# Computation of higher-order derivatives using the multi-complex step method

Adriaen Verheyleweghen, *(verheyle@stud.ntnu.no)*

December 6, 2014

# Contents

# 1 Introduction

This report is the result of the project in the course 'Advanced Simulation'. It investigates the possibility of calculating derivatives of functions using complex differentiation. The report will give a brief introduction to the theoretical background before presenting the algorithm. A large part of the report consists of examples to illustrate the described algorithm and theory. Though it is not necessary to read this part to understand the theory, it is recommended to do so to understand how it works in practice. Finally, a bicomplex class was written in MATLAB and can be found in the appendix.

# 2 Motivation

One of the most commonly used methods to numerically estimate the differential of a continuous function are so-called finite difference methods. These methods are based on a truncated version of the Taylor expansion series around a point. Consider a holomorphic function $f$, i.e. $f$ is infinitely differentiable. The Taylor series for $f$ around the point $x_0$ is then

$$\begin{aligned} f(x) &= f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2}(x - x_0)^2 + \frac{f^{(3)}(x_0)}{6}(x - x_0)^3 + ... && (1) \\ &= \sum_{i=0}^{\infty} \frac{f^{(i)}}{i!}(x - x_0)^i && (2) \end{aligned}$$

Inserting $x = x_0 + h$ yields

$$\begin{aligned} f(x) &= f(x_0) + hf'(x_0) + h^2 \frac{f''(x_0)}{2} + h^3 \frac{f^{(3)}(x_0)}{6} + ... && (3) \\ &= \sum_{i=0}^{\infty} h^i \frac{f^{(i)}}{i!} && (4) \end{aligned}$$

Assuming that $h$ is small such that all higher order terms can be neglected, the equation can be rearranged to yield the forward difference

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0)}{h} + O(h) \tag{5}$$

Where $h$ is the step size and $O$ is the truncation error, which stems from the truncation of the Taylor series after the first term. The accuracy of the method can be improved by choosing a central difference scheme.

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0 - h)}{2h} + O(h^2) \tag{6}$$

It seems like an arbitrary high accuracy can be achieved by choosing a sufficiently small $h$. Indeed, the definition of the derivative is closely related to the forward difference. Letting $h$ go to zero, one obtains the definition of the derivative

$$f'(x) \hat{=} \lim_{x \to 0} \frac{f(x + h) - f(x)}{h} \tag{7}$$

However, rounding error is introduced when using floating point arithmetic. In order to understand why this occurs, one must understand how numbers are stored in computers. In most computers today, numbers are stored as double-precision floating-point variables, meaning that each number is represented by 8 bytes or 64 bits of memory. Out of those 64 bits, 1 is used to store the sign (+ or -), 11 are used to store the exponent and the remaining 52 give the significand precision. Due to special encoding, one additional bit is available for the significand precision. 53 bits of storage in binary equal to $53 \log_{10}(2) \approx 15.96$ unique characters in decimal.

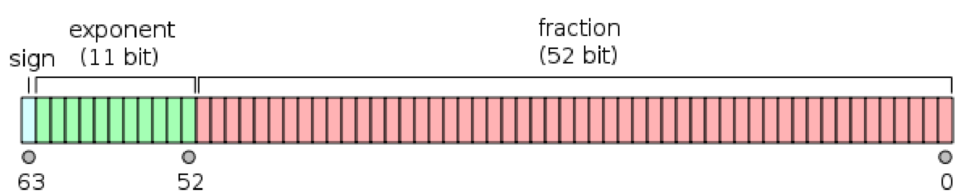


Figure 1: Storage of numbers as 64-bit floating-point variables. Illustration taken from Wikipedia Commons: `http://upload.wikimedia.org/wikipedia/commons/a/a9/IEEE_754_Double_Floating_Point_Format.svg`

In other words, two numbers which are identical in the first 16 significant digits will be stored as the same value inside the computer memory. Subsequent subtraction will yield zero as the result. The smallest distinguishable difference between two numbers is called the machine precision $\epsilon$, and typically has between 15 and 17 significant digits after the exponent.

This rounding error increases as $h$ approaches $\epsilon$, which is why the selection of the step size is subject to limitations when using finite difference methods to estimate differentials. The finite difference methods are said to be ill-conditioned.

This means that a balance between the rounding error and the truncation error has to be found in order to get the highest accuracy. The optimal value for $h$ depends on the nature of $f$, but in general a value close to $h = x \cdot \sqrt{\epsilon}$ gives a decent initial estimate.

## 2.1 Alternatives to finite difference methods

Finite difference methods are often used because they are relatively easy to implement. But due to their ill-conditioned nature, alternative methods have to be used when high precision is desired.

One common approach is to use automatic differentiation (AD). AD exploits the fact that all functions can be expressed as a combination of basic mathematical operations such as subtraction, addition, multiplication etc. By storing the derivative of each variable alongside its value, one can automatically calculate the derivatives of each following variable by repeatedly applying the chain rule to these basic mathematical operations. The disadvantage of this method is that is computationally intensive in terms of computation speed and storage. Each additional higher-order derivative requires the computation and storage of an additional value. Many AD tools are limited to first or second-order derivatives.

Another approach is to calculate the exact derivatives using symbolic calculations. Symbolic differentiation is very similar to manual differentiation, using a set of rules to obtain the derivative. Mathematica and Maple are examples of commercial software which use symbolic mathematics. The major drawback of this method is the high computation cost.

Complex differentiation is another alternative. It is not widely used due to various reasons, mainly the availability of good AD tools. However, some authors claim that complex differentiation has several advantages over AD [2]. The Cauchy integral method is already used to calculate higher-order derivatives in some communities. The Cauchy integral method will be briefly introduced in Section 3.2. The main focus of this report is the (multi)complex step method.

# 3 Complex differentiation

## 3.1 First order derivatives

Squire and Trapp first described an alternative method for calculating first-order derivatives without round-off error in 1998[6]. Their method is reminiscent of the the finite different method, but is extended to the complex plane. By stepping in the complex plane instead of in the real plane, round-off error can be eliminated.

Assume that $f$ is a holomorphic function, i.e. is infinitely differentiable. The Taylor series of $f$ evaluated at the complex point $x_0 + ih$ is then:

$$f(x_0 + ih) = f(x_0) + ihf'(x_0) - \frac{h^2}{2}f''(x_0) - \frac{ih^3}{6}f^{(3)}(x_0) + \frac{h^4}{24}f^{(4)} + ... \tag{8}$$

The imaginary part is

$$\Im(f(x_0 + ih)) = hf'(x_0) - \frac{h^3}{6}f^{(3)}(x_0) + ... \tag{9}$$

Assuming that $h$ is small, the series can be truncated after the first term, yielding the following expression for the first order derivative

$$f'(x_0) \approx \frac{\Im(f(x_0 + ih))}{h} \tag{10}$$

Since the above expression does not contain a subtraction, the rounding error is eliminated. The algorithm is thus well-behaved.

Note that this method only can be used to calculate the first derivative of a function. Trying to solve for the second or third derivative of $f$ will not give any improvement in accuracy because the expression contains a difference term, resulting in rounding error.

For examples on how to apply this method, see Section A.1, Section A.2 and Section A.3 in Appendix A.

## 3.2 n-th order derivatives using Cauchy's integral theorem

Lyness and Moler first used complex numbers to calculate approximations of higher-order differentials of functions in 1967[3].

Cauchy's integral formula states that

$$\oint_C \frac{f(z)}{z-z_0} dz = 2\pi i f(z_0) \tag{11}$$

The proof is straightforward

*Proof.* Let $z = z_0 + \epsilon e^{it}, 0 \leq t \leq 2\pi$ and $\epsilon$ is the radius of the circle. Then

$$\begin{aligned}
\frac{1}{2\pi} \oint_C \frac{f(z)}{z-z_0} dz - f(z_0) &= \frac{1}{2\pi} \oint_C \frac{f(z)}{z-z_0} dz - f(z_0) \frac{1}{2\pi} \oint_C \frac{1}{z-z_0} dz \\
&= \frac{1}{2\pi i} \oint_C \frac{f(z)-f(z_0)}{z-z_0} dz \\
&= \frac{1}{2\pi i} \int_0^{2\pi} \left( \frac{f(z(t))-f(z_0)}{\epsilon e^{it}} \epsilon e^{it} i \right) dt \\
&\leq \frac{1}{2\pi} \int_0^{2\pi} \left( \frac{|f(z(t))-f(z_0)|}{\epsilon} \epsilon \right) dt = \frac{1}{2\pi} \int_0^{2\pi} f'(z) \lim_{\epsilon \to 0} \epsilon dt \\
&\leq \max_{|z-z_0|=\epsilon} |f(z)-f(z_0)| \to 0 \text{ as } \epsilon \to 0
\end{aligned}$$

□

The last inequality results from the estimation lemma. Furthermore, it was used that

$$\oint_C \frac{1}{z-z_0} dz = \oint \frac{1}{\epsilon} e^{-it} \cdot i\epsilon e^{it} dt = \oint_C i dt = 2\pi i$$

Assuming that $f$ is analytic on a domain $D$ containing the closed curve $C$, then it can be shown that all the derivatives of $f$ can be calculated as

$$f^n(z_0) = \frac{n!}{2\pi i} \oint_C \frac{f(z)}{(z-z_0)^{n+1}} \tag{12}$$

Using the trapezoidal rule, the contour integral can be approximated as [4]

$$f^n(z_0) \approx \frac{n!}{m\epsilon} \sum_{j=0}^{m-1} \frac{f\left(z_0 + \epsilon e^{i\frac{2\pi j}{m}}\right)}{e^{i\frac{2\pi j n}{m}}} \tag{13}$$

This method also contains a subtraction of two equally sized terms, which might lead to rounding error when $h$ becomes too small. But unlike the finite difference method, $h$ is not the only parameter that can be adjusted in order to obtain higher accuracy. By selection a larger $m$, i.e. by using more points to approximate the contour integral, a higher accuracy can be achieved as well.

# 4 Multi-complex step differentiation

The disadvantage of using Cauchy's integral theorem for calculating higher order differentials is the large number of function evaluations required to obtain high accuracy when calculating the contour integral. Even then, the method is somewhat prone to rounding error due to the summation term in Equation 13

An alternative method for calculating higher-order derivatives can be found by extending Squire and Trapp's complex step method into the multicomplex domain. The multicomplex domain contains more than one complex plane, as the name suggests. Since one complex dimension is often sufficient to solve most problems, little attention has been paid to multicomplex mathematics. Price did substantial work on this field in the 70's, and his work will be used as a foundation to derive the multicomplex step method described in the next sections.

## 4.1 Definition of multicomplex numbers

Consider a multicomplex number $\zeta_n$. The set of multicomplex numbers of order $n$ is [5]

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

Each complex space can be defined in terms of variables from the underlying complex space. For example, the bicomplex space is defined as

$$\mathbb{C}_2 = \{\zeta_2 = z_1 + z_2 \cdot i_2 : z_1, z_2 \in \mathbb{C}_1\} \tag{15}$$

Finally, the "monocomplex" space is defined as

$$\mathbb{C}_1 = \{z = x_1 + x_2 \cdot i_1 : x_1, x_2 \in \mathbb{C}_0 = \mathbb{R}\} \tag{16}$$

From insertion it follows that $\mathbb{C}^n$ also can be defined as

$$\begin{aligned}\mathbb{C}_n = &\{\zeta_n = \zeta_{n-2,1} + \zeta_{n-2,2} \cdot i_{n-1} + \zeta_{n-2,3} \cdot i_n + \zeta_{n-2,4} \cdot i_{n-1} \cdot i_n : \\ &\zeta_{n-2,1}, \zeta_{n-2,2}, \zeta_{n-2,3}, \zeta_{n-2,4} \in \mathbb{C}_{n-2}\}\end{aligned} \tag{17}$$

Through recursive insertion, it follows that every multicomplex number in $\mathbb{C}^n$ can be represented by $2^n$ parameters in $\mathbb{R}$

Basic mathematical operations in $\mathbb{C}^n$ are similar to operations in $\mathbb{C}^1$. Mathematical operations in $\mathbb{C}^2$ and $\mathbb{C}^3$ are explained in great detail in Price's book [5]. Generalizations in $\mathbb{C}^n$ are also included.

### 4.1.1 Matrix representation of multicomplex numbers

A useful tool for doing basic mathematical operations with complex numbers is the so-called matrix representation of complex numbers. The monocomplex number $z$ can be represented by its matrix $\mathbf{Z}$

$$\mathbf{Z} = \begin{bmatrix} x_1 & -x_2 \\ x_2 & x_1 \end{bmatrix} \tag{18}$$

Basic mathematical operations such as addition, subtraction, multiplication and division of complex numbers are easily performed on their matrix representations. For example, multiplication of two complex numbers can be done as

$$\begin{aligned}\mathbf{Z}_c = \mathbf{Z}_a \cdot \mathbf{Z}_b \quad &= \begin{bmatrix} x_{a,1} & -x_{a,2} \\ x_{a,2} & x_{a,1} \end{bmatrix} \cdot \begin{bmatrix} x_{b,1} & -x_{b,2} \\ x_{a,2} & x_{b,1} \end{bmatrix} && (19) \\ &= \begin{bmatrix} x_{a,1}x_{b,1} - x_{a,2}x_{b,2} & -(x_{a,1}x_{b,2} - x_{a,2}x_{b,1}) \\ x_{a,1}x_{b,2} - x_{a,2}x_{b,1} & x_{a,1}x_{b,1} - x_{a,2}x_{b,2} \end{bmatrix} && (20) \\ &= \begin{bmatrix} x_{c,1} & -x_{c,2} \\ x_{c,2} & x_{c,1} \end{bmatrix} && (21)\end{aligned}$$

One necessary property of the matrix representation is that the square of the complex unit vector equals to one.

$$\mathbf{E}^2 = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix} \tag{22}$$

These matrices can also be used for addition, subtraction and division.

The equivalent of these complex matrix represenations can be defined for multicomplex numbers as well. Using the introduced notation, the matrix representation of the multicomplex number $\zeta_n$ can be written as

$$\mathbf{Z}_n = \begin{bmatrix} \zeta_{n-1,1} & -\zeta_{n-1,2} \\ \zeta_{n-1,2} & \zeta_{n-1,1} \end{bmatrix} \tag{23}$$

Due to the recursive nature of multicomplex numbers, one can use block matrices to represent higher dimensional multicomplex numbers as matrices of multicomplex number of lower dimensionality. For example, the matrix representation of the bicomplex number $\zeta_2$ can be written as

$$\mathbf{Z}_2 = \begin{bmatrix} z_1 & -z_2 \\ z_2 & z_1 \end{bmatrix} = \begin{bmatrix} x_1 & -x_2 & -x_3 & x_4 \\ x_2 & x_1 & -x_4 & -x_3 \\ x_3 & -x_4 & x_1 & -x_2 \\ x_4 & x_3 & x_2 & x_1 \end{bmatrix} \tag{24}$$

As can be seen, the matrix representation of an n-dimensional multicomplex number has is of size $2^n \times 2^n$ when written in terms of parameters in $\mathbb{R}$.

#### 4.1.2 Some mathematical functions in $\mathbb{C}^n$

In addition to the basic mathematical operators, functions of multicomplex numbers must also be defined. Just like functions of complex numbers can be rewritten in terms of their real and imaginary parts, multicomplex numbers must be rewritten in terms of their real and imaginary parts. Let us for example take the cosine function. It can be shown that

$$\cos(z) = \cos(x_1)\cosh(x_2) - i\sin(x_1)\sinh(x_2) \tag{25}$$

A very similar relationship is valid for numbers in higher complex dimensions

$$\cos(\zeta_n) = \cos(\zeta_{n-1,1})\cosh(\zeta_{n-1,2}) - i_n\sin(\zeta_{n-1,1})\sinh(\zeta_{n-1,2}) \tag{26}$$

Most functions are easily adapted to take multicomplex arguments. Problems arise when treating inverse functions, however. Due to their non-injective nature, it is difficult to define them in an unambiguous way. The inverse trigonometric functions, for example, have multiple solutions depending on the quadrant of the complex input. The situation is even worse for bicomplex numbers, in which case there will be different solutions depending on which octant the input is in. Caution must be taken when trying to implement these functions.

## 4.2 Complex step differentiation algorithm

With the definition of multicomplex numbers, one can now easily extend Squire & Trapp's complex step method to the multicomplex domain in order to calculate higher order derivatives. This idea was first explored by Lantoine et al. in 2012 [2].

Assume that $f$ is a holomorphic function, i.e. is infinitely differentiable. The Taylor series of $f$ around the real point $x_0$ can be written as:

$$\begin{aligned} f(x_0 + i_1 h + ... + i_n h) &= f(x_0) + (i_1 h + ... + i_n h) f'(x_0) + \frac{(i_1 h + ... + i_n h)^2}{2} f''(x_0) + ... && (27) \\ &= \sum_{k=0}^{\infty} \left( \left( \sum_{l=1}^{n} i_l \cdot h \right)^k \frac{f^{(k)}}{k!} \right) && (28) \end{aligned}$$

The term $\left(\sum_{l=1}^{n} i_l \cdot h\right)^k$ can be expanded using the multinomial theorem, which states that

$$\left( \sum_{i=1}^{m} i x_i \right)^n = \sum_{k_1 + k_2 + ... + k_m = n} \binom{n}{k_1, k_2, ..., k_m} \prod_{1 \leq i \leq m} x_i^{k_i} \tag{29}$$

where the binomial coefficient is defined as

$$\binom{n}{k_1, k_2, ..., k_m} = \frac{n!}{k_1! k_2! ... k_m!} \tag{30}$$

The n-th order derivative is the only derivative containing the unique term $h^n \left(\prod_l^n i_l\right)$, as can be seen from the multinomial theorem. In that case $k_1 = k_2 = ... = k_n = 1$. Let us define the function $\Im_{1...n}$ which retrieves the part of the multicomplex number corresponding to $x_{2^n} \in \mathbb{R}$.

$$x_{2^n} = \Im_1 \left( \Im_2 \left( ... \left( \Im_n \left( \zeta_n \right) \right) ... \right) \right) = \Im_{1...n} \left( \zeta_n \right) \tag{31}$$

It follows that the n-order derivative can be calculated as

$$f^{(n)}(x_0) = \frac{\Im_{1...n} \left( f \left( x_0 + \sum_{k=1}^{n} h \cdot i_k \right) \right)}{h^n} \tag{32}$$

which is very similar to the first order approximation derived by Squire & Trapp. By making $h$ sufficiently small, any accuracy can be obtained, down to machine precision.

# 5 Implementation of bicomplex numbers in MATLAB

The multicomplex step differentiation method is easily implemented in MATLAB once a multicomplex class has been constructed. Due to the recursive nature of multicomplex numbers, it is in theory only necessary to construct the class once. All subsequent classes can be defined recursively from the first class, inheriting all its methods. A bicomplex class was therefore written in MATLAB as a proof of concept. The script that implements the bicomplex class is attached in Appendix B.1.

In theory it should have been possible to inherit the methods and properties of the built-in complex class in MATLAB, but since built-in classes are not available for reading or writing, the author was not able to do this. Instead, all necessary methods were defined manually. This includes common operations such as initiation, indexing and concatenation, but also mathematical operations such as addition, subtraction, multiplication and division.

Functions such as the trigonometric and the exponential functions were overloaded manually using similar definitions as for monocomplex numbers. It was first attempted to overload all functions automatically by looping through all functions contained in the symbolic toolbox. The functions were split up into one real and three complex parts (corresponding to $i_1$, $i_2$ and $i_1 i_2$) through two sets of substitution. But this method was not well suited due to the symbolic toolbox struggling to split some functions into their real and imaginary parts, resulting in calls to the 'imag' and 'real'-function in the final expressions. The symbolic toolbox also failed to overload inverse functions such as arcsin and arccos, since it chose one particular solution resulting in $\Im_1 = \Im_{12} = 0$ for all bicomplex numbers.

The bicomplex class seems to work fine for the mentioned simple functions, but inverse functions have not yet been implemented due to the authors lacking mathematical knowledge. However, the framework is built and it should be relatively easy to implement the missing functions in the future.

The bicomplex class was written in such a way that it can easily be adopted for tricomplex or multicomplex numbers.

# 6 Comparison to other differentation methods

## 6.1 Automatic differentiation

It was difficult to find an AD package for MATLAB which is easy to use, so it was unfortunately not possible to compare the speed of the bicomplex differentation method with the speed of AD. It is expected that multicomplex differentiation and AD have similar performances since both techniques are based on breaking down the code to elementary operations. The idea behind AD is to apply the chain rule to each elementary operation in the code, whereas the idea in complex differentiation is to treat all variables as complex variables and perform elementary operations on them. This means that for both techniques, the computation time is expected to be related to the complexity of the differentiated function. For very large systems, the chain rule becomes increasingly computationally intensive. The same is true for complex differentiation. Consider the multiplication of two multicomplex numbers, for example. Since multiplication is done on the matrix representations of the numbers, the computation time scales quadratically with the size of the system.

The memory cost of multicomplex differentiation is comparable to the memory cost of AD for first order derivatives, but becomes increasingly larger for higher order derivatives. This is because each variable is associated with $n$ values in AD, whereas each variable is associated with $n^2$ values in multicomplex differentiation. In the case of bicomplex numbers and second order derivatives, multicomplex differentiation is twice as memory intensive as AD.

According to Lantoine, his MultiComplex Step method outperformed AD02, which is a AD method written for Fortran 90 [2]. Lantoine's MultiComplex Step method was outperformed by TAPENADE, but unlike AD02 it transforms the source program and is limited to first-order derivatives. Judging by Lantoine's results, it could seem that multicomplex differentiation methods on average perform on a par with AD methods. However, Lantoine also states that his results should only be used as an indication, as the performance is varying from problem to problem.

## 6.2 Symbolic differentiation

Symbolic differentiation is known to be relatively slow and memory intensive. Expressions for the derivative are known to grow exponentially, which can lead to problems in the execution of the code [1].

It was attempted to write a script that tests how the computation time of a problem increases with increasing complexity. The following equation was evaluated for a range of $x$.

$$f(x) = \frac{x^4 \sin(x)}{x + e^x}$$

$x$ is a square matrix with random values. The size of $x$ increases with each iteration. Figure 6.2 shows the obtained results.

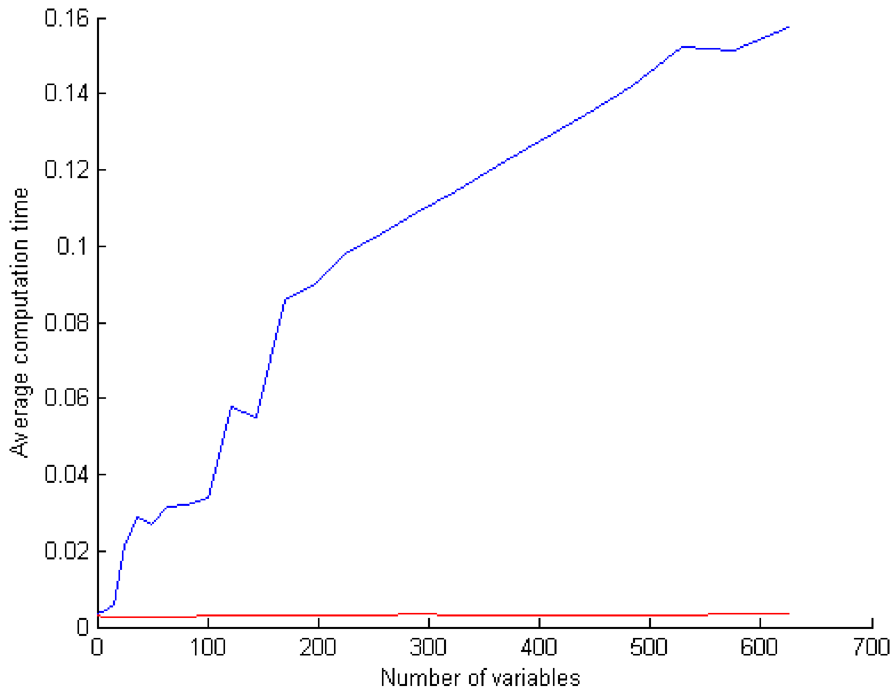


Figure 2: Computation time for calculating the first order derivative as a function of the number of variables. Symbolic differentiation in blue, multicomplex differentiation in red.

As can be seen from the figure, it seems as if the multicomplex differentiation method is much better suited for calculating the derivative of large systems, such as 25x25 matrices. Both methods seem to increase linearly with increasing number of variables. It could look like multicomplex differentiation is independent of input size, but this is not true. Figure B.1 shows the average computation time for the multicomplex differentiation method.

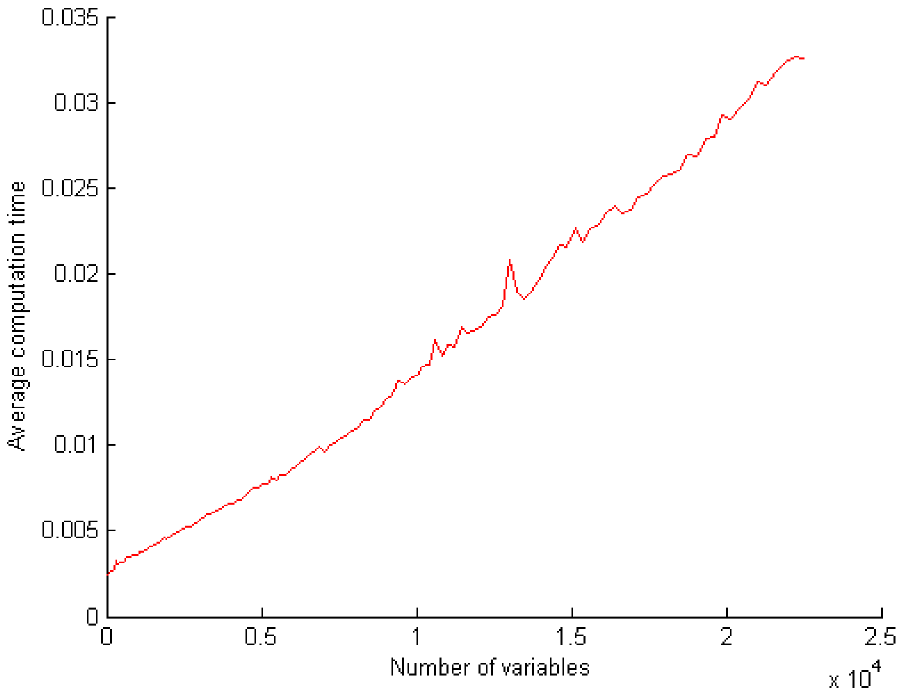


Figure 3: Computation time for calculating the first order derivative as a function of the number of variables using the multicomplex differentiation method.

As can be seen from the figure, the method still works exceptionally well for 150x150 systems with close to 25000 variables. It was attempted to compute the derivative of the same system using the symbolic toolbox, but the attempt was terminated after several seconds without a result.

The good performance of the multicomplex method can be attributed to the fact that MATLAB is optimized to perform large matrix calculations. Since the multicomplex method consists of elementary operations on matrix representations, it will be very fast. The script that was used to obtain the above figures is attached in Appendix B.2

However, the huge performance difference could also be due to implementation errors or other factors that were not considered here. One should therefore take the results with a pinch of salt.

## 6.3 Why is multicomplex differentiation not widely used?

The results from the previous sections indicate that multicomplex differentiation is a viable alternative to the most commonly used differentiation methods. Some possible reasons as to why multicomplex differentiation is not widely used include:

- Multicomplex numbers remain uncharted territory in mathematics, and only a few publications exist on the subject. Lantoine's paper on multicomplex differentiation was published in 2012, which is several decades later than when AD was first introduced.
- Automatic differentiation is based on a very simple principle and is easy to implement. A lot of research has been done to develop AD software and optimize it.

# 7 Conclusion and suggestions for future work

Multicomplex step differentiation is a good alternative to other differentiation methods if high precision is desired and small step sizes are necessary. Implementation of multicomplex numbers is relatively easy in MATLAB, though inverse functions still pose some problems.

The performance seems to be satisfactory. According to literature, multicomplex differentiation is a viable alternative to automatic differentiation. Results from tests give reason to believe that multicomplex differentiation outperforms MATLAB's built-in symbolic differentiation function from the Symbolic Toolbox.

Suggestions for future work:

- Implement the missing functions, including the inverse functions.
- Generalize the class such that it works for higher-dimensional complex numbers.
- Do an in-debth comparison of advantages and disadvantages of the most commonly used differentiation methods, including variations of finite difference methods, AD, symbolic differentiation and (multi)complex differentiation.

# References

[1] Dan Kalman. Doubly recursive multivariate automatic differentiation. *Mathematics Magazine*, 75(3):pp. 187–202, 2002.

[2] Gregory Lantoine, Ryan P Russell, and Thierry Dargent. Using multicomplex variables for automatic computation of high-order derivatives. *ACM Transactions on Mathematical Software (TOMS)*, 38(3):16, 2012.

[3] James N Lyness and Cleve B Moler. Numerical differentiation of analytic functions. *SIAM Journal on Numerical Analysis*, 4(2):202–210, 1967.

[4] Joaquim R. R. A. Martins, Ilan M. Kroo, and Juan J. Alonso. An automated method for sensitivity analysis using complex variables. *Proceedings of the 38th AIAA Aerospace Sciences Meeting*, January 2000. AIAA 2000-0689.

[5] Griffith Baley Price. *An Introduction to Multicomplex Spaces and Functions.* Chapman & Hall/CRC Pure and Applied Mathematics. Taylor & Francis, 1990.

[6] William Squire and George Trapp. Using complex variables to estimate derivatives of real functions. *Siam Review*, 40(1):110–112, 1998.

# Appendices

## A Examples

To demonstrate the practical applications of the described complex step differentiation methods, the first and second order derivatives of the two functions $f_1(x) = \frac{1}{x}$ and $f_2(x) = \frac{\sin(x)}{x}$ will be calculated manually in the following sections. It will also be shown how to use the extend the (multi)complex step method to calculate the Jacobian and Hessian matrices.

### A.1 Example 1: First order derivative of $f(x) = \frac{1}{x}$

Consider the function

$$f(x) = \frac{1}{x} \tag{33}$$

The derivative of the function is to be estimated at a point $x_0$ using the method described in Section 3.1. Defining

$$z = x_0 + ih \tag{34}$$

Substituting into Equation 33

$$f(z) = \frac{1}{z} = \frac{1}{x_0 + ih} \tag{35}$$

The function can be split up into its real and imaginary parts by remembering the relationship between the modulus and the complex conjugate

$$z \cdot \bar{z} = |z|^2 \tag{36}$$

where the complex conjugate of $z$ is defined as

$$\bar{z} = x_0 - ih \tag{37}$$

and the modulus of $z$ is defined as

$$|z| = \sqrt{x_0^2 + h^2} \tag{38}$$

Equation 35 can thus be written as

$$\frac{1}{z} = \frac{x_0 - ih}{x_0^2 + h^2} \tag{39}$$

Following the method from Section 3.1, the first order derivative can now be calculated as

$$f'(x_0) \approx \frac{\Im(f(x_0 + ih))}{h} = \frac{-1}{x_0^2 + h^2} \tag{40}$$

It can be seen that the expression does not contain any subtractions, and does therefore not suffer from rounding errors. Taking the limit as $h$ goes to zero yields the exact function

$$\lim_{h \to 0} \frac{\Im(f(x_0 + ih))}{h} = -\frac{1}{x_0^2} = f'(x) \tag{41}$$

## A.2 Example 2: First order derivative of $f(x) = \frac{\sin(x)}{x}$

Now consider the function

$$f(x) = \frac{\sin(x)}{x} \tag{42}$$

Substituting $z = x_0 + ih$ into the expression gives

$$f(z) = \frac{\sin(z)}{z} = \frac{\sin(x_0 + ih)}{x_0 + ih} \tag{43}$$

The expression for $f(z)$ can be split into its real and complex parts by remembering that

$$\sin(z) = \sin(x_0 + ih) = \sin(x_0)\cosh(h) + i \cdot \sin(x_0)\sinh(h) \tag{44}$$

Such that

$$f(z) = \left(\frac{x_0 - ih}{x_0^2 + h^2}\right) \cdot \big(\sin(x_0)\cosh(h) + i \cdot \sin(x_0)\sinh(h)\big) \tag{45}$$

The first order derivative can be expressed as

$$f'(x_0) \approx \frac{\Im(f(x_0 + ih))}{h} = \frac{x_0\cos(x_0)\frac{\sinh(h)}{h} - h\sin(x_0)\frac{\cosh(h)}{h}}{x_0^2 + h^2} \tag{46}$$

The derivative of $f$ is found by letting the limit of $h$ go to zero

$$f'(x) = \frac{\cos(x)}{x} - \frac{\sin(x)}{x^2} \tag{47}$$

Again, no subtraction of equally sized numbers occurs, eliminating the round-off error.

### A.2.1 Comparison to the finite difference method

The above function was evaluated using MATLAB. The attached script in Appendix B.3 evaluates the derivative of $\sin(x)/x$ at $x_0 = \frac{\pi}{2}$

Evaluated at the point $x_0 = \frac{\pi}{2}$, the exact solution is

$$f'(x_0) = -\frac{4}{\pi^2}$$

Running the attached script, the absolute errors between the exact value and the estimated values are calculated for different step sizes. The resulting graph is shown in Figure A.2.1. Note that the central difference method starts failing at a step size of approximately $h = 10^{-5}$. This value corresponds somewhat well with the rule of thumb saying that $h = x\sqrt{\epsilon} \approx 10^{-8}$ gives the best trade-off between rounding error and truncation error. It can be seen from the figure that values larger than $h = 10^{-5}$ result in increasing rounding error. For very small step sizes close to the machine precision, the method breaks down completely.

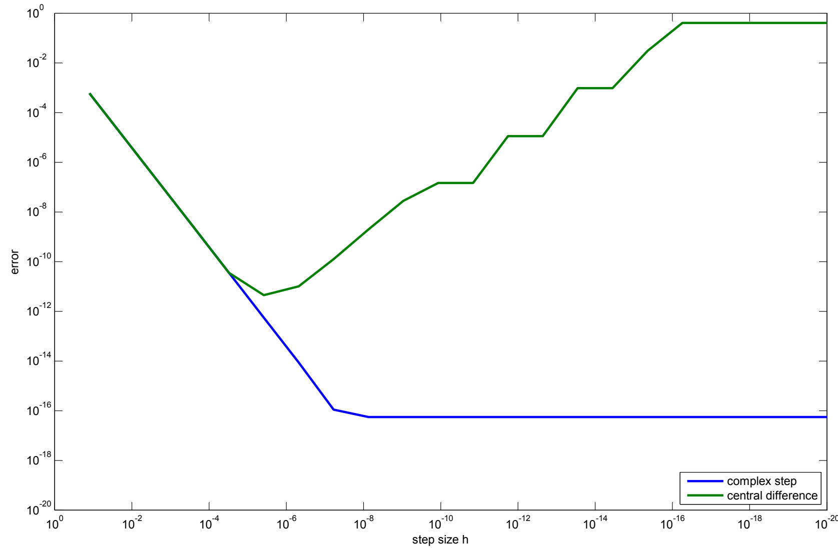


Figure 4: Absolute errors between exact value and estimated value of the first order derivative of $\frac{sin(x)}{x}$ evaluated at $x_0 = \frac{\pi}{2}$

## A.3 Example 3: Using complex step differentiation to find the Jacobian matrix

Given a system of equations $\mathbf{f}(\mathbf{x})$ where $\mathbf{x} = [x_1, x_2, ..., x_n]^T$, then the Jacobian matrix of the system can be defined as

$$J = \begin{pmatrix} \frac{\partial f_1(\mathbf{x})}{\partial x_1} & \frac{\partial f_1(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial f_1(\mathbf{x})}{\partial x_n} \\ \frac{\partial f_2(\mathbf{x})}{\partial x_1} & \frac{\partial f_2(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial f_2(\mathbf{x})}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m(\mathbf{x})}{\partial x_1} & \frac{\partial f_m(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial f_m(\mathbf{x})}{\partial x_n} \end{pmatrix} \tag{48}$$

Using the complex step differentiation method, the Jacobian matrix can be approximated as

$$J \approx \Im \begin{pmatrix} f_1(\mathbf{x} + ih\mathbf{e}_1) & f_1(\mathbf{x} + ih\mathbf{e}_2) & \cdots & f_1(\mathbf{x} + ih\mathbf{e}_n) \\ f_2(\mathbf{x} + ih\mathbf{e}_1) & f_2(\mathbf{x} + ih\mathbf{e}_2) & \cdots & f_2(\mathbf{x} + ih\mathbf{e}_n) \\ \vdots & \vdots & \ddots & \vdots \\ f_m(\mathbf{x} + ih\mathbf{e}_1) & f_m(\mathbf{x} + ih\mathbf{e}_2) & \cdots & f_m(\mathbf{x} + ih\mathbf{e}_n) \end{pmatrix} \frac{1}{h} \tag{49}$$

Where $\mathbf{e}_i$ is defined such that $I = [\mathbf{e}_1, \mathbf{e}_2, ..., \mathbf{e}_n]$

A function is written in MATLAB to calculate the Jacobian matrix for a system of equations. The script is attached in Appendix B.4

### A.4 Example 4: Second order derivative of $f(x) = \frac{1}{x}$

According to the derived rules in Section 4.1, the second order derivative of $f(x) = \frac{1}{x}$ can be calculated as

$$f''(x) \approx \frac{\Im_{12}(f(x + i_1 h + i_2 h))}{h^2} = \frac{\Im_1(\Im_2(f(x + i_1 h + i_2 h)))}{h^2} \tag{50}$$

Where $\Im_{12}$ is defined as

$$\Im_{12}(\zeta_2) = \zeta_{0,4} \tag{51}$$

and

$$\begin{aligned} \zeta_2 = \big(\zeta_{0,1} + \zeta_{0,2} \cdot i_1 + \zeta_{0,3} \cdot i_2 + \zeta_{0,4} \cdot i_1 \cdot i_2\big) : \\ \zeta_{0,1}, \zeta_{0,2}, \zeta_{0,3}, \zeta_{0,4} \in \mathbb{R} \end{aligned} \tag{52}$$

In other words, $\Im_{12}$ is the function which retrieves the term associated with both $i_1$ and $i_2$.

Substituting $x \to \zeta_2 = x + i_1 h + i_2 h$ into the expression gives

$$f(\zeta_2) = \frac{1}{x + i_1 h + i_2 h} \tag{53}$$

$$= \frac{(x + i_1 h) - i_2 h}{(x + i_1 h)^2 + h^2} \tag{54}$$

$$= \frac{(x + i_1 h) - i_2 h}{x^2 + 2 i_1 h x} \tag{55}$$

$$= \frac{(x + i_1 h - i_2 h)\left(x^2 - 2 i_1 h x\right)}{x^4 + 4h^2 x^2} \tag{56}$$

$$= \frac{x^3 + 2xh^2}{x^4 + 4h^2 x^2} + \frac{-x^2 h}{x^4 + 4h^2 x^2} i_1 + \frac{-x^2 h}{x^4 + 4h^2 x^2} i_2 + \frac{2xh^2}{x^4 + 4h^2 x^2} i_1 i_2 \tag{57}$$

It was used that

$$\zeta \bar{\zeta} = |\zeta|^2 \to \frac{1}{\zeta} = \frac{\bar{\zeta}}{|\zeta|^2} \tag{58}$$

Comparison of Equation 71 with Equation 52 gives

$$f_{0,1}(x) = \frac{x^3 + 2xh^2}{x^4 + 4h^2 x^2} \tag{59}$$

$$f_{0,2}(x) = -\frac{x^2 h}{x^4 + 4h^2 x^2} \tag{60}$$

$$f_{0,3}(x) = -\frac{x^2 h}{x^4 + 4h^2 x^2} \tag{61}$$

$$f_{0,4}(x) = \frac{2xh^2}{x^4 + 4h^2 x^2} \tag{62}$$

$$\tag{63}$$

With $f_{0,i}(x)$ being related to $f_2(\zeta_2)$ in a similar way to how $\zeta_{0,i}$ is related to $\zeta_2$, namely being the part of the function which gives the corresponding imaginary term.

The derivative of $f(x) = \frac{1}{x}$ can now be calculated from Equation 50

$$f''(x) \approx \frac{\Im_{12}(f(x_0 + i_1 h + i_2 h))}{h^2} = \frac{f_{0,4}}{h^2} = \frac{2x}{x^4 + 4h^2 x^2} \tag{64}$$

Taking the limit as $h$ goes to zero gives the exact solution

$$f''(x) = \lim_{h \to 0} \left( \frac{2x}{x^4 + 4h^2 x^2} \right) = \frac{2}{x^3} \tag{65}$$

Alternatively, one can use the definition on the right hand side of Equation 50 to calculate the derivative.

$$\Im_2(f(\zeta_2)) = \Im_2\left(\frac{(x+i_1h)-i_2h}{x^2+2i_1hx}\right) = \frac{-h}{x^2+2i_1hx} \tag{66}$$

$$= \frac{-h(x^2-2i_1hx)}{x^4+4h^2x^2} \tag{67}$$

$$f''(x) \approx \frac{\Im_1(\Im_2(f(x_0+i_1h+i_2h)))}{h^2} = \frac{2x}{x^4+4h^2x^2} \tag{68}$$

The two methods are equivalent, though the first approach might be more efficient if implemented into a computer program. This is because fewer function calls are required (only one call to $\Im_{12}$ instead of two calls to $\Im_1$ and $\Im_2$)

It should also be noted that Equation 71 contains terms associated with all the lower order derivatives. In fact, substitution of $x \to \zeta_n$ into a function $f(x)$ will not only provide the $n^{\text{th}}$ derivative, but also all lower order derivatives from $f^{(n-1)}(x)$ to $f^{(1)}(x)$.

## A.5 Example 5: Second order derivative of $f(x) = \frac{\sin(x)}{x}$

Substituting $x \to \zeta_2 = x + i_1 h + i_2 h$ into $f(x) = \frac{\sin(x)}{x}$ gives

$$f(\zeta_2) = \frac{\sin(x + i_1 h + i_2 h)}{x + i_1 h + i_2 h} \tag{69}$$

$$\tag{70}$$

The result from the previous section can be utilized

$$f(\zeta_2) = \left( \frac{x^3 + 2xh^2}{x^4 + 4h^2x^2} + \frac{-x^2h}{x^4 + 4h^2x^2} i_1 + \frac{-x^2h}{x^4 + 4h^2x^2} i_2 + \frac{2xh^2}{x^4 + 4h^2x^2} i_1 i_2 \right) \sin(x + i_1 h + i_2 h) \tag{71}$$

The term $\sin(x + i_1 h + i_2 h)$ can be expanded using the following rules

$$\begin{aligned}
\sin(x_1 + x_2 i) &= \sin(x_1)\cos(x_2 i) + \cos(x_1)\sin(x_2 i) && (72)\\
&= \sin(x_1)\cosh(x_2) + i \cdot \cos(x_1)\sinh(x_2) && (73)\\
\cos(x_1 + x_2 i) &= \cos(x_1)\cos(x_2 i) - \sin(x_1)\sin(x_2 i) && (74)\\
&= \cos(x_1)\cosh(x_2) - i \cdot \sin(x_1)\sinh(x_2) && (75)\\
& && (76)
\end{aligned}$$

The relationships can be derived using basic trigonometric identities and the relationship between the trigonometric functions and the exponential function.

Let $g = \sin(x + i_1 h + i_2 h)$. Then $g$ can be written as

$$\begin{aligned}
g(\zeta_2) &= \sin(x + i_1 h + i_2 h) && (77)\\
&= \sin(x + i_1 h)\cosh(h) + i_2 \cos(x + i_1 h)\sinh(h) && (78)\\
&= \sin(x)\cosh^2(h) + i_1 \cos(x)\sinh(h)\cosh(h) + i_2 \cos(x)\cosh(h)\sinh(h) - i_1 i_2 \cdot \sin(x)\sinh^2(h) && (79)
\end{aligned}$$

$f(\zeta_2)$ can now be rewritten as a sum of the different complex terms

$$\begin{aligned}
f(\zeta_2) = &\left( \frac{2\,h^2\,\sin(x) + x^2\,\cosh(h)^2\,\sin(x) + h\,x\,\sinh(2\,h)\,\cos(x)}{4\,h^2\,x + x^3} \right)\\
&- i_1 \left( \frac{h\,\cosh(2\,h)\,\sin(x) - \frac{x\,\sinh(2\,h)\,\cos(x)}{2}}{4\,h^2 + x^2} \right)\\
&- i_2 \left( \frac{h\,\cosh(2\,h)\,\sin(x) - \frac{x\,\sinh(2\,h)\,\cos(x)}{2}}{4\,h^2 + x^2} \right)\\
&+ i_1 i_2 \left( \frac{2\,h^2\,\sin(x) + \frac{x^2\,\sin(x)}{2} - \frac{x^2\,\cosh(2\,h)\,\sin(x)}{2} - h\,x\,\sinh(2\,h)\,\cos(x)}{4\,h^2\,x + x^3} \right)
\end{aligned} \tag{80}$$

The function is now on the form

$$\begin{gathered}
f(\zeta_2) = f_{0,1}(x) + f_{0,2}(x) i_1 + f_{0,3}(x) i_2 + f_{0,4}(x) i_1 i_2\\
f_{0,i}(x) : \mathbb{R} \to \mathbb{R}
\end{gathered} \tag{81}$$

The second derivative of $f(x) = \frac{\sin(x)}{x}$ can now be approximated as

$$f''(x) \approx \frac{\Im_{12}(f(x + i_1 h + i_2 h))}{h^2} = \frac{f_{0,4}(x)}{h^2} \tag{82}$$

Letting the limit of $h$ go to zero, the exact expression is obtained

$$f''(x) = \lim_{h\to 0} \frac{f_{0,4}(x)}{h^2} = -\frac{x^2\,\sin(x) - 2\,\sin(x) + 2\,x\,\cos(x)}{x^3} \tag{83}$$

As the calculations from this section show, things get out of hands rather quickly, with calculations being difficult to do even for relatively simple functions.
[2]

### A.5.1 Comparison to the finite difference method

The above function was evaluated using MATLAB. The attached script in Appendix B.5 evaluates the derivative of $\sin(x)/x$ at $x_0 = \frac{\pi}{2}$

Evaluated at the point $x_0 = \frac{\pi}{2}$, the exact solution is

$$f''(x_0) = \frac{16}{\pi^3} - \frac{2}{\pi}$$

Running the attached script, the absolute errors between the exact value and the estimated values are calculated for different step sizes. The resulting graph is shown in Figure A.5.1. Note that the central difference method starts failing at a step size of approximately $h = 10^{-3}$. For very small step sizes close to the machine precision, the method breaks down completely and gives oscillatory behaviour. It can also be observed that for relatively large step sizes, it seems as if the central difference method outperforms the multicomplex step method. This could either be an implementation error or be related to the remaining $h$-terms in the expression, which somehow decrease the accuracy.

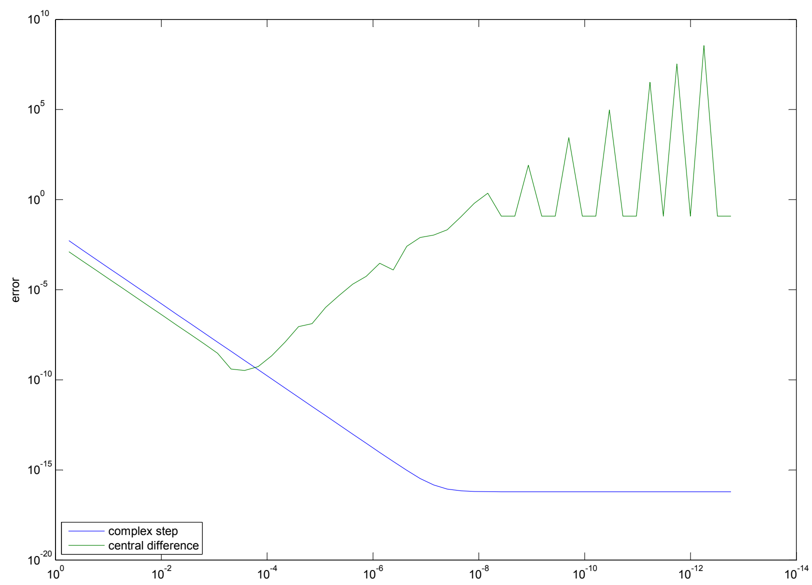


Figure 5: Absolute errors between exact value and estimated value of the second order derivative of $\frac{sin(x)}{x}$ evaluated at $x_0 = \frac{\pi}{2}$

## A.6 Example 6: Using multicomplex step differentiation to calculate the Hessian matrix

Given the multivariable equation $f(\mathbf{x})$ where $\mathbf{x} = [x_1, x_2, ..., x_n]^T$, then the Hessian matrix of $f$ can be defined as

$$H = \begin{pmatrix} \frac{\partial^2 f(\mathbf{x})}{\partial x_1^2} & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_2} & \cdots & \frac{\partial f^2(\mathbf{x})}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_2 \partial x_1^2} & \frac{\partial^2 f(\mathbf{x})}{\partial x_2^2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_n \partial x_1} & \frac{\partial^2 f(\mathbf{x})}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_n^2} \end{pmatrix} \tag{84}$$

Using the multicomplex step differentiation method, the Hessian matrix can be approximated as

$$H \approx \Im_{12} \begin{pmatrix} f(\mathbf{x} + i_1 h\mathbf{e}_1 + i_2 h\mathbf{e}_1) & f(\mathbf{x} + i_1 h\mathbf{e}_2 + i_2 h\mathbf{e}_1) & \cdots & f(\mathbf{x} + i_1 h\mathbf{e}_n + i_2 h\mathbf{e}_1) \\ f(\mathbf{x} + i_1 h\mathbf{e}_1 + i_2 h\mathbf{e}_2) & f(\mathbf{x} + i_1 h\mathbf{e}_2 + i_2 h\mathbf{e}_2) & \cdots & f(\mathbf{x} + i_1 h\mathbf{e}_n + i_2 h\mathbf{e}_2) \\ \vdots & \vdots & \ddots & \vdots \\ f(\mathbf{x} + i_1 h\mathbf{e}_1 + i_2 h\mathbf{e}_n) & f(\mathbf{x} + i_1 h\mathbf{e}_2 + i_2 h\mathbf{e}_n) & \cdots & f(\mathbf{x} + i_1 h\mathbf{e}_n + i_2 h\mathbf{e}_n) \end{pmatrix} \frac{1}{h^2} \tag{85}$$

Where $\mathbf{e}_i$ is defined as the unit vector such that $I = [\mathbf{e}_1, \mathbf{e}_2, ..., \mathbf{e}_n]$

A function is written in MATLAB to calculate the Hessian of a function. The script is attached in Appendix B.6

# B MATLAB scripts

## B.1 Bicomplex class

```
classdef bicomplex
    %% BICOMPLEX(z1,z2)
    % Creates an instance of a bicomplex object.
    % zeta = z1 + j*z2, where z1 and z2 are complex numbers.

   properties
        z1, z2
   end

   methods % Initialization
      function self = bicomplex(z1,z2)
         if nargin ~= 2
             error('Requires exactly 2 inputs')
         end
         if ~isequal(size(z1),size(z2))
            error('Inputs must be equally sized')
         end
            self.z1  = z1;
            self.z2  = z2;
         end
   end

   methods % Basic operators

       function mat = matrep(self)          % Returns matrix representation
            mat = [self.z1,-self.z2;self.z2,self.z1];
       end

       function display(self)
          disp('z1:')
          disp(self.z1)
          disp('z2:')
          disp(self.z2)
       end

       function out = subsref(self,index)                        % Indexing
           if strcmp('()',index.type)
               out = bicomplex([],[]);
               out.z1 = builtin('subsref',self.z1,index);
               out.z2 = builtin('subsref',self.z2,index);
           elseif strcmp('.',index.type)
               out = eval(['self.',index.subs]);
           end
       end

       function out = subsasgn(self,index,value)                 % Asigning
           if strcmp('()',index.type)
               out = bicomplex([],[]);
               out.z1 = builtin('subsasgn',self.z1,index,value);
               out.z2 = builtin('subsasgn',self.z2,index,value);
```

```
            elseif strcmp('.',index.type)
                if ~(strcmp(index.subs,'z1') || strcmp(index.subs,'z2'))
                    error('No such field exists. Use z1 and z2 instead')
                else
                    if strcmp(index.subs,'z1')
                         z_1 = value;
                         z_2 = self.z2;
                    else
                         z_2 = value;
                         z_1 = self.z1;
                    end
                    out = bicomplex(z_1,z_2);
                end

            end
        end

        function out = horzcat(self,varargin)     % Horizontal concatenation
            z_1 = [self.z1];
            z_2 = [self.z2];
            for i = 1:length(varargin)
                 [~,tmp] = isbicomp([],varargin{i});
                 z_1 = [z_1,tmp.z1];
                 z_2 = [z_2,tmp.z2];
            end
            out = bicomplex(z_1,z_2);
        end

        function out = vertcat(self,varargin)       % Vertical concatenation
            z_1 = [self.z1];
            z_2 = [self.z2];
            for i = 1:length(varargin)
                 [~,tmp] = isbicomp([],varargin{i});
                 z_1 = [z_1;tmp.z1];
                 z_2 = [z_2;tmp.z2];
            end
            out = bicomplex(z_1,z_2);
        end

        function out = plus(self,other)                            % Addition
             [self,other] = isbicomp(self,other);
             zeta = matrep(self)+matrep(other);
             out = mat2bicomp(zeta);
        end

        function out = minus(self,other)                        % Subtraction
             [self,other] = isbicomp(self,other);
             zeta = matrep(self)- matrep(other);
             out = mat2bicomp(zeta);
        end

        function out = uplus(self)                               % Unary plus
             out = self;
        end
```

```

        function out = uminus(self)                            % Unary minus
             out = -1*self;
        end

        function out = mtimes(self,other)                  % Multiplication
             [self,other] = isbicomp(self,other);
             if ~prod(size(self)==size(other)) && numel(self) == 1
                 mat = matrep(self.*other);
             elseif ~prod(size(self)==size(other)) && numel(other) == 1
                 mat = matrep(self.*other);
             else
                 mat = matrep(self)*matrep(other);
             end
             out = mat2bicomp(mat);
        end

        function out = times(self,other)        % Elementwise multiplication
             [self,other] = isbicomp(self,other);
             if size(self) == size(other)
                 sizes = size(self);
                 z_1 = zeros(sizes);
                 z_2 = zeros(sizes);
                 for i = 1:prod(sizes)
                     sr.type = '()';
                     sr.subs = {i};
                     tmp = subsref(self,sr)*subsref(other,sr);
                     z_1(i) = tmp.z1;
                     z_2(i) = tmp.z2;
                 end
             elseif numel(self) == 1
                 sizes = size(other);
                 z_1 = zeros(sizes);
                 z_2 = zeros(sizes);
                 for i = 1:prod(sizes)
                     sr.type = '()';
                     sr.subs = {i};
                     tmp = self*subsref(other,sr);
                     z_1(i) = tmp.z1;
                     z_2(i) = tmp.z2;
                 end
             elseif numel(other) == 1
                 sizes = size(self);
                 z_1 = zeros(sizes);
                 z_2 = zeros(sizes);
                 for i = 1:prod(sizes)
                     sr.type = '()';
                     sr.subs = {i};
                     tmp = subsref(self,sr)*other;
                     z_1(i) = tmp.z1;
                     z_2(i) = tmp.z2;
                 end
             else
                 error('Matrix dimensions must agree')
```

```
            end
            out = bicomplex(z_1,z_2);
        end

        function out = mrdivide(self,other)                       % Division
             if numel(other) == 1 && numel(other) ~=numel(self)
                        mat = matrep(self./other);
             else
                 [self,other] = isbicomp(self,other);
                 mat = matrep(self)/matrep(other);
             end
             out = mat2bicomp(mat);
        end

        function out = rdivide(self,other)            % Elementwise division
             [self,other] = isbicomp(self,other);
             if size(self) == size(other)
                 sizes = size(self);
                 z_1 = zeros(sizes);
                 z_2 = zeros(sizes);
                 for i = 1:prod(sizes)

                     sr.type = '()';
                     sr.subs = {i};
                     tmp = subsref(self,sr)/subsref(other,sr);
                     z_1(i) = tmp.z1;
                     z_2(i) = tmp.z2;
                 end
             elseif numel(self) == 1
                 sizes = size(other);
                 z_1 = zeros(sizes);
                 z_2 = zeros(sizes);
                 for i = 1:prod(sizes)
                     sr.type = '()';
                     sr.subs = {i};
                     tmp = self/subsref(other,sr);
                     z_1(i) = tmp.z1;
                     z_2(i) = tmp.z2;
                 end
             elseif numel(other) == 1
                 sizes = size(self);
                 z_1 = zeros(sizes);
                 z_2 = zeros(sizes);
                 for i = 1:prod(sizes)
                     sr.type = '()';
                     sr.subs = {i};
                     tmp = subsref(self,sr)/other;
                     z_1(i) = tmp.z1;
                     z_2(i) = tmp.z2;
                 end
             else
                 error('Matrix dimensions must agree')
             end
             out = bicomplex(z_1,z_2);
```

```
        end

        function out = power(self,other)                 % Elementwise power
            sizes = size(self);
            z_1 = zeros(sizes);
            z_2 = zeros(sizes);

            for i = 1:length(z_1(:))
                sr.type = '()';
                sr.subs = {i};
                r = modc(subsref(self,sr));
                theta = argc(subsref(self,sr));
                z_1(i) = r^other*cos(other*theta);
                z_2(i) = r^other*sin(other*theta);
            end
            out = bicomplex([],[]);
            out.z1 = z_1;
            out.z2 = z_2;

        end

        function out = mpower(self,other)                % Elementwise power
            sizes = size(self);
            z_1 = zeros(sizes);
            z_2 = zeros(sizes);

            for i = 1:length(z_1(:))
                sr.type = '()';
                sr.subs = {i};
                r = modc(subsref(self,sr));
                theta = argc(subsref(self,sr));
                z_1 = r^other*cos(other*theta);
                z_2 = r^other*sin(other*theta);
            end
            out = bicomplex([],[]);
            out.z1 = z_1;
            out.z2 = z_2;

        end

        function dims = size(self)                 % Returning size of array
            dims = size(self.z1);
        end

       function n = numel(self)                % Returning number of elements
            n = numel(self.z1);
        end

        function out = modc(self)                          % Complex modulus
           out = sqrt(self.z1.^2 + self.z2.^2);
        end

        function out = norm(self)                                     % Norm
           out = sqrt(real(self.z1).^2 + real(self.z2).^2 + ...
```

```
                        imag(self.z1).^2 + imag(self.z2).^2);
        end

        function theta = argc(self)                         % Complex argument
            theta = atan2(self);
        end

        function out = lt(self,other)                % Less than, self < other
            out = false;
            if real(self.z1) < real(other.z1)
                out = true;
            end
        end

        function out = gt(self,other)             % Greater than, self > other
            out = false;
            if real(self.z1) > real(other.z1)
                out = true;
            end
        end

        function out = le(self,other)    % Less than or equal, self <= other
            out = false;
            if real(self.z1) <= real(other.z1)
                out = true;
            end
        end

        function out = ge(self,other) % Greater than or equal, self >= other
            out = false;
            if real(self.z1) >= real(other.z1)
                out = true;
            end
        end

        function out = eq(self,other)                % Equality, self == other
            out = false;
            if self.z1 == other.z1 && self.z2 == other.z2
                out = true;
            end
        end

        function out = ne(self,other)               % Not equal, self ~= other
            out = true;
            if self.z1 == other.z1 && self.z2 == other.z2
                out = false;
            end
        end

    end

    methods % Mathematical functions
        %% Exponential function and logarithm
        function out = exp(self)                                  % Exponential
```

```
            out = bicomplex([],[]);
             out.z1=exp(self.z1).*cos(self.z2);
             out.z2=exp(self.z1).*sin(self.z2);
        end

        function out = log(self)                                    % Natural logaritm
            out = bicomplex([],[]);
             out.z1=log(modc(self));
             out.z2=argc(self);
        end

        %% Basic trigonometric functions
        function out = sin(self)                                                 % sin
            out = bicomplex([],[]);
             out.z1=cosh(self.z2).*sin(self.z1);
             out.z2=sinh(self.z2).*cos(self.z1);
        end

        function out = cos(self)                                                 % cos
            out = bicomplex([],[]);
             out.z1=cosh(self.z2).*cos(self.z1);
             out.z2=-sinh(self.z2).*sin(self.z1);
        end

        function out = tan(self)                                                 % tan
             out = sin(self)./cos(self);
        end

        function out = cot(self)                                                 % cot
             out = cos(self)./sin(self);
        end

        function out = sec(self)                                                 % sec
             out = 1./cos(self);
        end

        function out = csc(self)                                                 % csc
             out = 1./sin(self);
        end

        %% Basic hyperbolic functions
        function out = sinh(self)
            out = bicomplex([],[]);
             out.z1=cosh(self.z1).*cos(self.z2);
             out.z2=sinh(self.z1).*sin(self.z2);
        end

        function out = cosh(self)
            out = bicomplex([],[]);
             out.z1=sinh(self.z1).*cos(self.z2);
             out.z2=cosh(self.z1).*sin(self.z2);
        end

        function out = tanh(self)
```

```
            out = sinh(self)./cosh(self);
        end

        function out = coth(self)
            out = cosh(self)./sinh(self);
        end

        function out = sech(self)
            out = 1./cosh(self);
        end

        function out = csch(self)
            out = 1./sinh(self);
        end

        function out = atan2(self)
            sizes = size(self);
            ang = zeros(sizes);

            for i = 1:prod(sizes)
                 sr.type = '()';
                 sr.subs = {i};
                 if real(self.z1(i)) > 0;
                     ang(i) = atan(self.z2(i)./ self.z1(i));
                 elseif real(self.z1(i))<0 && real(self.z2(i))>= 0;
                     ang(i) = atan(self.z2(i)./self.z1(i))+pi;
                 elseif real(self.z1(i))<0 && real(self.z2(i))<0;
                     ang(i) = atan(self.z2(i)./self.z1(i))-pi;
                 elseif real(self.z1(i))==0 && real(self.z2(i))> 0;
                     ang(i) = pi/2;
                 elseif real(self.z1(i))==0 && real(self.z2(i))< 0;
                     ang(i) = -pi/2;
                 else
                     error('atan(0,0) undefined');
                 end
            end
            out = ang;
        end
        function out = sqrt(self)
            out = self.^0.5;
        end
    end

    methods % Functions for returning the imaginary and real parts
        function out = real(self)
            out = real(self.z1);
        end
        function out = imag1(self)
            out = imag(self.z1);
        end
        function out = imag2(self)
            out = real(self.z2);
        end
        function out = imag12(self)
```

```
            out = imag(self.z2);
        end
    end
end

%% Utility functions

function [self,other] = isbicomp(self,other)
% Verifies that self and other are bicomplex, or converts them to bicomplex
% if possible

    if isa(self,'double')
        self = bicomplex(self,zeros(size(self)));
    elseif ~isa(self,'bicomplex')
        error('Self is not of class bicomplex')
    end

    if isa(other,'double')
        other = bicomplex(other,zeros(size(other)));
    elseif ~isa(other,'bicomplex')
        error('Other is not of class bicomplex')
    end

end

function zeta = mat2bicomp(mat)
% Takes the matrix representation and returns the corresponding bicomplex
    sizes = size(mat);
    str1 = '1:sizes(1)/2,1:sizes(2)/2';
    str2 = 'sizes(1)/2+1:end,1:sizes(2)/2';
    for i = 3:length(sizes);
        str1 = [str1 sprintf(',1:sizes(%i)',i)];
        str2 = [str2 sprintf(',1:sizes(%i)',i)];
    end
    str1 = sprintf('mat(%s)',str1);
    str2 = sprintf('mat(%s)',str2);
    z1 = eval(str1);
    z2 = eval(str2);
    zeta = bicomplex(z1,z2);
end
```

## B.2 Testing the performance of bicomplex differentiation

```
m = 25;

syms x
f_symbolic = [x*x*x*x*cos(x)/(x+(exp(x)))];
f_fnhandle = @(x) [x*x*x*x*cos(x)/(x+(exp(x)))];

time_sym = zeros(1,m);
time_bcx = zeros(1,m);

cnt = 1;
for k = 1:10

    h = 0.0001;
    for i = 1:m
        x0 = rand(i);
        tic
        res = imag1(f_fnhandle(bicomplex(x0+ones(size(x0))*h*1i,...
            zeros(size(x0)))))/h;

        time_bcx(i) = (cnt-1)/cnt*time_bcx(i)+toc/cnt;
    end

    for i = 1:m
        x0 = rand(i);
        tic
            res = subs(diff(f_symbolic),x,x0);
        time_sym(i) = (cnt-1)/cnt*time_sym(i)+toc/cnt;
    end

    cnt = cnt + 1;
end

close all
hold on
plot([1:m].^2,time_bcx,'r')
plot([1:m].^2,time_sym,'b')
hold off
```

## B.3 Calculating the first-order derivative of sin(x)/x

```
%% Complex differentiation

% Function to be differentiated at x0
x0 = pi/2;
F = @(x) sin(x)./x;

dF_cmplx = @(x,h) imag1(F(bicomplex(x+1i*h,0)))/h;           % multicomplex
dF_cdiff = @(x,h) (F(x+h) - F(x-h))/(2*h);             % central difference

% Exact solution:
exact_sol = -4/pi^2;

% Calculating the residuals
hs = 2.^(-(1:50)');
errs = zeros(50,2);

for k = 1:50
  errs(k,1) = abs(dF_cmplx(x0,hs(k))-exact_sol);
  errs(k,2) = abs(dF_cdiff(x0,hs(k))-exact_sol);
end

% Plotting the residuals
close all
loglog(hs,errs)
set(gca,'XDir','Reverse')
legend('complex step','central difference','location','southwest')
xlabel('step size h')
ylabel('error')
```

## B.4 Function to calculate the Jacobian matrix

```
function jacobian = bcjacobian(f,x0,h)
    m = length(f);
    n = length(x0);
    jacobian = zeros(m,n);

    for j = 1:n
        for k = 1:m
            ej = eye(n); ej = ej(:,j);
            bicomplex(x0+h*ej*1i,zeros(size(x0)))
            jacobian(j,k) = imag1(f(bicomplex(x0+h*ej*1i,zeros(size(x0)))))/h;
        end
    end
end
```

### B.5 Calculating the second-order derivative of sin(x)/x

```
%% Complex differentiation

% Function to be differentiated at x0
x0 = pi/2;
F = @(x) sin(x)./x;

dF_cmplx = @(x,h) imag12(F(bicomplex(x+i*h,h)))/(h^2);       % multicomplex
dF_cdiff = @(x,h) (F(x+h) - 2*F(x) + F(x-h))/(h^2);% 2nd order central diff

% Exact solution:
exact_sol = 16/pi^3 - 2/pi;

% Calculating the residuals
hs = 2.^(-(1:50)');
errs = zeros(50,2);

for k = 1:50
  errs(k,1) = abs(dF_cmplx(x0,hs(k))-exact_sol);
  errs(k,2) = abs(dF_cdiff(x0,hs(k))-exact_sol);
end

% Plotting the residuals
close all
loglog(hs,errs)
set(gca,'XDir','Reverse')
legend('complex step','central difference','location','southwest')
xlabel('step size h')
ylabel('error')
```

## B.6 Function to calculate the Hessian matrix

```
function hessian = bchessian(f,x0,h)
    n = length(x0);
    hessian = zeros(n,n);

    for j = 1:n
        for k = 1:n
            ej = eye(n); ej = ej(j,:);
            ek = eye(n); ek = ek(k,:);
            hessian(j,k) = imag12(f(bicomplex(x0+h*ek*1i,h*ej)))/h^2;
        end
    end
end
```