1;
428                     ar=multicomplex(ar);
429
430                     out=ar*0.5*log(multicomplex(arrayM((
                            matrep(ar+input))/(matrep(ar-input))))
                            );
431
432                 end
433
434             end
435         end
436
437         function out = log(input)
                                                % Log
438
439             l=length(input.zn);
440
```

```
441             if l == 1
442
443                 out=multicomplex(log(input.zn));
444
445             elseif l == 2
446
447                 % The correction below is for the cases where
                        you need to calculate log(1+x), and x is
                       very small
448                 % It uses the built in log1p(x) ? x which
                       essentially eliminates
449                 % the roundoff error in the log(1+x)
                       calculation.
450
451                 if abs(input.zn(1)) > 0.999999999 && abs(
                       input.zn(1)) < 1.0000000001 && abs(input.
                       zn(2)) < 10^-10
452                     if input.zn(2) > 0
453                         out=multicomplex([0.5*log1p(input.zn
                               (2)^2),atan2(input.zn(2),input.zn
                               (1))]);
454                     else
455                         out=multicomplex([-0.5*log1p(input.zn
                               (2)^2),atan2(input.zn(2),input.zn
                               (1))]);
456                     end
457                 else
458                     out=multicomplex([0.5*log(input.zn(1)^2+
                           input.zn(2)^2),atan2(input.zn(2),input
                           .zn(1))]);
459                 end
460
461             else
462
463                 half=l/2;
464                 L_1 = multicomplex(input.zn(1:half));
465                 L_2 = multicomplex(input.zn(half+1:end));
466
467                 O_1=log(((L_1^2)+(L_2^2))^0.5);
468                 O_2=atan2(L_2,L_1);
469
470                 [O_1,O_2] = consimulti(O_1,O_2);
471                 out=multicomplex([O_1.zn,O_2.zn]);
472
473             end
474         end
475
476
```

```
477         %       All of the trigonometric functions work
               similarly, separating the
478         %       C_n multicomplex numbers into two C_n-1
               coponents, extending the
479         %       trigonometric definitions of C_1 complex
               numbers into the
480         %       multicomplex domain.
481
482
483
484         function out = sin(input)
                                              % Sin
485
486             l=length(input.zn);
487
488             if l == 1
489                 out=sin(input.zn);
490
491
492             elseif l==2
493
494                 out=multicomplex([(sin(input.zn(1))*cosh(
                       input.zn(2))),(cos(input.zn(1))*sinh(input
                       .zn(2)))]);
495
496             else
497
498                 half=l/2;
499                 L_1 = multicomplex(input.zn(1:half));
500                 L_2 = multicomplex(input.zn(half+1:end));
501
502                 O_1=sin(L_1)*cosh(L_2);
503                 O_2=cos(L_1)*sinh(L_2);
504
505                 out=multicomplex([O_1.zn,O_2.zn]);
506
507             end
508         end
509
510
511         function out = cos(input)
                                              % Cos
512
513             l=length(input.zn);
514
515             if l == 1
516
517                 out=cos(input.zn);
```

```
518
519             elseif l==2
520
521                 out=multicomplex([(cos(input.zn(1))*cosh(
                       input.zn(2))),-(sin(input.zn(1))*sinh(
                       input.zn(2)))]);
522
523             else
524
525                 half=l/2;
526                 L_1 = multicomplex(input.zn(1:half));
527                 L_2 = multicomplex(input.zn(half+1:end));
528
529                 O_1=cos(L_1)*cosh(L_2);
530                 O_2=-sin(L_1)*sinh(L_2);
531
532                 out=multicomplex([O_1.zn,O_2.zn]);
533
534             end
535         end
536
537
538         function out = sinh(input)                              %
               Sinh
539
540             l=length(input.zn);
541
542             if l == 1
543
544                 out=sinh(input.zn);
545
546
547             elseif l==2
548
549                 out=multicomplex([(sinh(input.zn(1))*cos(
                       input.zn(2))),(cosh(input.zn(1))*sin(input
                       .zn(2)))]);
550
551             else
552
553                 half=l/2;
554                 L_1 = multicomplex(input.zn(1:half));
555                 L_2 = multicomplex(input.zn(half+1:end));
556
557                 O_1=sinh(L_1)*cos(L_2);
558                 O_2=cosh(L_1)*sin(L_2);
559
560                 out=multicomplex([O_1.zn,O_2.zn]);
```

```
561
562             end
563         end
564
565
566         function out = cosh(input)
                                            % Cosh
567
568             l=length(input.zn);
569
570             if l == 1
571
572                 out=cosh(input.zn);
573
574
575             elseif l==2
576
577                 out=multicomplex([(cosh(input.zn(1))*cos(
                       input.zn(2))),(sinh(input.zn(1))*sin(input
                       .zn(2)))]);
578
579             else
580
581                 half=l/2;
582                 L_1 = multicomplex(input.zn(1:half));
583                 L_2 = multicomplex(input.zn(half+1:end));
584
585                 O_1=cosh(L_1)*cos(L_2);
586                 O_2=sinh(L_1)*sin(L_2);
587
588                 out=multicomplex([O_1.zn,O_2.zn]);
589             end
590         end
591
592
593         function out = tan(input)
                                            % Tan
594
595             out=sin(input)/cos(input);
596
597         end
598
599
600         function out = asin(input)
                                           % Arcsin
601
602             % Note that the value of the real part of the
                   input is restricted
```

```
603             % to |x|<1.
604
605             l=length(input.zn);
606
607             if l == 1
608
609                 out=asin(input.zn);
610
611
612             elseif l==2
613
614                 outp=asin(input.zn(1)+(1i*input.zn(2)));
615                 out=multicomplex([real(outp),imag(outp)]);
616
617             else
618
619
620                 if  abs(input.zn((l/2)+1)) < 10^-4
621
622                     half=l/2;
623
624                     outp1=asin(multicomplex(input.zn(1:half))
                        );
625                     outp2=multicomplex(input.zn(half+1:end))
                        /((1-(multicomplex(input.zn(1:half)))
                        ^2)^0.5);
626
627                     out=multicomplex([outp1.zn(1:end),outp2.
                        zn(1:end)]);
628
629                 else
630
631                     half=l/2;
632                     ar=zeros(1,l);
633                     ar((l/2)+1)=1;
634                     ar=multicomplex(ar);
635                     d=1-(input^2);
636
637                     d1=multicomplex(d.zn(1:half));
638                     d2=multicomplex(d.zn(half+1:end));
639
640                     dd1=(d1^2+d2^2)^0.5;
641                     dd2=atan2(d2,d1);
642
643                     out=-ar*log((ar*input)+(((dd1)^0.5)*exp
                        (0.5*ar*dd2)));
644
645                 end
```

```
646             end
647         end
648
649
650         function out = acos(input)                                  %
                Arccos
651
652             % Note that the value of the real part of the
                   input is restricted
653             % to |x|<1.
654
655             l=length(input.zn);
656
657             if l == 1
658
659                 out=acos(input.zn);
660
661
662             elseif l==2
663
664                 outp=acos(input.zn(1)+(1i*input.zn(2)));
665                 out=multicomplex([real(outp),imag(outp)]);
666
667             else
668
669                 if  abs(input.zn((l/2)+1)) < 10^-4
670
671                     out=(pi/2)-asin(input);
672
673                 else
674
675                     half=l/2;
676                     ar=zeros(1,l);
677                     ar((l/2)+1)=1;
678                     ar=multicomplex(ar);
679                     d=1-(input^2);
680
681                     d1=multicomplex(d.zn(1:half));
682                     d2=multicomplex(d.zn(half+1:end));
683
684                     dd1=(d1^2+d2^2)^0.5;
685                     dd2=atan2(d2,d1);
686
687                     out=-ar*log(input+(ar*(((dd1)^0.5)*exp
                           (0.5*ar*dd2))));
688
689                 end
690             end
```

```
691         end
692
693
694         function out = exp(input)                                   %
               Exp
695
696             l=length(input.zn);
697
698             if l == 1
699
700                 out=exp(input.zn);
701
702             elseif l==2
703
704                 outp=exp(input.zn(1)+(1i*input.zn(2)));
705                 out=multicomplex([real(outp),imag(outp)]);
706
707             else
708
709                 half=l/2;
710                 L_1 = multicomplex(input.zn(1:half));
711                 L_2 = multicomplex(input.zn(half+1:end));
712
713                 O_1=exp(L_1)*cos(L_2);
714                 O_2=exp(L_1)*sin(L_2);
715
716                 out=multicomplex([O_1.zn,O_2.zn]);
717
718
719             end
720         end
721
722         %% Utility functions %%
723
724         function out = imgn(C)
725
726             % This function essentially ust extracts the '
                   last' imaginary term, or the
727             % one that contains all of the imaginary terms.
728
729             out=C.zn(end);
730
731         end
732
733
734         function out = CX2(C,im,in)
735
736             % This function extracts the coefficient of the
```

```
                user inputed imaginary part inim. C is the
                multicomplex number
737             % you wan to extract the coefficient of. Ex: CX2(
                C,3,4) extracts the i3i4 coefficient of C.
738             % It is useful for second partial derivative
                calculation.
739
740             if im > in
741                 store = in;
742                 in = im;
743                 im = store;
744             end
745             if in > log2(length(C.zn))
746                 error('input not in required form or out of
                    bounds')
747             end
748
749             cj=1.5;
750             val=4;
751             w=2;
752             u=1;
753             ci=0.5;
754
755             for j=2:in
756                 for i=1:im
757
758                     if j-w == 1
759
760                         cj=(2*cj)-1;
761
762                         val=val+cj;
763
764                         ci=0.5;
765
766                     elseif i-u == 1
767
768                         ci=2*ci;
769
770                         val=val+ci;
771
772                     end
773
774                     w=j;
775                     u=i;
776
777                     if j-i == 1
778                         break
779                     end
```

```
780
781                 end
782
783             end
784
785             out=C.zn(val);
786
787         end
788
789
790         function [self,other] = consimulti(self,other)
791
792             % Verifies that self and other are multicomplex,
                    converts different sized
793             % multicomplex numbers into the same format. This
                     funciton enables the
794             % operations between differently sized
                    multicomplex numbers (and with real
795             % numbers).
796
797             if length(self)~=1 || length(other)~=1
798                 error('inputs to function cannot be arrays or
                        matrices')
799             end
800
801             if isa(self,'double')==1 && isa(other,'
                    multicomplex')==1
802
803                 aa=size(other.zn)-1;
804                 self = multicomplex([self,zeros(1,aa(2))]);
805
806             elseif isa(self,'multicomplex')==1 && isa(other,'
                    double')==1
807
808                 bb=size(self.zn)-1;
809                 other = multicomplex([other,zeros(1,bb(2))]);
810
811             elseif isa(self,'multicomplex')==1 && isa(other,'
                    multicomplex')==1
812
813                 aa=size(other.zn);
814                 bb=size(self.zn);
815
816                 if aa(2) > bb(2)
817
818                     self = multicomplex([self.zn,zeros(1,(aa
                            (2)-bb(2)))]);
819
```

```
820                 elseif aa(2)<bb(2)
821
822                     other = multicomplex([other.zn,zeros(1,(
                            bb(2)-aa(2)))]);
823
824                 else
825
826                 end
827             else
828
829                 error('Inputs to function are not of double
                        or multicomplex type objects')
830
831             end
832
833         end
834
835
836         function out = modcheck(input)
837
838             % Function to be used with atan2 function for the
                    convergence test,
839             % which is needed in order to use the small
                    imaginary term
840             % approximation. Due to the nature of the small
                    imaginary component
841             % multicomplex numbers, when employing atan2 the
                    convergence criteria
842             % is almost always satisfied, so this check
                    should be turned off for
843             % increase in computational efficiency.
844
845             % modc(input) is our first type of modulus
846             % modc2(input) is our second type of modulus
847             if (modc(input) < 1) && (modc2(input^2)<modc2(
                    input))
848                 % Only of both of these are true the
                        multicomplex number is likely
849                 % to converge
850
851                 out=('converge');
852
853             else
854
855                 out=('diverge');
856
857             end
858
```

```
859         end
860
861
862         function out = modc(input)
863             % First type of modulus: recursive
864             l=length(input.zn);
865             input1=input;
866
867             % input1 will be our first type of modulus (
                    recursive)
868             while l > 2
869
870                 l=length(input1.zn);
871                 input1=(((multicomplex(input1.zn(1:l/2)))^(2)
                        )+(multicomplex(input1.zn(l/2+1:end)))^(2)
                        )^(1/2);
872
873             end
874
875             out=input1.zn;
876
877         end
878
879
880         function out = modc2(input)
881             % Second type of modulus: square root of all the
                    coefficients of the
882             % multicomplex object squared.
883
884             out=(sum((input.zn(1:end)).^(2))^(1/2));
885
886
887         end
888
889
890         function out = conj(input)
891             % Complex conjugate
892
893             half = length(input.zn)/2;
894
895             out = multicomplex([input.zn(1:half),-input.zn(
                    half+1:end)]);
896
897         end
898
899     end
900
901 end
```

B Fractional Power Function

```
function A = fractionalpow(m,p)


de=0;
for i = 1:length(m)

    de = de + m(i,i);

end

nu = sum(m(:).^2);

k = nu/de;

I=eye(length(m));
C=(1/k)*m-I;

A = I + p*C;

temp = p*(p-1);

for i = 2:50

    A = A + (temp/factorial(i)) * C^i;

    temp = temp * (p - i);
end

A = (k^p)*A;
end
```

C Matrix-less Power Operation Function

```
function A = arr4mat(f,p) %f is a multicomplex number & p is
   the integer power
arr = f.zn;
for l = 1:p-1
    for i = 1: length(f.zn)
        for j = 1 : length(f.zn)
            b = j;
            a = i;
            x = arr;
            n = length(f.zn);
            while length(x) > 1
                if a <= n/2 & b <= n/2 % upper left
                    x = x(1:n/2);
```

```
                elseif a <= n/2 & b > n/2 % upper right
                    x = -x(n/2 +1 :end);
                    b = b - n/2;
                elseif a > n/2 & b <= n/2 % lower left
                    x = x(n/2 +1 :end);
                    a = a - n/2;
                elseif a > n/2 & b > n/2 % lower right
                    x = x(1:n/2);
                    a = a - n/2;
                    b = b - n/2;
                end
                n = n/2;
            end
            s(j)=x; % up to this point
        end
        y(i) = dot(s,f.zn);
    end
    arr = y;
end
A = y;
end
```

D Matrix-less Multiplication Operation Function

```
function A = arr4matmulti(p,q) %both p and q need to be in
   the same size
x = p.zn;
y = q.zn;
    for i = 1: length(x)
        for j = 1 : length(x)
            b = j;
            a = i;
            n = length(x);
            while length(x) > 1
                if a <= n/2 & b <= n/2 % upper left
                    x = x(1:n/2);
                    y = y(1:n/2);
                elseif a <= n/2 & b > n/2 % upper right
                    x = -x(n/2 +1 :end);
                    y = -y(n/2 +1 :end);
                    b = b - n/2;
                elseif a > n/2 & b <= n/2 % lower left
                    x = x(n/2 +1 :end);
                    y = y(n/2 +1 :end);
                    a = a - n/2;
                elseif a > n/2 & b > n/2 % lower right
                    x = x(1:n/2);
                    y = y(1:n/2);
```

```
                    a = a - n/2;
                    b = b - n/2;
                end
                n = n/2;
            end
            xx(j)=x; % up to this point
            yy(j)=y;
        end
        z(i) = dot(xx,yy);
    end
A = z;
end
```

E Fractional Power Function for MEX version

```
function A = cppfracpow(f,p)

n = length(f);

de = f(1)*n;
nu = n*f(1)^2;
k = nu/de;

C=((1/k)*f)-inputconverter(1,[1:log2(n)],0);

A = inputconverter(1,[1:log2(n)],0) + p * C;

temp = p*(p-1);
s = zeros(1,length(C));
y = zeros(1,length(C));
for i = 2:15

    arr = C;
    for pow = 1:i-1
        for m = 1: length(C)
            for j = 1 : length(C)
                b = j;
                a = m;
                x = arr;
                l = length(C);
                while length(x) > 1
                    if a <= l/2 & b <= l/2 % upper left
                        x = x(1:l/2);
                    elseif a <= l/2 & b > l/2 % upper right
                        x = -x(l/2 +1 :end);
                        b = b - l/2;
                    elseif a > l/2 & b <= l/2 % lower left
                        x = x(l/2 +1 :end);
```

```
                        a = a - l/2;
                    elseif a > l/2 & b > l/2 % lower right
                        x = x(1:l/2);
                        a = a - l/2;
                        b = b - l/2;
                    end
                    l = l/2;
                end
                s(j)=x; % up to this point
            end
            y(m) = dot(s,C);
        end
        arr = y;
    end
    C1 = y;

    A = A + (temp/factorial(i)) * C1;

    temp = temp * (p - i);
end

A = (k^p)*A;
end
```

F Limitation of fractional power in different orders

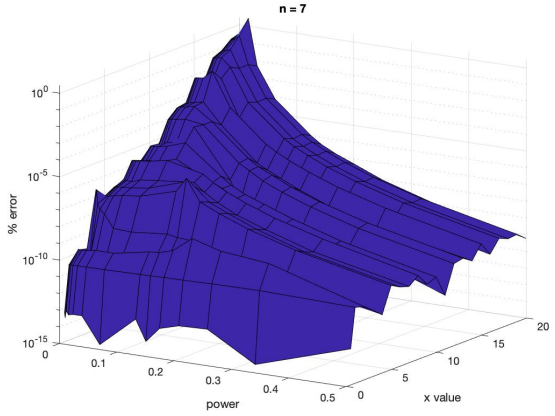


Figure 29: Variation of Error in a range of power and x value in $f(x)$ for $\mathbb{C}_7$

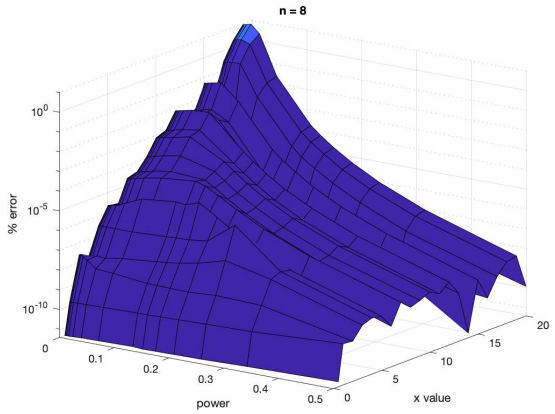


Figure 30: Variation of Error in a range of power and x value in $f(x)$ for $\mathbb{C}_8$

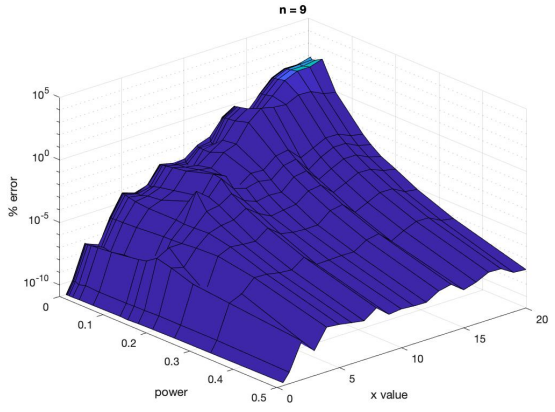


Figure 31: Variation of Error in a range of power and x value in $f(x)$ for $\mathbb{C}_9$

G Integrator the Relative Motion Problem

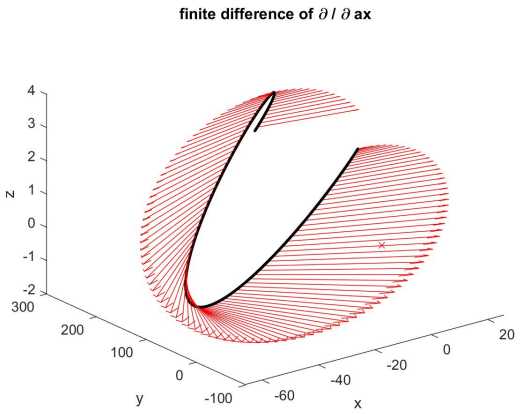
```
function [x,y,z] = newmarkbeta( t, dt, X0, Fx, Fy, Fz, h)
% X0 = initial_state(x,y,z,vx,vy,vz) if use multicomplex step method
% X0 = normal array [x,y,z,vx,vy,vz] if use finite difference
load('earth.mat')
a = 0.5;
b = 0.25;
if isa(X0,'cell') == 1
    x(1) = X0{1,1}; %%change them to complex number
    y(1) = X0{2,1};
    z(1) = X0{3,1};
    vx(1) = X0{4,1};
    vy(1) = X0{5,1};
    vz(1) = X0{6,1};
    ax(1)= multicomplex([0,h]);
    ay(1)= multicomplex([0,0,h,0]);
    az(1)= multicomplex([0,0,0,0,h,0,0,0]);

    for i=1:round(t/dt)
        ax(i+1)= 3*(w^2)*x(i) + 2*w*vy(i) + Fx;
        ay(i+1)= -2*w*vx(i) + Fy;
        az(i+1)= -(w^2)*z(i) + Fz;

        vx(i+1)=vx(i)+(1-a)*dt*ax(i)+((a*dt)*ax(i+1));
        vy(i+1)=vy(i)+(1-a)*dt*ay(i)+((a*dt)*ay(i+1));
        vz(i+1)=vz(i)+(1-a)*dt*az(i)+((a*dt)*az(i+1));

        x(i+1)=x(i)+(dt*vx(i))+((1-2*b)*dt*dt/2)*ax(i)+((b*dt*dt)*ax(i+1));
        y(i+1)=y(i)+(dt*vy(i))+((1-2*b)*dt*dt/2)*ay(i)+((b*dt*dt)*ay(i+1));
        z(i+1)=z(i)+(dt*vz(i))+((1-2*b)*dt*dt/2)*az(i)+((b*dt*dt)*az(i+1));

    end

else
    x(1) = X0(1);
    y(1) = X0(2);
    z(1) = X0(3);
    vx(1) = X0(4);
    vy(1) = X0(5);
    vz(1) = X0(6);
    ax(1)= 0;
    ay(1)= 0;
    az(1)= 0;
```

```
43
44      for i=1:round(t/dt)
45          ax(i+1)= 3*(w^2)*x(i) + 2*w*vy(i) + Fx ;
46          ay(i+1)= -2*w*vx(i) + Fy  ;
47          az(i+1)= -(w^2)*z(i) + Fz;
48
49          vx(i+1)=vx(i)+(1-a)*dt*ax(i)+((a*dt)*ax(i+1));
50          vy(i+1)=vy(i)+(1-a)*dt*ay(i)+((a*dt)*ay(i+1));
51          vz(i+1)=vz(i)+(1-a)*dt*az(i)+((a*dt)*az(i+1));
52
53          x(i+1)=x(i)+(dt*vx(i))+((1-2*b)*dt*dt/2)*ax(i)+((b*dt
               *dt)*ax(i+1));
54          y(i+1)=y(i)+(dt*vy(i))+((1-2*b)*dt*dt/2)*ay(i)+((b*dt
               *dt)*ay(i+1));
55          z(i+1)=z(i)+(dt*vz(i))+((1-2*b)*dt*dt/2)*az(i)+((b*dt
               *dt)*az(i+1));
56
57      end
58
59  end
60  end
```

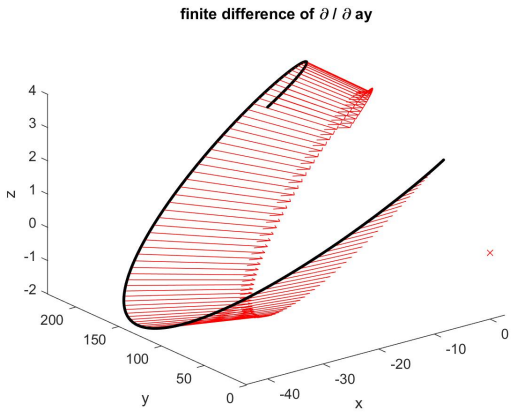
H Finite difference method for Sensitivity Analysis

```
1   function [x2,y2,z2,dx,dy,dz] = finitediff( t, dt, X0, Fx, Fy,
        Fz,h)
2   load('earth.mat')
3
4   [x1,y1,z1] = newmarkbeta( t, dt, X0, Fx, Fy, Fz, h);
5   [x2,y2,z2] = newmarkbeta( t, dt, X0, Fx, Fy, Fz, 0);
6
7   for i=1:round(t/dt)
8   dx(i) = (x1(i)-x2(i))/h;
9   dy(i) = (y1(i)-y2(i))/h;
10  dz(i) = (z1(i)-z2(i))/h;
11  end
12
13  end
```

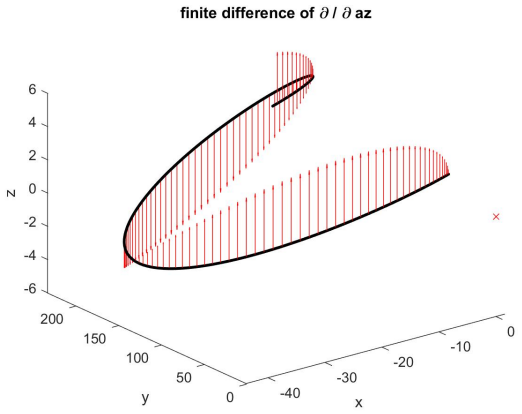
I Sensitivity plot in each directions acceleration perturbation by finite difference



(a) Sensitivity in position with each x direction acceleration perturbation



(b) Sensitivity in position with each y direction acceleration perturbation



(c) Sensitivity in position with each z direction acceleration perturbation

Figure 32: Sensitivity in position with each directions acceleration perturbation by finite difference