ORDER TRUNCATED IMAGINARY ALGEBRA FOR COMPUTATION OF MULTIVARIABLE HIGH-ORDER DERIVATIVES IN FINITE ELEMENT ANALYSIS

by

MAURICIO ARISTIZABAL CANO, M.Sc.

DISSERTATION
Presented to the Department of
Mechanical Engineering of
Universidad EAFIT
in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY IN ENGINEERING

ADVISOR:
Manuel J. Garcia, Ph.D.

Universidad EAFIT
School of Engineering
Department of Mechanical Engineering
November 2020

Copyright © 2020 Mauricio Aristizabal Cano
All rights reserved.

DEDICATION

To my wife and family for their unconditional support all along this project.

ACKNOWLEDGEMENTS

I would like to express my profound gratitude to my advisor, Dr. Manuel J. García, for his support, guidance and encouragement during this doctoral project. I would also like to thank Dr. Harry R. Millwater for the opportunity to work with his team at UTSA, his participation as a member of the commitee and his feedback in all discussions regarding hypercomplex numbers; to the other members of the committee, Dr. Omar D. López and Dr. Juan D. Gómez, for taking the time to read and evaluate this dissertation; and to the fellow graduate students from the Applied Mechanics research group (EAFIT) and the Computational Reliability Laboratory (UTSA).

This doctoral project was possible thanks to the scholarship program No. 647-2014 from Departamento Administrativo de Ciencia, Tecnología e Innovación Colciencias. Also, it received partial support from the Department of Defense through grant No. W911NF-15-1-0456.

November 2020

ORDER TRUNCATED IMAGINARY ALGEBRA FOR COMPUTATION OF MULTIVARIABLE HIGH-ORDER DERIVATIVES IN FINITE ELEMENT ANALYSIS

Mauricio Aristizabal Cano, M.Sc.
Universidad EAFIT,

Supervising Professor: Manuel J. Garcia, Ph.D.

Computation of sensitivities in engineering problems has become a necessity in many areas, including mechanical, chemical and biomedical engineering. Recently, the use of hypercomplex algebras, like multicomplex and multidual numbers, has increased in the context of derivative computation as they are machine error accurate and easy to implement. However, current hypercomplex implementations suffer from a doubling process where the amount data required for a computation doubles each time a new order of derivative is required. In this work, Cayley-Dickson algebras like quaternions and octonions were used as a first step to optimize computation of multiple first order derivatives, in finite element analyses. While significant performance improvements were achieved, steps toward a more efficient algebra were found. In this dissertation a new hypercomplex algebra named Order Truncated Imaginary (OTI) numbers is proposed to tackle the excessive growth of current algebras when computing high-order derivatives of multivariable functions. The algebra truncates imaginary directions whose order exceeds a specified value n. As a consequence, when a function is evaluated using a particular OTI number, the result is another OTI number that contains all derivatives up to order n within its imaginary directions. The OTI Finite Element Method (OTIFEM) is proposed in order to integrate OTI capabilities into Finite Element analysis. This allows computation of high order multivariable derivatives efficiently. The key ingredient to this integration is the developed OTI matrix form, that seamlessly integrate the OTI numbers with standard linear algebra procedures. Computation of up to 30th order derivatives were performed in heat transfer, linear elasticity and fluid dynamics analyses and were used to generate a reduced order model that approximated the finite element solution. Results show that derivatives with respect to shape, material and loading parameters are accurate, and can be computed effortlessly and efficiently; and higher truncation orders increase the region of validity of the reduced models obtained with OTIFEM solutions.

Keywords: finite element method, high order derivatives, hypercomplex algebras, numerical differentiation, order truncated imaginary numbers

TABLE OF CONTENTS

Acknowledgements . . . . **iv**

Abstract . . . . **v**

List of Tables . . . . **xi**

List of Figures . . . . **xiii**

Nomenclature **1**

Chapter 1: Introduction and Problem Statement **3**

1.1 Research objectives 4

1.1.1 Specific Objectives 5

1.2 Outline and Scope of the Thesis 5

Chapter 2: Background **7**

2.1 Analytic methods 8

2.2 Automatic Differentiation 10

2.2.1 Forward mode AD 10

2.2.2 Reverse mode AD 12

2.3 Hypercomplex Algebras 12

2.4 Summary 14

Chapter 3: Quaternion and Octonion-based Finite Element Analysis Methods for Computing Multiple First Order Derivatives **16**

3.1 Motivation 16

3.2 Quaternion Taylor Series Expansion Method 18

3.2.1 Convergence with Respect to the Step-size 20

3.2.2 Extension to Higher Dimensional Hypercomplex Algebras. 20

3.3 Numerical Example: Analytic 7 variable function 22

3.3.1 Complex Taylor Series Expansion . . . 22
3.3.2 Quaternion Taylor Series Expansion . . . 22
3.3.3 Octonion Taylor Series Expansion . . . 23
3.4 The Truncated Taylor Series Approach to Compute Functions of Cayley-Dickson Numbers . 24
3.5 Quaternion Finite Element Method . . . 25
3.5.1 Hypercomplex Finite Element method . . . 26
3.5.2 Dependency of Intermediate Variables . . . 32
3.5.3 Assembly and solution of the global system of equations . . . 33
3.6 Abaqus Implementation . . . 36
3.6.1 Direct Cauchy-Riemann approach . . . 36
3.6.2 Block-Solver approach . . . 37
3.7 Numerical Examples . . . 38
3.7.1 Finite Element Abaqus example: Hollow Cylinder with Internal Heat Generation and Convective Surfaces . . . 38
3.7.2 Finite Element Abaqus Example: Cantilever Beam . . . 40
3.8 Discussion . . . 45
3.9 Conclusions . . . 46

Chapter 4: Order Truncated Imaginary Algebra . . . 48
4.1 Motivation . . . 48
4.2 Order Truncated Imaginary Numbers . . . 50
4.2.1 Addition . . . 54
4.2.2 Multiplication . . . 54
4.2.3 Evaluation of elementary functions . . . 56
4.2.4 Computation of Derivatives with OTI Numbers . . . 58
4.2.5 OTI Reduced Order Models . . . 61
4.2.6 Matrix representation . . . 64
4.2.7 Relationship with other hypercomplex algebras. . . . 68
4.3 Support Library . . . 68
4.3.1 Static dense implementation . . . 69

4.3.2 Sparse implementation . . . 71
4.3.3 Python OTI Library (pyOTI) . . . 75
4.4 Results . . . 78
4.4.1 OTI numbers vs Multicomplex and Multidual Algebras . . . 78
4.4.2 Computational Comparison . . . 80
4.4.3 Numerical Example using pyOTI . . . 82
4.4.4 Comparison of OTI implementation with an automatic differentiation library. . . . 83
4.4.5 Numerical Example of OTI Reduced Order Model. . . . 84
4.5 Discussion . . . 87

Chapter 5: Order Truncated Imaginary Finite Element Method . . . 90
5.1 OTI Finite Element Method . . . 90
5.2 Solution of Linear Systems of OTI Equations . . . 95
5.3 Computational Implementation . . . 102
5.3.1 Linear algebra support . . . 102
5.3.2 Finite Element support . . . 103
5.3.3 Usage of the library . . . 104
5.3.4 Expression Equivalence . . . 107
5.4 Results . . . 108
5.4.1 High-Order Gradients and Shape Derivatives . . . 109
5.4.2 Heat Transfer: Hollow Cylinder with Internal Heat Generation and Convective Surfaces. . . . 112
5.4.3 Linear Elasticity: Thick Walled Cylinder Subject to Uniform Pressure. . . . 114
5.4.4 System of Stokes: Lid-driven cavity. . . . 128
5.5 Discussion . . . 134

Chapter 6: Conclusions, Contributions and Future Work . . . 139
6.1 Contribution . . . 140
6.2 Future work . . . 141

Appendix A: Hypercomplex Differentiation Methods **142**
A.1 Complex and Dual Taylor Series Expansion Methods 142
A.2 Multicomplex and Multidual Taylor Series Expansion 143
A.2.1 Computation of Single Variable High Order Derivatives 145
A.2.2 Computation of Multivariable High Order Derivatives 146
A.3 Matrix and Vector Forms of Hypercomplex Numbers 151
A.4 Integration with the Finite Element Method 154
A.4.1 Solution of Hypercomplex System of Equations 155
A.5 Summary 156

Appendix B: Quaternions **157**
B.1 Algebra 157
B.2 Matrix and vector representations 159
B.3 Quaternion Taylor Series Expansion 160
B.4 Fortran support library 164
B.5 Scalar Type 165
B.6 Vector / Matrix support. 166
B.7 Source Code 166

Appendix C: Octonions **170**
C.1 Algebra 170
C.2 Matrix and Vector Representations 172

Appendix D: Weak Forms of Some Partial Differential Equations **173**
D.1 Laplace equation 173
D.2 Heat Conduction for Isotropic Homogeneous Materials 174
D.3 System of Elasticity for Solid Isotropic Materials 175
D.4 Stokes System for Stationary Incompressible Fluids 177

Appendix E: Supported Elements in pyOTI FEM library **178**
E.1 Serendipity elements. 179

Appendix F: Example pyOTI FEM Source Codes 180

F.1 1D Example 180

F.2 2D cylinder 181

F.3 2D Cavity 182

Bibliography 183

LIST OF TABLES

Table 2.1 Comparison between methodologies (deficient areas marked as black) 15

Table 3.1 Perturbation scheme for 7 complex Taylor series expansion analyses to compute 7 first order derivatives. 22

Table 3.2 Perturbation scheme for 3 quaternion Taylor series expansion analysis to compute 7 first order derivatives. 23

Table 3.3 Perturbation scheme for an octonion Taylor series expansion analysis to compute 7 first order derivatives. 23

Table 3.4 Alternative truncated Taylor form to evaluate functions of Cayley-Dickson numbers for CDTSE. 26

Table 3.5 Basic operations required to hypercomplexify a Finite Element code. 36

Table 3.6 CPU time overhead per derivative (1 for CFEM, 3 for QFEM and 7 for OFEM) normalized to a real analysis. 43

Table 3.7 Computational cost relative to a real-only analysis required to compute derivatives using a block-solver solution scheme for a mesh of 81920 elements. 44

Table 4.1 OTI imaginary directions $\boldsymbol{\alpha}_s^p$ in terms of multiplication of imaginary bases. For instance, $\boldsymbol{\alpha}_6^8 = \epsilon_1^3\epsilon_2^5$ and $\boldsymbol{\alpha}_{10}^9 = \epsilon_2^9$. 51

Table 4.2 Corresponding operations of OTI matrix and vector forms. 67

Table 4.3 Capabilities of the standard distribution of OTIlib. 75

Table 4.4 Overloaded operators supported in the current version of the pyOTI. 76

Table 4.5 pyOTI support for elementary functions (for both sparse and static dense versions). . 76

Table 4.6 Comparative example to evaluate $f(x) = x^2$ using OTI and bidual numbers in order to compute all up to second order derivatives. 81

Table 4.7 Maximum relative error per order of derivatives obtained after evaluating function $f(x, y, z) = \sin\left(\log(x^2 e^{yz^3}) \cos\left(x^3 y^2 z^4\right)\right)$. Relative error compared with the derivatives obtained using Sympy. 84

Table 4.8 CPU time comparison of current implementation of pyOTI vs pyAuDi runtime for evaluation of function $f(\mathbf{x}) = \sin\left(\prod_{i=1}^{m}\left(x_i^{10}\right)\right)\cos\left(\prod_{i=1}^{m} x_i\right) / \prod_{i=1}^{m} x_i$ computing all derivatives of m variables up to order n. 85

Table 5.1 Equivalence between analytical variational expressions with its corresponding pyOTI functions. 108

Table 5.2 Input variable values used in the simulation of the thick walled cylinder analysis. . . 115

Table 5.3 Properties of the system of equations for an OTIFEM analysis with 80k degree of freedom for a 2D linear elastic problem, using pyOTI. 116

Table 5.4 Properties of the system of equations FEM analysis solving a 1M degree of freedom 2D linear elastic analysis, using the real module in pyOTI. 121

Table 5.5 CPU time in seconds of the real FEM analysis solving a 1M degree of freedom 2D linear elastic analysis, using the real module in pyOTI. 121

Table 5.6 Properties of the system of equations for an OTIFEM analysis with 91k degrees of freedom for a 2D lid driven cavity problem, using pyOTI. 129

Table A.1 Growth of the number of evaluations the repetitive approach does with respect to the number of variables (m) and order of derivatives (n). 151

Table A.2 Equivalent Cauchy Riemann matrix form of bicomplex and tricomplex numbers. . . 153

Table A.3 Equivalent Cauchy Riemann matrix form of bidual and tridual numbers. 153

Table B.1 List of supported operations and functions available in the Fortran library. 165

Table B.2 List of minimum operations and functions available in the Quaternion Fortran package 165

Table C.1 Multiplication table of Octonion imaginary units. 170

Table D.1 Input variables that define the Laplace boundary value problem. 174

Table D.2 Input variables that define the heat transfer problem for isotropic materials. 175

Table D.3 Input variables that define the linear elastic problem for isotropic materials. 176

Table D.4 Input variables that define the stokes problem. 177

Table E.1 Standard finite elements supported in pyOTI finite element module. 178

Table E.2 Serendipity elements supported in pyOTI finite element module. 179

LIST OF FIGURES

Figure 3.1 Cayley-Dickson doubling process. 17

Figure 3.2 Convergence of the relative error of function $f(a, b, c, d, x, y, z)$ and first derivative $\partial f/\partial x$ for forward differences (FD), central differences (CDi), CTSE, QTSE and OTSE analyses. 24

Figure 3.3 2D 8-noded quaternion quadrilateral element with the corresponding duplicates given by the imaginary directions i, j and k in natural coordinates and the 3×3 full integration scheme. A 3D 20-noded brick element with 20 DOF can be similarly constructed. 27

Figure 3.4 Examples of the nodal perturbations to obtain shape derivatives of a circular feature in a domain. The green circle marks the perturbed region inside which all nodes are perturbed, while the blue and red arrows determine the magnitude of the perturbation, h and $h/2$ respectively. 30

Figure 3.5 Dependency of the intermediate variables of the finite element problem with respect to the input variables for (a) heat transfer and (b) linear elasticity problems. 33

Figure 3.6 (a) Problem schematic (b) Finite element mesh for axysimmetric model 40

Figure 3.7 CFEM, QFEM and OFEM dimensionless nodal temperature sensitivities with respect to h_{ci}, h_{co}, T_i, T_o, r_0, $\hat{s}$ and α. 41

Figure 3.8 Sketch of a cantilever beam anchored at the left end with a vertical load at the right end. 41

Figure 3.9 Performance results of the linear elastic problem with different mesh densities showing (a) CPU time and (b) memory normalized with respect to real analysis of CFEM (1 derivative), QFEM (3 derivatives) and OFEM (7 derivatives) approaches using both the direct Cauchy Riemann (CR) approach and the block-solver (BS) method. A closeup is presented showing the normalized (c) CPU time and (d) memory of the block-solver approach only. 43

Figure 3.10 Optimal selection of analyses type based on the required number of derivatives for a mesh of 81920 elements. 45

Figure 4.1 General distribution of coefficients in OTI matrix form. 65

Figure 4.2 Blocks determined by order separation of the OTI matrix form for an OTI number $\mathbb{OTI}_2^3$. 67

Figure 4.3 Static dense data types for an OTI number with truncation order $n = 3$ and $m = 2$ bases $\mathbb{OTI}_2^3$ (left) and truncation order $n = 2$ and $m = 3$ bases $\mathbb{OTI}_3^2$ (right). 70

Figure 4.4 Sample of a static dense multiplication between two static dense OTI numbers with $m = 3$ bases and truncation order $n = 2$, $\mathbb{OTI}_3^2$. 71

Figure 4.5 Data structure used for the sparse OTI implementation in the C portion of OTIlib. . . 72

Figure 4.6 Visual representation of the sparse data structures defined in OTIlib. 73

Figure 4.7 Comparison between sparse and dense implementations 74

Figure 4.8 Example of multiplication table for imaginary directions with order 1 times directions with order 2 and 3 bases, using (a) a representation using imaginary basis and (b) the final form of the multiplication table using indices. 74

Figure 4.9 Sample codes for evaluation of function $f(x, y, z) = \cos\left(x^3y^2z^4\right)$ using real algebra (left) and OTI numbers for computation of fourth order derivatives with pyOTI (right). 77

Figure 4.10 Comparison of the number of coefficients of multicomplex, multidual and OTI numbers forms. The number of coefficients of the corresponding vector forms (left) and number of non-zero coefficients of the corresponding matrix forms (right) are displayed. 78

Figure 4.11 Number of function evaluations required by multicomplex and multidual algebras for computing multivariable derivatives. OTI numbers in contrast only requires 1 evaluation. 79

Figure 4.12 Computation time of function OTI comparison with two multidual implementations. 82

Figure 4.13 Implementation of the analytic function $f(x,y,z) = \sin\left(\log(x^2e^{yz^3})\cos\left(x^3y^2z^4\right)\right)$ to compute all 7'th order derivatives. 83

Figure 4.14 Sample code for the generation and evaluation of an 50th order OTIRO for function $f(x, y) = \log(xy)$ using pyOTI. The evaluation point, as implemented in this code example, is $x' = 0.5$, $y' = 0.8$ and the approximated evaluation point is $x' = 1.0$, $y' = 1.6$. 86

Figure 4.15 Values of the even $2p$'th mixed derivatives of function $f(x, y) = \log(xy)$ by evaluating it using the traditional form and the expanded form using pyOTI. 87

Figure 4.16 Contour lines delimiting the regions where the relative error of the OTI reduced order model is below 10^{-14} of the function (a) $f(x, y) = \log(xy)$ and (b) $f(x, y) = \log(x) + \log(y)$ using pyOTI. 88

Figure 5.1 2D 8-noded OTI quadrilateral element with the corresponding duplicates given by the imaginary directions ϵ_1, ϵ_2, etc, in natural coordinates and the 3×3 full integration scheme. Other elements for 1D and 3D analyses can be similarly constructed. . . 92

Figure 5.2 Sketch of the OTI nodal perturbations of a rectangular mesh for computation of shape derivatives with respect to (a) the base L_x and (b) the hight L_y of the domain. 93

Figure 5.3 Sample implementation of the 2-node element basis functions in natural coordinates for pyOTI. 104

Figure 5.4 Example domain for Laplace problem. 105

Figure 5.5 Example program that implements the 2D Laplace variational equation using the pyOTI Finite Element module. 106

Figure 5.6 Mesh example for domain in the 2D Laplace equation. 107

Figure 5.7 Result of the 2D Laplace problem contained in `"oti_laplace.vtk"`. The real solution (top figure) shows the standard solution of the problem. The imaginary solution of direction ϵ_1 (bottom figure) shows the sensitivity of the solution $u(x, y)$ with respect to the variation of the left boundary condition value. 108

Figure 5.8 Imaginary perturbations applied to every node in the domain in direction ϵ_1 in order to compute the gradient. 110

Figure 5.9 Gradient derivatives using P1 basis functions. 5.9a shows the approximation of u using the conventional basis functions and 5.9b shows the gradient using OTI perturbations along the nodes that does not belong to the boundary. 111

Figure 5.10 Imaginary perturbations to obtain shape derivatives. 111

Figure 5.11 Shape derivatives using P1 basis functions and OTIFEM. 112

Figure 5.12 Selective perturbations of the domain. 112

Figure 5.13 Result of the selective perturbation to obtain both Eulerian and Lagrangian shape derivatives in a domain with 33 nodes. 113
Figure 5.14 Relative error of the p'th order derivatives of the non-dimensional temperature θ with respect to the source $\hat{s}$. 114
Figure 5.15 Sketch of the thick walled cylinder with internal and external uniform pressures and its computational analysis domain Ω. 115
Figure 5.16 Relative error of the p'th order derivative of the displacement field with respect each of the following input variables: Poisson ratio ν, Young modulus E, outer P_o and inner P_i pressures and outer radius r_o. The absolute error of the derivative of the displacement field $\mathbf{u}$ with respect to ν is shown with a dashed line as the corresponding analytical derivative is zero. 117
Figure 5.17 Perturbation regions for the evaluation of the high order shape derivative behavior for a total of (a) 25%, (b) 50%, (c) 75% and (d) 100% of the elements in the mesh. . 118
Figure 5.18 Relative error of the p'th order shape derivative of the displacement field u with respect to the outer radius r_o evaluated at the point $\hat{q}$ for different number of perturbed elements in the mesh. 119
Figure 5.19 Contour lines delimiting the regions where the relative error of the p'th order OTIROM of the nodal displacements $\mathbf{u}(\hat{q})$ is below 10^{-4} with respect to the analytic solution evaluated using $r_o' = r_o + \Delta r_o$ and $E' = E + \Delta E$. 120
Figure 5.20 Result of the 30'th order OTIROM of the nodal displacement magnitude $\mathbf{u}$ for the new input values $r_o' = r_o + 40\%\ r_o$ and $E' = E + 70\%\ E$. 120
Figure 5.21 Normalized CPU time of the static dense and sparse implementations of real-equivalent numbers in pyOTI for a 2D linear elastic finite element analysis of a mesh with 1M degrees of freedom using (a) Cholesky and (b) SuperLU factorization algorithms. . . 122

Figure 5.22 Normalized CPU time of the static dense OTI, static dense multidual and sparse OTI implementations using pyOTI for a 2D linear elastic finite element analysis on a mesh with 1M degrees of freedom for computation of single variable high order derivatives using (a) Cholesky and (b) SuperLU factorization algorithms; and for computation of high order derivatives of two variables using (c) Cholesky and (d) SuperLU solvers. Multidual evaluations are implemented using the repetitive scheme. 124

Figure 5.23 Normalized CPU time of static dense OTI, multidual and sparse OTI implementations using pyOTI for a 2D linear elastic finite element analysis on a mesh with 1M degrees of freedom for computation of multivariable derivatives of first order using (a) Cholesky and (b) SuperLU factorization algorithms; and for computation of multivariable second order derivatives using (c) Cholesky and (d) SuperLU solvers. Multidual evaluations are implemented using the repetitive scheme. 125

Figure 5.24 Normalized CPU time of sparse OTI numbers in pyOTI a 2D linear elastic finite element analysis in a problem with 80k degrees of freedom for computation of first order derivatives of up to 800 variables, when perturbing less than 1% of the total number of nodes and the total number of nodes in the mesh. 126

Figure 5.25 Normalized CPU time of sparse OTI numbers used for 2D linear elastic finite element analysis in a problem with 80k degrees of freedom for computation of up to 30th order derivatives with respect to the Young modulus E and the domain outer radius r_o, when derivatives were computed independently or in a single analysis. . . 127

Figure 5.26 Normalized CPU time of an OTIROM generated from a 2D linear elastic OTIFEM in a problem with 80k degrees of freedom for up to 30th order reduced order models with respect to the Young modulus E and the domain outer radius r_o. The OTIROM evaluation time correspond to the evaluation of all 80k degrees of freedom. Results are normalized with respect to the total CPU time of a real FEM analysis with 80k degrees of freedom. 128

Figure 5.27 Domain of the cavity analysis. 129

Figure 5.28 Perturbation applied to compute shape derivatives with respect to the domain dimensions L_x and L_y. Note that this image shows quadrilateral elements for simplification purposes only and the actual OTIFEM analysis was performed using triangular elements. 130

Figure 5.29 Magnitude of the velocity field and pressure distribution of the lid-driven cavity problem with dimensions $L_x = L_y = 1$. 131

Figure 5.30 Relative error distribution of the 30'th order OTIROM generated from OTIFEM analysis to the lid-driven cavity problem for variation of the domain dimensions L_x and L_y. The relative error of (a) the velocity magnitude and (b) the pressure are evaluated at point $\hat{q}$. 132

Figure 5.31 Contour lines delimiting the regions with relative error of 10^{-4} for the p'th order OTIROM of the lid-driven cavity problem for (a) the velocity magnitude and (b) the pressure at the point $\hat{q}$ with respect to the FEM solution on the warped mesh. 133

Figure 5.32 Comparison of the results of the simulation of the lid-driven cavity problem for a domain dimensions $L'_x = 1$ and $L'_y = 1.7$. The first row shows the magnitude of the velocity field for (a) the finite element solution on the domain L'_x and L'_y; and (b) the OTIROM generated solving the problem with domain $L_x = 1$ and $L_y = 1$ with perturbed nodal coordinates. The second row of figures show the pressure distribution for (c) the finite element solution on the domain L'_x and L'_y; and (d) the OTIROM approximation. 135

Figure 5.33 Error distribution of the OTI reduced order model for a domain perturbation of $L'_x = 1$ and $L'_y = 1.7$ for (a) the magnitude of the velocity field and (b) the pressure field of the lid-driven cavity simulation. 136

Figure A.1 Representation of the nodal layers according to the imaginary directions in the hypercomplex number. 154

Figure B.1 Example of how to define vectors and matrices using a Fortran quaternion module. . 166

Figure B.2 Example of how to define vectors and matrices using a Fortran quaternion module. . 167

Figure B.3 Implementation of a minimal quaternion number module to be used to compute derivatives of analytic function such as the one presented in section 3.3. 168

Figure B.4 Implementation of the quaternion program to compute all derivatives of the numerical example shown in section 3.3. 169

Figure F.1 Program that implements a 1D example. 180

Figure F.2 Program that implements a 2D Elasticity analysis of a cylinder and computes second order derivatives with respect to the modulus of elasticity. 181

Figure F.3 Program that implements a 2D Stokes analysis of the cavity problem and computes the 30th order OTIROM of the velocity and pressure fields at $L'_x = 0$, $L'_y = 1.7$. 182

Nomenclature

$\mathbb{C}$	Complex number set.
$\mathbb{C}_m$	Multicomplex number set with m bases.
$\mathbb{D}$	Dual number set.
$\mathbb{D}_m$	Multidual number set with m bases.
$\mathbb{H}$	Quaternion number set.
$\mathbb{N}$	Natural numbers. (0, 1, 2, 3,...).
$\mathbb{O}$	Octonion number set.
$\mathbb{OTI}_m^n$	Order Truncated Imaginary number set for m bases with truncation order n.
$\mathbb{R}$	Real numbers.
CTSE	Complex Taylor Series Expansion.
CD	Cayley-Dickson
FEM	Finite Element Method
HTSE	Hypercomplex Taylor Series Expansion.
OTI	Order Truncated Imaginary number.
OTIFEM	Order Truncated Imaginary Finite Element Method
OTIROM	Order Truncated Imaginary Reduced Order Model
OTSE	Octonion Taylor Series Expansion.
QTSE	Quaternion Taylor Series Expansion.
QFEM	Quaternion Finite Element Method
TSE	Taylor Series Expansion.

a^*	Hypercomplex number.
i	Complex imaginary unit: $i = \sqrt{-1}$.
m	Number of imaginary basis.
n	Truncation order.
ϵ	Dual imaginary unit $\epsilon^2 = 0$.
ϵ_l	l'th imaginary basis.
$\boldsymbol{\alpha}_s$	s'th imaginary direction.
$\boldsymbol{\alpha}_s^p$	s'th imaginary direction of order p.
$\mathbf{T}(a^*)$	Matrix form of hypercomplex number a^*.
$\mathbf{t}(a^*)$	Vector form of hypercomplex number a^*.
$\text{Re}[a^*]$	Real coefficient of the imaginary number a^*.
$\text{Im}_{\boldsymbol{\alpha}_s^p}[a^*]$	Coefficient of imaginary direction $\boldsymbol{\alpha}_s^p$.
$(\cdot)_{,x}$	First order derivative with respect to x, i.e. $\partial(\cdot)/\partial x$
$(\cdot)_{,x^2}$	Second order derivative with respect to x, i.e. $\partial^2(\cdot)/\partial x^2$
$(\cdot)_{,xy}$	Second order mixed derivative with respect to x and y, i.e. $\partial^2(\cdot)/\partial x \partial y$

Chapter 1: INTRODUCTION AND PROBLEM STATEMENT

The ability to easily calculate highly accurate first and higher-order derivatives across the broad spectrum of finite element modeling has the promise to transform engineering and computational mechanics. Access to first and higher derivatives with respect to shape, material properties, and loads will provide improvements in material modeling, design and optimization, structural dynamics, machine learning, reliability analysis, structural health monitoring, uncertainty quantification, among others.

In a more detailed manner, the scientific community demands for the following characteristics of sensitivity computation methods in Finite Element analyses:

i) *Highly accurate:* it must provide accurate and usable results,

ii) *Step-size independent*: it must not depend on heuristic determination of its parameters for the method to be reliable,

iii) *Variable-type independent:* (regardless of shape, material property or other parameter),

iv) *Problem independent:* elliptic, parabolic and hyperbolic; linear and nonlinear problems should not impose re-formulation or dismissal of the method,

v) *Efficient*: low memory and low computation time are desired so that impulse the incorporation into larger problems,

vi) *High-order support*: it must provide derivatives of second and higher order,

vii) *Multivariable support*: it should be able to obtain sensitivities with respect to multiple parameters,

viii) *Low programming effort*: complexity of the implementation is proportional to the adoption rate of the method,

ix) *Re-compilation free*: When a modification to the derivatives are required (new sensitivity parameters, new function to compute the sensitivity, etc) should not require to re-compile the program; and

x) *Easy to use*: it should not impose major intellectual restrictions to those that will *use* the finite element method to obtain the desired results.

Much progress has been made within the area of computation of derivatives within the last 40 years but its application to finite element programs is lacking. For example, the current state-of-the-art is encompassed by direct differentiation [1], adjoint methods [2], automatic differentiation (AD) [3] in the form of source code transformations [4], reverse mode AD [5] and forward mode AD [6]; and hypercomplex methods [7,8]. Each of these methods has limitations that prevent widespread application and use. Direct differentiation requires extensive analytic chain rule formulations and source code modifications, and usually only for first order derivatives. This method is efficient but time consuming to implement and not practical for large-scale finite element programs that contain a large variety of element formulations. Adjoint methods require the analytical formulation and programming of an adjoint system of equations and derivatives are usually limited to first order. Source code transformations interrogate the existing source code and develop a new code that computes the numerical derivatives. This method is specific to the supported languages, e.g, C, C++, Fortran, and must be specifically programmed to compute derivatives with respect to the parameter of interest. Both first and second order derivatives can be computed. Reverse AD requires coupling with the analytical adjoint method for efficient finite element implementation; therefore it is problem specific and requires source code modification. Truncated Taylor series polynomials provide high-order derivatives for forward AD, and it is usually implemented through operator overloading. This method can compute arbitrary order derivatives but its current implementation is very inefficient. Hypercomplex methods use complex and dual numbers for first order derivatives and multicomplex [7] and multidual [8] algebras for computation of high order derivatives. The method only lacks efficiency because it does not scale well when increasing the number of variables and order of derivatives required.

Hence, the open requirement in the scientific community, to be studied in the current work, is: *there is no one method that meets the needs for computation of multivariable high-order derivatives in large-scale finite element analyses.* For reasons stated above, the integration of differentiation methods within finite element codes is not prevalent and is not standard practice. In response to this, the present work focuses to address the current limitation of hypercomplex differentiation methods in finite element application by improving the efficiency of the computation of multivariable high order derivatives.

1.1 Research objectives

The general objective of this thesis is to develop and implement a new hypercomplex Finite Element methodology that is specific for efficient derivative computation, such that it empowers the Finite Element Method

with the capability to compute multivariable-high order derivatives of any kind in an easily implementable, accurate and efficient fashion for small and large 1D, 2D and 3D problems.

1.1.1 Specific Objectives

The general objective can be subdivided in the following specific objectives:

i) Explore, implement and validate the use of other hypercomplex algebras, e.g. quaternions, octonions, etc; to compute derivatives more efficiently by avoiding repetition of coefficients.

ii) Develop, implement and validate a new hypercomplex method to efficiently compute large number of high order multivariable derivatives.

iii) Develop, implement and validate a method to integrate the new hypercomplex method into the finite element method to allow the computation of a large number of derivatives*.

*The goal of this thesis is to achieve at least 50 derivatives for a finite element system with 500.000 nodes.

1.2 Outline and Scope of the Thesis

To accomplish the proposed objectives, three main contributions are presented in this dissertation: *i*) the novel use of current hypercomplex algebras to compute multiple first order derivatives efficiently by using quaternions and octonions in finite element analyses; *ii*) a new hypercomplex algebra, called Order Truncated Imaginary (OTI) numbers, that scales accordingly to the number of derivatives computed, and *iii*) a methodology to integrate the new algebra into the Finite Element method based in its matrix form to allow the computation of a large number of derivatives.

This document is organized as follows. Chapter 1 introduces the current methods for computation of derivatives in finite element analysis, problem statement, research objectives and scope of the study. Chapter 2 presents an overview of the state-of-the-art of differentiation methods, with specific focus to finite element analysis. Appendix A compliments this chapter and is provided to the reader who is not familiar with hypercomplex differentiation methods.

Chapter 3 introduces a novel use of Cayley-Dickson algebras for computation of multiple first order derivatives more efficiently. The Complex Taylor Series Expansion was extended to quaternion, octonion and any order Cayley-Dickson algebra. The advantage of this new approach is that highly accurate multiple

first order derivatives can be obtained in a single analysis. Quaternion and octonion-based finite element analysis methods were developed in order to compute up to three (quaternion) and up to seven (octonion) first order derivatives of shape, material properties, and/or loading conditions in a single analysis.

Chapter 4 introduces a new hypercomplex algebra for efficient computation of multivariable high order derivatives, named Order Truncated Imaginary (OTI) numbers. A thorough description of OTI algebra is provided by defining the algebraic operations, elementary function evaluation and the steps required to compute derivatives. An equivalent matrix form of the algebra is derived and a method to create reduced order models from OTI evaluations is presented. A library that implements the algebra is described and tested in multiple scenarios. The accuracy of the method is addressed and performance comparisons provided against multicomplex, multidual and forward mode automatic differentiation implementations. The accuracy of an OTI reduced order model was evaluated for a two variable function formed by computing up to 50th order derivatives with respect to all variables in a single evaluation (for a total of 1326 derivatives).

Chapter 5 introduces the Order Truncated Imaginary Finite Element Method (OTIFEM) method, an OTI empowered finite element analysis tool to compute multivariable high order derivatives. The key component of the method is the matrix form of the OTI number as it allows to formulate an efficient integration with real linear algebra programs. A finite element module was added to the OTI library is described and tested in implementations for heat transfer, linear elasticity and fluid dynamics. Derivatives of up to 30th order were computed and evaluated against the corresponding analytical solutions and used to generate reduced order models. Finally, chapter 6 recapitulates the major findings and contributions of this thesis, and proposes challenges for future research.

Chapter 2: BACKGROUND

Finite differences is probably the most adopted methodology for computation of derivatives in any computer application. The method approximates the first order derivative by evaluating the difference of the function at two points and averaging it by the separating distance h (step-size). Examples of this are

$$\text{Forward Difference: } f_{,x}(x_0) \approx \frac{f(x_0+h)-f(x_0)}{h} \tag{2.1}$$

$$\text{Backward Difference: } f_{,x}(x_0) \approx \frac{f(x_0)-f(x_0-h)}{h} \tag{2.2}$$

$$\text{Central Difference: } f_{,x}(x_0) \approx \frac{f(x_0+h)-f(x_0-h)}{2h} \tag{2.3}$$

where $f(x)$ is a scalar function, x_0 is an evaluation point and $f_{,x}(x_0)$ represents the first order derivative of $f(x)$ with respect to x evaluated at x_0. In order words, the method applies a small perturbation to the variable in the *real* direction to compute the derivative. Higher order derivatives may be also obtained, requiring more evaluations. Second order derivatives can, for example, be obtained by evaluating the function three times:

$$f_{,x^2}(x_0) \approx \frac{f(x_0+h)-2f(x_0)+f(x_0-h)}{h^2} \tag{2.4}$$

where the notation $f_{,x^2}$ corresponds to the second order derivative of f with respect to x, $\partial^2 f/\partial x^2$.

In a computational perspective, only a variation of the input values are necessary and no modifications to the program are required. This makes finite differences applicable to almost any method, and usable by anyone without specific knowledge of the problem (i.e., easy-to-use and easy-to-implement). Hence, this justifies its large adoption among the wide range of scientific areas.

As seen in equation (2.1), the result is an approximation of the derivative and its accuracy depends on the step size value. A large step size reduces the approximation accuracy, and hence a small step size is required. In computational applications however, a small step size might induce *subtractive cancellation error* [9]. Therefore, selection of the step size is a relevant and difficult task because it depends on the value of the variable of interest, and the value of the function to be evaluated.

2.1 Analytic methods

Direct differentiation [1] consist in extensively applying the chain rule to the analytical formulation of the problem. Consider a scalar function $g(\mathbf{u}, p)$ that depends on the solution $\mathbf{u}$ of a linear system of equations and an input parameter p. For instance, in the sense of a physical problem, $g(\cdot)$ could be, e.g. the strain energy and $\mathbf{u}$ the solution of

$$\mathbf{K}\mathbf{u} = \mathbf{b} \tag{2.5}$$

where $\mathbf{K}$ is the global stiffness matrix, $\mathbf{b}$ is the load vector and $\mathbf{u}$ the state function vector. The differentiation of $g(\mathbf{u}, p)$ with respect to p results in

$$\frac{dg}{dp}(\mathbf{u}, p) = g_{,p}(\mathbf{u}, p) + \langle g_{,\mathbf{u}}(\mathbf{u}, p), \mathbf{u}_{,p} \rangle \tag{2.6}$$

where $dg(\mathbf{u}, p)/dp$ is the total derivative of $g(\mathbf{u}, p)$ with respect to p, $g_{,p}(\mathbf{u}, p)$ is the partial derivative $\partial g(\mathbf{u}, p)/\partial p$ and $\mathbf{u}_{,p} = \partial \mathbf{u}/\partial p$.

Differentiating equation (2.5) with respect to any parameter p, the general expression to obtain the derivative of $\mathbf{u}$ with respect to p is

$$\begin{aligned} (\mathbf{K}\mathbf{u})_{,p} &= (\mathbf{b})_{,p} && (2.7) \\ \mathbf{K}\mathbf{u}_{,p} &= \mathbf{b}_{,p} - \mathbf{K}_{,p}\mathbf{u} && (2.8) \end{aligned}$$

where $\mathbf{b}_{,p} = \partial \mathbf{b}/\partial p$ and $\mathbf{K}_{,p} = \partial \mathbf{K}/\partial p$.

This new system of equations will solve for $\mathbf{u}_{,p}$ given the derivatives of the global matrix and load vector with respect to p. Second order derivatives are obtained by, e.g. differentiating with respect to a parameter q,

$$\mathbf{K}\mathbf{u}_{,qp} = \mathbf{b}_{,qp} - \mathbf{K}_{,q}\mathbf{u}_{,p} - \mathbf{K}_{,p}\mathbf{u}_{,q} - \mathbf{K}_{,qp}\mathbf{u} \tag{2.9}$$

where $\mathbf{u}_{,qp}$ is the second order mixed derivative of $\mathbf{u}$ with respect to q and p, $\partial^2 \mathbf{u}/\partial q \partial p$.

In general, the method is computationally efficient and it can be used to obtain high order derivatives given that derivatives of $\mathbf{K}$ and $\mathbf{b}$ are known. In practice, derivatives of the global matrix and load vector are problem dependent, and generally difficult to obtain. Expressions for higher order derivatives, such as (2.8) or (2.9), will become more complex to obtain, as well as the higher order derivatives of K and $\mathbf{b}$. Each required derivative is therefore difficult to obtain, and require extensive implementation details, as each one of them must be coded. If a new derivative is required, the whole code must be added and hence recompilation is necessary.

The semi-analytic method [10, 11] complements direct differentiation by computing the derivatives of $\mathbf{K}$ and $\mathbf{b}$ with a numerical algorithm, e.g. finite differences. This method still requires analytical derivation of some derivatives, e.g. equations (2.8) or (2.9), and is bounded by the accuracy of the method to compute the adjacent derivatives.

The *Adjoint method* [2] for differentiation is another commonly used analytic method. It is most attractive to problems that require derivatives of one function with respect to a large number of variables. Such a case is present, e.g. in Finite Element Analyses that require derivatives of scalar functions that depends on the (energy, compliance, total force, drag coefficient, etc) with respect to multiple nodal coordinates. The method uses the solution of a secondary problem (adjoint problem) in order to replace the computation of $\mathbf{u}_{,p}$ using equation (2.5), and reformulated using its *adjoint* system,

$$\boldsymbol{\lambda} = \mathbf{K}^{-T} g_{,u}(\mathbf{u}, p) \tag{2.10}$$

where $\boldsymbol{\lambda}$ is *the Lagrange* multiplier. The following expression summarizes the adjoint derivative approximation.

$$\frac{dg(\mathbf{u}, p)}{dp} = g_{,p}(\mathbf{u}, p) + \langle \boldsymbol{\lambda}, (\mathbf{b}_{,p} - \mathbf{K}_{,p}\mathbf{u}) \rangle \tag{2.11}$$

The beauty of this method is the fact that $\boldsymbol{\lambda}$ is independent of the sensitivity variable p. Therefore, computing another derivative of g with respect to any other parameter, say q, requires only $\mathbf{b}_{,p}$ and $\mathbf{K}_{,p}$. However, formulation of the adjoint system is highly dependent on the problem, because of its dependence on $g_{,\mathbf{u}}(\mathbf{u}, p)$, $\mathbf{b}_{,p}$ and $\mathbf{K}_{,p}$, which are difficult to obtain and to code. In the general case (where $\mathbf{K}$ is not symmetric), $\boldsymbol{\lambda}$ requires a solution of a full secondary system of equations (equation (2.10)). A variation to function $g(\cdot)$ will require re-formulation and recompilation of the code and re-solution of the numerical

problem.

2.2 Automatic Differentiation

Automatic Differentiation (AD) [3, 5] is a set of numerical methods that use the sequence of elementary operations that computers execute to evaluate the function and compute its derivatives. By applying the chain rule to each operation, it can be propagated in order to obtain the desired derivatives of the functions of interest. Two approaches generalize the application of AD: *Forward mode* and *Reverse mode*, and two implementation methods have been reported: *source code transformation* and *operator overloading*.

Source code transformation [12] is a technique to pre-process the source code of a program so that every relevant operation is transformed to apply the corresponding chain rule (in either forward or reverse AD mode). The advantage of this approach is that compiler-wise optimizations can be applied to every operation and therefore code may run in an efficient manner. However, the resulting programs compute a fixed set of derivatives and will not be able to provide new derivatives unless reprocessed and recompiled [4]. Many approaches, such as [13], will also require source code modifications.

Operator overloading [6, 14, 15] method provides more flexibility to compute derivatives than source code transformation. It is a strategy to modify the behavior of arithmetic operations and functions (`+`, `-`, `*`, `/`, `exp`, etc) according to the type of variable being used.

2.2.1 Forward mode AD

Consider a function $f(p)$ that can be decomposed into a set of intermediate evaluations w_j, such that first $w_1$is evaluated, then $w_2$and so on. Its chain rule decomposition is considered as

$$f(p) = w_n(\dots w_j(\dots w_1(p))) \tag{2.12}$$

$$f_{,p}(p) = f_{,w_n}(p)\dots w_{j+1,w_j} w_{j,w_{j-1}} \dots w_{1,p} \tag{2.13}$$

Forward mode AD [3] propagates the chain rule by computing every derivative of the intermediate variables w_j with respect to input parameter of interest p, i.e. every $w_{j,p}$, evaluating derivatives in the same 'direction' the function is evaluated (first $w_{1,p}$ then $w_{2,p}$,etc). Taylor truncated polynomials (TTP) [6] is an implementation of forward mode AD in order to compute high order, multivariable derivatives. The method consist

of matching the variables of the function with polynomial variables (e.g., x, y, etc). Then, replacing the variable by its corresponding Taylor polynomials, e.g.,

$$p \to T_p = p + p_{,p}x + \frac{1}{2!}p_{,p^2}x^2 \tag{2.14}$$

$$T_p = p + x \tag{2.15}$$

Then, functions are overloaded by the polynomial equivalent of the variables and operations are overloaded by its polynomial equivalents. Hence, function $f(p)$ will become

$$f(p) \to T_f \circ T_p \tag{2.16}$$

where T_f is the Taylor polynomial of $f(p)$, and the expression $T_f \circ T_p$ represents the composition of the two functions. The resulting polynomial will contain the derivatives of the function within its coefficients. The degree of the polynomial term matches explicitly the order of the derivative to be obtained, e.g. $x^2 \to \partial(\cdot)/\partial x^2$, $xy \to \partial(\cdot)/\partial x\partial y$. In spite of its ability to compute high order derivatives, current implementations do not provide the required performance for a large scale computer implementation. Dense univariate implementations, such as Arbogast or Sacado [14, 16], can compute derivatives with respect to only one variable per run, and since they manipulate the full polynomials, it becomes expensive. Dense multivariate implementations, e.g., ADOL-C [15], also suffers performance drawbacks from manipulating the full polynomials for high order derivatives, and are very difficult to use. Sparse multivariate implementations, e.g., AUDI [6], depend on arbitrary precision arithmetics for the polynomial manipulations [17, 18], increasing the overall computational cost. In every case, interface with linear algebra solvers is a bottleneck, and similar to the reverse mode AD, integration with an analytic method is required, in this case the semi-analytic method. This makes the method problem dependent, complicated to use and inefficient.

Another forward AD high order method is the nilpotent matrix approach [19]. In this approach the variables are overloaded by sparse matrices that will evaluate the derivatives through the operations. This is particularly inefficient because every real arithmetic operation are replaced by sparse matrix algebra operation.

2.2.2 Reverse mode AD

Reverse mode AD [5] applies the chain rule in the backward direction so that derivatives of the function f with respect to every intermediate variable are computed, i.e., every $f_{,w_j}$. To do so, the method requires storing every intermediate variable w_j. Reference [20] pointed out that application of the reverse mode to problems with linear system of equations will result in excessive CPU and memory loads. Strategies are therefore required to avoid passing through the linear algebra solvers. Also, reference [5] highlight the need to use the analytic adjoint formulation of the problem in order to obtain the desired derivatives, which therefore implies the disadvantages of analytical methods: high difficulty to formulate, problem dependent, necessity to re-compile a program, dependency of the type of variable needed, and high programming effort. Although reverse AD has been used in computational fluid dynamics to compute shape derivatives [21–23], it is not attractive for a large scale application in the finite element method.

2.3 Hypercomplex Algebras

The complex Taylor series expansion (CTSE) method has been shown to be an effective procedure for obtaining accurate numerical first order derivatives from arbitrary holomorphic functions [24, 25]. The method is simple in concept and straightforward to implement: convert a real-valued code to complex-valued and introduce a small perturbation along the imaginary axis of the parameter of interest. The derivative of each degree-of-freedom with respect to the parameter of interest is then contained within its imaginary component. Derivatives of auxiliary results can then be obtained using standard post-processing methods such as obtaining strains and stresses and their derivatives within an elasticity analysis. This method obtains results with machine precision accuracy if the step size is sufficiently small, say $\leq 10^{-10}$ times the parameter of interest. CTSE is subtractive cancellation error free, meaning that derivatives can be computed with an arbitrarily small step size.

The set of hypercomplex algebras [26] contain those formed by one or more "imaginary" units. Examples of algebras with one imaginary unit are complex ($i^2 = -1$), dual ($\epsilon^2 = 0$) and double ($\varepsilon^2 = 1$) numbers. Dual numbers provide an equivalent method to CTSE regarding the computation of first order sensitivities [27]. In dual algebra, the first order sensitivities are obtained by perturbing the variable in the imaginary direction ϵ by h. In contrast to CTSE, dual algebra analyses are insensitive to the step-size h and thus the derivatives are always machine error accurate. However, dual libraries are not standard in most pro-

gramming languages, while complex types are more widely available to the end user and most programming languages have efficient complex implementations.

The extension of CTSE or dual methods to compute more derivatives requires larger algebras. For instance, quaternions [28], can be used to compute up to three first order derivatives [29]. Multiple higher order multivariable derivatives require the use of hypercomplex algebras such as multicomplex [7, 30] or hyperdual [8] (herein referred as multidual). For example, computing two first order derivatives requires a bicomplex or bidual implementation, three derivatives requires tricomplex or tridual, etc. This approach is accurate but becomes grossly inefficient as more derivatives are added because the resulting size of the matrix representation increases by a factor of 2 with each increase in dimension. That is, a bicomplex matrix is 4 times larger and a tricomplex is 8 times larger than a real-valued analysis. As a result, using multi-complex/dual methods to compute multivariable high order derivatives is not advantageous within a finite element formulation as the size of the element and global stiffness matrices becomes intractable.

Successful implementation of hypercomplex algebras in the Finite Element Method include applications in heat transfer [31, 32], structural dynamics [33], thermoelasticity [34], solid mechanics [35], plasticity [36], fluid dynamics [37, 38], aerodynamics and aero-structural analysis [39, 40], dynamic system optimization [41], pseudospectral [42] and eigenvalue sensitivity methods [43], among others. In [32], complex algebra was used to improve the performance of the semi-analytic method in order to reduce both the computational time expense and the sensitivity to the perturbation size compared with the standard finite-difference-driven semi-analytic method. This, however, was only developed for first order derivatives. A particularly successful application area of hypercomplex differentiation methods is to determine the energy release rate for linear [44] and nonlinear fracture mechanics [45]. The energy release rate is the derivative of the strain energy with respect to a crack extension. Multicomplex and multidual numbers can be used to compute high order strain energy derivatives with respect to the crack extension in [46, 47] to perform 2D curvilinear progressive fracture analyses. Up to 6th order derivatives with respect to the crack tip extension were reported. This methodology, however, requires multiple solutions of finite element problems to compute all high order derivatives required, making the method computationally intensive.

Multiduals were used in [48] to generate reduced order models by generating a Taylor series expansion of a finite element solution using derivatives with respect to shape and material input parameters. Reduced order models were generated to the vibration analysis of a beam, and results showed that third order derivatives were sufficient to generate a reduced model for the 100% variation of material inputs but not as

successful for a shape inputs, as evaluating the model with less than 5% variation to the shape inputs was enough to exceed 1% of error from the solution.

The key element for a successful finite element hypercomplex method is the *equivalent matrix form* of the algebras that allows a direct integration with linear algebra solvers [33, 47, 49, 50]. The disadvantage of the methodology is that the number of degrees of freedom of the analysis is multiplied by the size of the algebra, exponentially increasing the computational resources needed to solve the problem with large algebras hypercomplex algebras. A novel approach presented in [51] for computation of the energy release rate, avoided forming the larger system of equations by computing the derivatives of the stiffness matrix and force vector at an element level. The method is mostly equivalent to the adjoint method in the sense that an analytical approach was required to bypass the computation of the derivatives of the displacement vector with respect to the crack extension via equation (2.11). It is, however, a particular solution for the computation of the energy release rate, thus not a solution generalized for any derivative computation.

A broader adoption of the hypercomplex differentiation methods has been limited by the availability of support libraries, as only the complex variable type is supported in the majority of languages. Linear algebra routines can be used with the matrix form of the algebras to avoid requirement of specific implementations, as discussed in [52] for multiduals, but the overhead of matrix computations make this type of implementations not fit for large problems as in finite elements analyses [19]. A library with support for multicomplex and multidual algebras was developed [53] with support for Python and Fortran programming languages.

Current hypercomplex algebras, therefore, do not scale well when computing high order multivariable derivatives because of the excessive growth in the size of the algebra does not match the number of derivatives required. As a result, the method is not computationally efficient. The reader not familiar with hypercomplex algebras is referred to Appendix A for an extended insight.

2.4 Summary

Table 2.1 summarizes the state-of-the-art methods with deficiencies marked as empty blocks. It can be observed that the full suite of these demands are not all satisfied by any current method to date, specially in regards of the Finite Element Method. This work will address this open gap in knowledge by three main contributions: *i*) A novel method that extends the use of use *curren*t hypercomplex algebras to efficiently compute multiple first order derivatives by avoiding the repetition of coefficients; *ii*) a new hypercomplex algebra, called Order Truncated Imaginary (OTI) numbers, which is efficient to compute derivatives of high

order multivariable functions and *iii*) a methodology to integrate the new algebra into the Finite Element method to allow the computation of a large number of derivatives.

Method	***(i)*** Highly accurate	***(ii)*** Step-size independent	***(iii)*** Variable-type independent	***(iv)*** Problem independent	***(v)*** Efficient	***(vi)*** Support of high order derivatives	***(vii)*** Support multivariable derivatives	***(viii)*** Low programming effort	***(ix)*** Recompilation free	***(x)*** Easy to use
Finite Differences			+	+		+	+		+	+
Direct Differentiation	+	+		+	+	+	+			
Adjoint Method	+	+		+	+	+	+			
Source Code Transformation	+	+			+	+	+	+		+
Reverse Automatic Differentiation	+	+				+	+			
Truncated Taylor Polynomials	+	+	+			+	+	+	+	
Current Hypercomplex Algebras	+	+	+	+		+	+	+	+	+

Table 2.1: Comparison between methodologies (deficient areas marked as black)

Chapter 3: QUATERNION AND OCTONION-BASED FINITE ELEMENT ANALYSIS METHODS FOR COMPUTING MULTIPLE FIRST ORDER DERIVATIVES

M. Aristizabal, D. Ramirez-Tamayo, M. Garcia, A. Aguirre-Mesa, A. Montoya and H. Millwater

Journal of Computational Physics 397 (2019) 108831

Disclaimer: *This section corresponds to the publication [54] "Quaternion and octonion-based finite element analysis methods for computing multiple first order derivatives" published in the Journal of Computational Physics 397 (2019) 108831 and is a partial contribution of this thesis. The publication has been reformatted to fit the style of this thesis and some sections may have been modified and renamed to better fit this document. The author of this thesis acknowledges the contributions from all other co-authors.*

Current hypercomplex differentiation methods do not scale well for computation of first order derivatives with respect to multiple variables as either high-order derivatives are also computed or multiple re-evaluations of the problem are necessary (see Appendix A). This chapter describes an extension of the complex Taylor series expansion method to quaternion, octonion and any order Cayley-Dickson algebra. The advantage of this new approach is that highly accurate multiple first order derivatives can be obtained in a single analysis. Quaternion and octonion-based finite element analysis methods were developed in order to compute up to three (quaternion) and up to seven (octonion) first order derivatives of shape, material properties, and/or loading conditions in a single analysis. The traditional finite element formulation was modified such that each degree-of-freedom was augmented with three or seven additional imaginary nodes. The quaternion and octonion-based methods were integrated within the Abaqus commercial finite element code through a user element subroutine. Numerical examples are presented for thermal conductivity and linear elasticity; however, the methodology is general. The results indicate that the quaternion and octonion-based methods provide derivatives of the same high accuracy as the complex finite element method but are significantly more efficient.

3.1 Motivation

Quaternion algebra was introduced by Hamilton in 1843 to represent the mechanics of motion in three-dimensional space [55, 56]. Quaternion numbers contain a real coefficient plus three coefficients along three imaginary directions. Their main application today is in three-dimensional computer graphics and

computer vision, where they are used to compute three-dimensional rigid-body rotations [57]. Quaternions have also been used within the finite element method in order to represent rotations for beam element formulations [58–60].

Quaternion numbers are a subset of the Cayley-Dickson (CD) hypercomplex numbers [26]. CD algebras are formed by doubling a member of CD numbers, which contains the duplication of the number of dimensions of the initial algebra. For example, complex algebra (2 dimensions) is the result of doubling the real numbers, quaternion algebra (4 dimensions) is the result of doubling the complex numbers, etc. This process generates the set of CD numbers, which includes Octonions (8), Sedenions (16), etc, and associated algebraic rules. Figure 3.1 shows the algebraic hierarchy available from the Cayley-Dickson doubling process. The number of real values required to describe a number for each case is doubled for each new algebra; Real (1), Complex (2), Quaternions (4), etc. In each case, $n-1$ imaginary units are available for computing derivatives in a manner analogous to the complex Taylor series expansion method; Real (0), Complex (1), Quaternions (3), Octonions (7), Sedenions (15), etc.

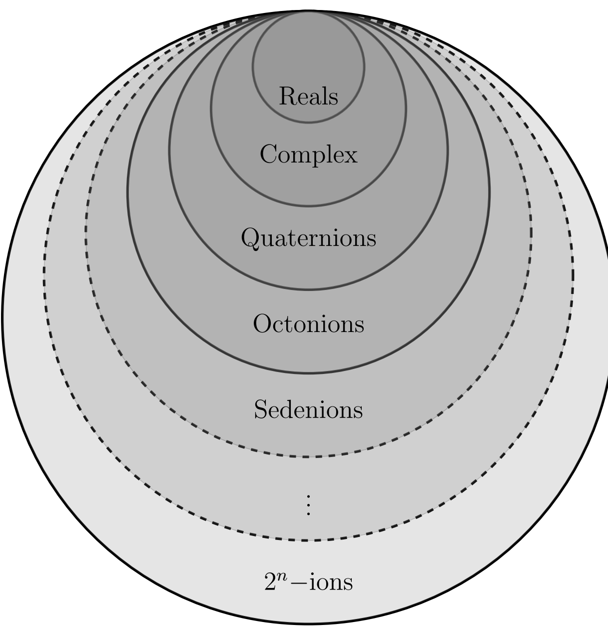


Figure 3.1: Cayley-Dickson doubling process.

Turner presented a method that used quaternions to compute up to three first order derivatives [29]. The work used all three imaginary directions with small perturbations, such that when using the equivalent quaternion operations, the result contained the gradient of the function with respect to the perturbed variables, stating quadratic convergence with respect to the step size h. Turner also presented a method to solve

systems of linear quaternion equations, which required programming a quaternion-based solver.

The present work extends Turner's work through the use of higher dimensional algebras such as octonions. It also shows that quadratic convergence of the derivatives with respect to the step size is only achieved under certain conditions and that linear convergence, not quadratic, is the expected convergence rate of the derivative results.

Moreover, the present work incorporates the quaternion and octonion Taylor Series expansion method into the Finite Element Method. A quaternion-based finite element formulation (QFEM), as described in this article, is capable of computing up to three first order derivatives in a single analysis without increasing the size of the stiffness matrix and an octonion-based finite element method (OFEM) can compute up to seven first order derivatives. This approach can be extended to sedenions (15 derivatives) and higher dimensional CD algebras in a similar manner, thus generalized in the CDFEM method. The advantage of these methods is that the QFEM, OFEM, etc, can obtain multiple first order derivatives in a single analysis far more efficiently than, for instance, an equivalent multicomplex/multidual implementation.

Also, the present work introduces two methods for solving systems of linear hypercomplex equations using standard real-based solvers consisting of: i) a method based on the Cauchy-Riemann equivalent matrix form of the hypercomplex algebra, and ii) a new block solver method that subdivides the hypercomplex system into multiple real-only systems of equations that can be solved using multiple right hand sides with a single factorization of the real-valued stiffness matrix.

3.2 Quaternion Taylor Series Expansion Method

The quaternion Taylor series method (QTSE) is an extension of the complex Taylor series method (CTSE). CTSE [25, 61] is a methodology to compute the sensitivity of a real function with respect to a variable of interest that is analogous to the finite difference (FD) method in the sense that a perturbation is applied to the parameter of interest. However, in CTSE the perturbation of the variable of interest x is applied along the imaginary axis, becoming $x^* = x_0 + hi$, where x_0 represents the point at which the derivative of the function is required and h is the perturbation step, addressed after Equation (3.3). The typical procedure consists of replacing every operation in the real function of interest f by its complex-equivalent operation (a process known as complexification of f), e.g. real multiplication is replaced by complex multiplication, etc. Using the complex-variable Taylor series, a function F can be expressed as

$$F(x^*) = F(x_0 + hi) = f(x_0) + f_{,x}(x_0)h\, i - \frac{1}{2!}f_{,x^2}(x_0)h^2 - \frac{1}{3!}f_{,x^3}(x_0)h^3\, i + \mathcal{O}(h^4) \tag{3.1}$$

where F is the result of complexifying f. Also, $f_{,x}$ represents the first order derivative with respect to x, $f_{,x^2}$ the second order derivative, etc. Note that the negative signs in Equation (3.1) come from factoring terms with i^2 and replacing it by the complex identity $i^2 = -1$. Taking the real and imaginary parts of both sides, Re$[\cdot]$ and Im$[\cdot]$ respectively; and solving for the function and its first order derivative, the following is obtained

$$f(x_0) = \text{Re}\,[F(x^*)] + \mathcal{O}(h^2) \tag{3.2}$$

$$f_{,x}(x_0) = \frac{1}{h}\text{Im}\,[F(x^*)] + \mathcal{O}(h^2) \tag{3.3}$$

Note that the perturbation step size h can be made arbitrarily small with no concern about subtraction cancellation error. Hence, higher order effects of $\mathcal{O}(h^2)$ can be made negligible through the use of a small h. A typical value for h is 10^{-10} times the value of the parameter being perturbed.

The Quaternion Taylor Series Expansion (QTSE) method uses quaternion algebra (see Appendix B) to compute up to 3 first order derivatives of a real function evaluated at $x = x_0$, $y = y_0$, $z = z_0$. The variables x, y and z are perturbed by a small step size h in imaginary directions i, j and k respectively:

$$\begin{aligned} x^* &= x_0 + \boldsymbol{hi} + 0j + 0k, \\ y^* &= y_0 + 0i + \boldsymbol{hj} + 0k, \\ z^* &= z_0 + 0i + 0j + \boldsymbol{hk} \end{aligned} \tag{3.4}$$

The quaternion-variable Taylor series expansion (see B.3) adapted for the perturbations as shown in Equation (3.4), becomes

$$\mathcal{F}(x^*, y^*, z^*) = f(x_0, y_0, z_0) + h\,(f_{,x}(x_0, y_0, z_0)i + f_{,y}(x_0, y_0, z_0)j + f_{,z}(x_0, y_0, z_0)k) + \mathcal{O}(h^2) \tag{3.5}$$

where $\mathcal{F}$ is the quaternion-equivalent function of the real function f, which corresponds to the quaternion-

ization of the operations in f. The result of $\mathcal{F}(x^*, y^*, z^*)$ is a quaternion containing the real function result and first order derivatives of f with respect to x, y and z, one lying along each imaginary direction:

$$f(x_0, y_0, z_0) \approx \text{Re}\left[\mathcal{F}(x^*, y^*, z^*)\right] \tag{3.6}$$

$$f_{,x}(x_0, y_0, z_0) \approx \text{Im}_i\left[\mathcal{F}(x^*, y^*, z^*)\right]/h \tag{3.7}$$

$$f_{,y}(x_0, y_0, z_0) \approx \text{Im}_j\left[\mathcal{F}(x^*, y^*, z^*)\right]/h \tag{3.8}$$

$$f_{,z}(x_0, y_0, z_0) \approx \text{Im}_k\left[\mathcal{F}(x^*, y^*, z^*)\right]/h \tag{3.9}$$

where $\text{Re}\left[\cdot\right]$ extracts the real coefficient of the quaternion number, $\text{Im}_i\left[\cdot\right]$ extracts the coefficient associated with the imaginary direction i, etc. See B.3 for a more detailed elaboration.

Since no subtractions are performed during quaternion algebra operations, the method does not suffer from subtraction cancellation error. In practical numerical applications, the minimum possible error is usually bounded by machine error.

3.2.1 Convergence with Respect to the Step-size

Although Turner [29] expressed that when using quaternions to compute derivatives the convergence of the first order derivatives is quadratic with respect to the step size (as in CTSE), it can be shown that it is not the case when quaternion multiplication is present.

For the most general case, the function value converges quadratically with respect to the step-size h whereas the derivatives, Equations (3.7) - (3.9), converge linearly. These convergence rates are shown in the numerical examples section. In some special cases but not generally, quadratic convergence may be obtained in the first order derivatives. See B.3 for more details.

On the other hand, CTSE converges quadratically on both function, Equation (3.2), and its derivative, Equation (3.3). However, since both complex and quaternion methods are subtraction cancellation error free, both will converge to the same error when using a sufficiently small h with no implication in performance.

3.2.2 Extension to Higher Dimensional Hypercomplex Algebras.

In order to use higher dimensional CD algebras to compute multiple first order derivatives, e.g. octonions [62], sedenions [63], etc., each variable is associated with and perturbed independently along a unique

imaginary direction, analogous to Equation (3.4). The maximum number of variables that the algebra is capable to compute derivatives with respect to in a single analysis is given by the total number of imaginary directions. This forms the generalized Cayley-Dickson Taylor series expansion method (CDTSE).

In the case of octonions (see Appendix C), a total of 8 coefficients are present in the algebra, corresponding to the real and 7 imaginary directions. Therefore, the algebra can be used to compute up to 7 first order derivatives in a single analysis. In addition, the 16-dimensional algebra of sedenions (one real and 15 imaginary directions) is able to compute 15 derivatives in a single analysis.

For octonions, forming the octonion Taylor series expansion (OTSE) method, the following perturbation scheme is used for a 7 variable function $f : \mathbb{R}^7 \to \mathbb{R}$, $f(x, y, z, a, b, c, d)$:

$$\begin{aligned} &x^* = x_0 + \boldsymbol{hi} + 0j + 0k + 0e' + 0i' + 0j' + 0k', && b^* = b_0 + 0i + 0j + 0k + 0e' + \boldsymbol{hi'} + 0j' + 0k', \\ &y^* = y_0 + 0i + \boldsymbol{hj} + 0k + 0e' + 0i' + 0j' + 0k', && c^* = c_0 + 0i + 0j + 0k + 0e' + 0i' + \boldsymbol{hj'} + 0k', \\ &z^* = z_0 + 0i + 0j + \boldsymbol{hk} + 0e' + 0i' + 0j' + 0k', && d^* = d_0 + 0i + 0j + 0k + 0e' + 0i' + 0j' + \boldsymbol{hk'} \\ &a^* = a_0 + 0i + 0j + 0k + \boldsymbol{he'} + 0i' + 0j' + 0k' \end{aligned} \tag{3.10}$$

Derivatives are obtained by evaluating f at $(x^*, y^*, z^*, a^*, b^*, c^*, d^*)$, which results in a hypercomplex octonion number, then extracting the imaginary coefficients and dividing each by the perturbation h. In the case of the perturbed variables in Equation (3.10), the evaluated function gives

$$f(x_0, y_0, z_0, a_0, b_0, c_0, d_0) \approx \text{Re}\,[f(x^*, y^*, z^*, a^*, b^*, c^*, d^*)] \tag{3.11}$$

$$f_{,x}(x_0, y_0, z_0, a_0, b_0, c_0, d_0) \approx \text{Im}_i\,[f(x^*, y^*, z^*, a^*, b^*, c^*, d^*)]/h \tag{3.12}$$

$$f_{,y}(x_0, y_0, z_0, a_0, b_0, c_0, d_0) \approx \text{Im}_j\,[f(x^*, y^*, z^*, a^*, b^*, c^*, d^*)]/h \tag{3.13}$$

$$f_{,z}(x_0, y_0, z_0, a_0, b_0, c_0, d_0) \approx \text{Im}_k\,[f(x^*, y^*, z^*, a^*, b^*, c^*, d^*)]/h \tag{3.14}$$

$$f_{,a}(x_0, y_0, z_0, a_0, b_0, c_0, d_0) \approx \text{Im}_{e'}\,[f(x^*, y^*, z^*, a^*, b^*, c^*, d^*)]/h \tag{3.15}$$

$$f_{,b}(x_0, y_0, z_0, a_0, b_0, c_0, d_0) \approx \text{Im}_{i'}\,[f(x^*, y^*, z^*, a^*, b^*, c^*, d^*)]/h \tag{3.16}$$

$$f_{,c}(x_0, y_0, z_0, a_0, b_0, c_0, d_0) \approx \text{Im}_{j'}\,[f(x^*, y^*, z^*, a^*, b^*, c^*, d^*)]/h \tag{3.17}$$

$$f_{,d}(x_0, y_0, z_0, a_0, b_0, c_0, d_0) \approx \text{Im}_{k'}\,[f(x^*, y^*, z^*, a^*, b^*, c^*, d^*)]/h \tag{3.18}$$

Similar to QTSE, using CDTSE methods, the convergence with respect to the step size is quadratic for the function value and linear for the derivatives. All results are also subtractive cancellation error free.

3.3 Numerical Example: Analytic 7 variable function

The proposed sensitivity method is applied to an analytic function of seven variables

$$f(a,b,c,d,x,y,z) = \sin(a^3b^2c^3d^4x^3y^2z^4) \tag{3.19}$$

to compute $f_{,a}$, $f_{,b}$, $f_{,c}$, $f_{,d}$, $f_{,x}$, $f_{,y}$ and $f_{,z}$ at the values of $a = 0.1$, $b = 0.2$, $c = 0.3$, $d = 0.4$, $x = 0.5$, $y = 0.6$ and $z = 0.7$. In particular, this example will be computed using seven CTSE, three QTSE and one OTSE analyses to serve for comparison purposes.

3.3.1 Complex Taylor Series Expansion

Using CTSE, seven analyses must be performed to obtain the desired derivatives. In this case, each analysis computes the first order derivative with respect to one variable. Table 3.1 shows the perturbations required per function evaluation in order to obtain all seven derivatives.

	Evaluation						
Variable	**1**	**2**	**3**	**4**	**5**	**6**	**7**
a^*	$0.1+hi$	0.1	0.1	0.1	0.1	0.1	0.1
b^*	0.2	$0.2+hi$	0.2	0.2	0.2	0.2	0.2
c^*	0.3	0.3	$0.3+hi$	0.3	0.3	0.3	0.3
d^*	0.4	0.4	0.4	$0.4+hi$	0.4	0.4	0.4
x^*	0.5	0.5	0.5	0.5	$0.5+hi$	0.5	0.5
y^*	0.6	0.6	0.6	0.6	0.6	$0.6+hi$	0.6
z^*	0.7	0.7	0.7	0.7	0.7	0.7	$0.7+hi$

Table 3.1: Perturbation scheme for 7 complex Taylor series expansion analyses to compute 7 first order derivatives.

According to Table 3.1, the first evaluation computes the first order derivative $f_{,a}$. The second evaluation computes $f_{,b}$, the third $f_{,c}$, etc.

3.3.2 Quaternion Taylor Series Expansion

Three evaluations are required to compute all seven derivatives using quaternion algebra. The perturbation schemes per evaluation are described in Table 3.2.

	Evaluation		
Variable	**1**	**2**	**3**
a^*	$0.1 + hi$	0.1	0.1
b^*	$0.2 \quad + hj$	0.2	0.2
c^*	$0.3 \quad\quad + hk$	0.3	0.3
d^*	0.4	$0.4 + hi$	0.4
x^*	0.5	$0.5 \quad + hj$	0.5
y^*	0.6	$0.6 \quad\quad + hk$	0.6
z^*	0.7	0.7	$0.7 + hi$

Table 3.2: Perturbation scheme for 3 quaternion Taylor series expansion analysis to compute 7 first order derivatives.

According to Table 3.2, the first evaluation computes $f_{,a}$, $f_{,b}$ and $f_{,c}$, contained in the imaginary directions i, j and k respectively. The second evaluation computes $f_{,d}$, $f_{,x}$ and $f_{,y}$. Finally, the third evaluation computes $f_{,z}$.

3.3.3 Octonion Taylor Series Expansion

Octonion algebra is formed by 7 imaginary directions, namely i, j, k, e', i', j' and k'. For a description of the algebra see Appendix C.

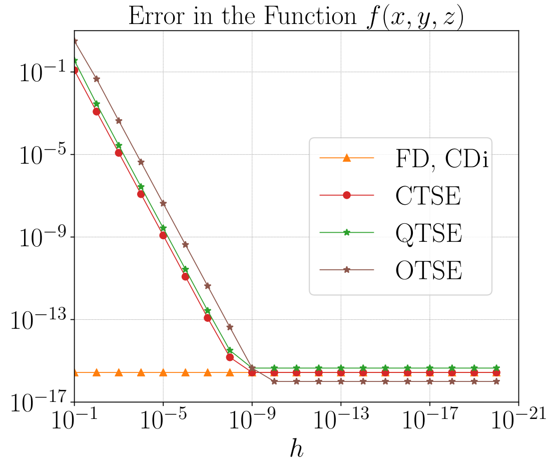
In order to evaluate the function in Equation (3.19) with octonions, only one analysis is required in order to compute all 7 derivatives. The perturbations are done as shown in Table 3.3.

	Evaluation
Variable	**1**
a^*	$0.1 + hi$
b^*	$0.2 \quad + hj$
c^*	$0.3 \quad\quad + hk$
d^*	$0.4 \quad\quad\quad + he'$
x^*	$0.5 \quad\quad\quad\quad + hi'$
y^*	$0.6 \quad\quad\quad\quad\quad + hj'$
z^*	$0.7 \quad\quad\quad\quad\quad\quad + hk'$

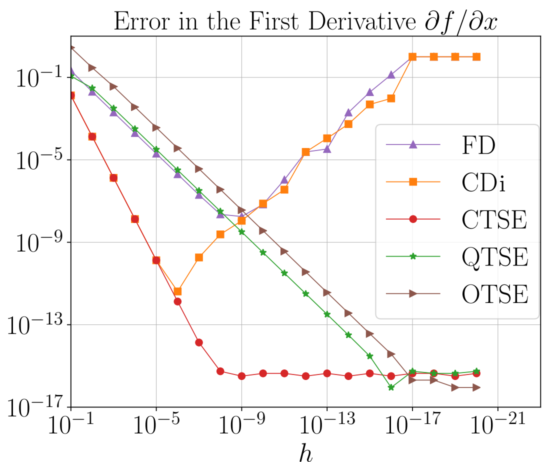
Table 3.3: Perturbation scheme for an octonion Taylor series expansion analysis to compute 7 first order derivatives.

Convergence of the relative error for first order derivatives and function results with respect to the step-size are shown in Figure 3.2. Notice that the convergence rate of the derivative for the quaternion and octonion analyses is of first order. Also, all CDTSE methods evaluated: CTSE, QTSE and OTSE reached machine precision accuracy using step sizes h equal to 10^{-8}, 10^{-16} and 10^{-17} respectively. In practical

terms, all analyses can be evaluated with a step-size $h = 10^{-17}$ or smaller. The source code for this example is given in B.4.



(a) Relative error of the function $f(a, b, c, d, x, y, z)$ value with respect to the step size h.



(b) Relative error of the first derivative $\partial f/\partial x$ with respect to the stepsize h.

Figure 3.2: Convergence of the relative error of function $f(a, b, c, d, x, y, z)$ and first derivative $\partial f/\partial x$ for forward differences (FD), central differences (CDi), CTSE, QTSE and OTSE analyses.

3.4 The Truncated Taylor Series Approach to Compute Functions of Cayley-Dickson Numbers

The equations to compute typical functions of quaternions are well known [29] and it is possible to use them within the QTSE method described in section 3.2. However, the computational complexity of quaternion functions is usually high, and the methods for evaluation of other CD functions are not widely available. Hence a new method to evaluate functions of quaternions, octonions, and other hypercomplex algebras is presented here for use with the CD numbers containing very small imaginary components. The method is based on a Taylor series expansion of the CD function, extending the multicomplex and multidual function evaluation method proposed in [53] to CD numbers, with the condition that in the context of CDTSE, the step size h is small so that high order terms can be neglected.

For a single variable function $f(x) : \mathbb{R} \to \mathbb{R}$, e.g. $\sin(x)$, the proposed truncated Taylor series approach consists of the following. Consider a hypercomplex function $f(q^*)$. Its second order Taylor series expansion is

$$f(q^*) = f(q_r) + f_{,x}(q_r)\, v^* + \frac{1}{2} f_{,xx}(q_r)\ (v^*)^2 \tag{3.20}$$

where $q_{_{Re}}$ is the real coefficient of the number and v^* is the whole imaginary part of q^*. For example, in the case of a quaternion q^*,

$$q^* = q_r + q_i i + q_j j + q_k k \tag{3.21}$$

v^* is equal to the same quaternion number with real coefficient equal to zero, i.e.

$$v^* = q^* - q_r = q_i i + q_j j + q_k k \tag{3.22}$$

Since v^* is only imaginary, and in particular to CDTSE applications all imaginary coefficients are multiplied by small values of h, its square and higher powers will result in terms multiplied by h^2 and higher. Thus, only for CDTSE applications, higher order terms may be neglected. Therefore, a function for CDTSE applications is evaluated as:

$$f(q^*) = f(q_r) + f_{,x}(q_r)\, v^* \tag{3.23}$$

For instance, evaluating the sine function of a CD number q^*can be evaluated as:

$$\sin(q^*) = \sin(q_r) + \cos(q_r) v^* \tag{3.24}$$

where $\sin(q_r)$ and $\cos(q_r)$ are the standard real trigonometric functions, evaluated at the real-value $q_{_{Re}}$.

The error induced by the higher order terms is negligible due to the accompanied power of the imaginary part of q^*. Hence, the effect of higher order terms can be made negligible through the use of a small h. The result is that functions of quaternions, octonions, etc, can be evaluated using one evaluation of a function of a real variable. Table 3.4 summarizes some commonly used functions.

3.5 Quaternion Finite Element Method

The Quaternion Finite Element Method (QFEM) is based in the Hypercomplex Finite Element Method (ZFEM) that consists of replacing the real-valued variables with hypercomplex (quaternion) variables in order to compute the desired derivatives. In the case of quaternions, derivatives with respect to a maximum

Function	Symbol	Equivalence
Sine	$\sin(q^*)$	$\sin(q_r) + \cos(q_r)v^*$
Cosine	$\cos(q^*)$	$\cos(q_r) - \sin(q_r)v^*$
Square root	$\sqrt{q^*}$	$\sqrt{q_r} + \dfrac{1}{2\sqrt{q_r}}v^*$
Power	$(q^*)^m$	$q_r^m + mq_r^{(m-1)}v^*$
Exponential	e^{q^*}	$e^{q_r} + e^{q_r}v^*$
Natural logarithm	$\ln(q^*)$	$\ln(q_r) + \dfrac{v^*}{q_r}$

Table 3.4: Alternative truncated Taylor form to evaluate functions of Cayley-Dickson numbers for CDTSE.

of three variables. For simplicity, a heat transfer and a linear elasticity finite element implementations will be used to show the methodology, see Appendix D. However, any other physics can also be implemented similarly.

3.5.1 Hypercomplex Finite Element method

The Hypercomplex Finite Element Method (ZFEM) formulation [44, 50, 64] is described here in terms of quaternion variables but can be similarly extended to octonions, and other hypercomplex algebras. The Quaternion Finite Element Method (QFEM), a particular flavor of ZFEM, has been developed based on the QTSE method described in section 3.2. The finite element computations are conducted using quaternion variables in order to compute first order derivatives with respect to up to three variables of interest ϕ_1, ϕ_2 and ϕ_3 per QFEM analysis (see Table D.2 or Table D.3). In order to perform the quaternion analysis, these variables are perturbed in specific imaginary directions, e.g.

$$\phi_1^* = \phi_1 + \boldsymbol{hi} + 0j + 0k \tag{3.25}$$

$$\phi_2^* = \phi_2 + 0i + \boldsymbol{hj} + 0k \tag{3.26}$$

$$\phi_3^* = \phi_3 + 0i + 0j + \boldsymbol{hk} \tag{3.27}$$

where the step-size for a quaternion analysis is typically $h = 10^{-17}$ or smaller times the value of the perturbed parameter for a double precision analysis due to the linear convergence of QTSE. The three perturbed

variables may be any combination of the input parameters of the problem, which may come, for instance, from Table D.2 or Table D.3. For example, to obtain the sensitivity in a heat transfer analysis of the temperature solution with respect to the thermal conductivity α, heat generation rate s and convection coefficient h_c, the variables are perturbed as: $\alpha^* = \alpha + hi$, $s^* = s + hj$ and $h_c^* = h_c + hk$. Note that the real part of the quaternion input variables corresponds to the input values of the conventional finite element analysis.

Due to the quaternion inputs, additional degrees of freedom have to be defined in order to obtain the derivative information. As an example, Figure 3.3 shows an 8-noded quaternion quadrilateral element appropriate for a thermal analysis with 4 degrees of freedom per node ($\mathbf{u}$, $\mathbf{u}_{,\phi_1}h$, $\mathbf{u}_{,\phi_2}h$ and $\mathbf{u}_{,\phi_3}h$) with a 3×3 integration scheme in natural coordinates with a node for each hypercomplex direction.

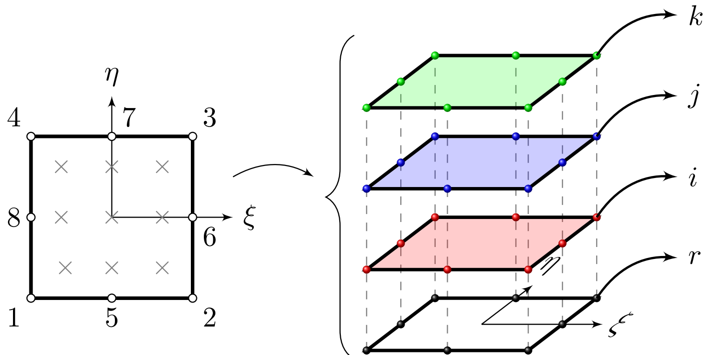


Figure 3.3: 2D 8-noded quaternion quadrilateral element with the corresponding duplicates given by the imaginary directions i, j and k in natural coordinates and the 3×3 full integration scheme. A 3D 20-noded brick element with 20 DOF can be similarly constructed.

Due to the quaternion inputs, the QFEM analysis generates a quaternion system of equations:

$$\mathbf{K}^*\mathbf{u}^* = \mathbf{f}^* \tag{3.28}$$

where $\mathbf{u}^*$ is the hypercomplexified nodal solution vector which represents either the nodal temperature if solving a heat transfer analysis or the nodal displacements if solving a linear elastic problem. Regardless of the problem being solved, the result of the QFEM analysis is composed of quaternion numbers and thus it is formed by the sum of vectors directed along imaginary directions:

$$\mathbf{u}^* = \mathbf{u}_r + \mathbf{u}_i i + \mathbf{u}_j j + \mathbf{u}_k k \tag{3.29}$$

where $\mathbf{u}_r$ is equivalent to the conventional finite element nodal solution vector (e.g. nodal temperature or

nodal displacements), $\mathbf{u}_i$ is a vector containing the derivative of nodal solution with respect to parameter ϕ_1, $\mathbf{u}_j$ is a vector with the derivative with respect to ϕ_2 and $\mathbf{u}_k$ contains the derivative with respect to ϕ_3, that is:

$$\mathbf{u}_r \approx \mathbf{u} \tag{3.30}$$

$$\mathbf{u}_i \approx h\mathbf{u}_{,\phi_1} \tag{3.31}$$

$$\mathbf{u}_j \approx h\mathbf{u}_{,\phi_2} \tag{3.32}$$

$$\mathbf{u}_k \approx h\mathbf{u}_{,\phi_3} \tag{3.33}$$

In general ZFEM, the number of imaginary directions in the hypercomplex algebra determines the number of additional vectors contained in $\mathbf{u}^*$. The relation of the imaginary components and the derivatives is dependent on the capabilities of the algebra.

As in the conventional finite element method, traditional real-valued element shape functions are used in QFEM for interpolation of the nodal solution within a finite element:

$$u^*(x, y, z) = \mathbf{N}\mathbf{u}^* \tag{3.34}$$

where $\mathbf{N}$ is the matrix of real-valued element shape functions. Consequently, the derivatives of the solution field $u(x, y, z)$ with respect to the input parameters are approximated using the shape function matrix $\mathbf{N}$ as

$$u(x, y, z) \approx \mathbf{N}\mathbf{u}_r \tag{3.35}$$

$$u_{,\phi_1}(x, y, z) \approx \frac{1}{h}\mathbf{N}\mathbf{u}_i \tag{3.36}$$

$$u_{,\phi_2}(x, y, z) \approx \frac{1}{h}\mathbf{N}\mathbf{u}_j \tag{3.37}$$

$$u_{,\phi_3}(x, y, z) \approx \frac{1}{h}\mathbf{N}\mathbf{u}_k \tag{3.38}$$

The quaternion global system matrix $\mathbf{K}^*$ is a matrix whose elements are quaternion numbers, and $\mathbf{f}^*$ is the force vector which is also formed by quaternion elements. Each $\mathbf{K}^*$ and $\mathbf{f}^*$ can be separated by its imaginary coefficients as follows:

$$\mathbf{K}^* = \mathbf{K}_r + \mathbf{K}_i i + \mathbf{K}_j j + \mathbf{K}_k k \tag{3.39}$$

$$\mathbf{f}^* = \mathbf{f}_r + \mathbf{f}_i i + \mathbf{f}_j j + \mathbf{f}_k k \tag{3.40}$$

where $\mathbf{K}_r$ and $\mathbf{f}_r$ are the conventional global system matrix and global vector of independent terms, $\mathbf{K}_i$ and $\mathbf{f}_i$ contain approximations of the derivatives with respect to ϕ_1, i.e. $h\mathbf{K}_{,\phi_1}$ and $h\mathbf{f}_{,\phi_1}$ respectively. Analogously, the matrices and vectors in j and k directions contain the derivatives with respect to parameters ϕ_2 and ϕ_3 respectively. Both $\mathbf{K}^*$ and $\mathbf{f}^*$ depend on the input parameters of the problem being solved. They are formed using an element-by-element approach as done in a traditional FE method albeit now using quaternion degrees-of-freedom.

The Jacobian matrix $\mathbf{J}$ depends on the coordinates (x, y, z) of the nodes that define the element in the original domain. Since the nodal coordinates are input parameters, they may be perturbed in any imaginary direction and thus they are quaternion-valued inputs. As a consequence, $\mathbf{J}$ becomes a matrix of quaternion numbers and is computed as follows:

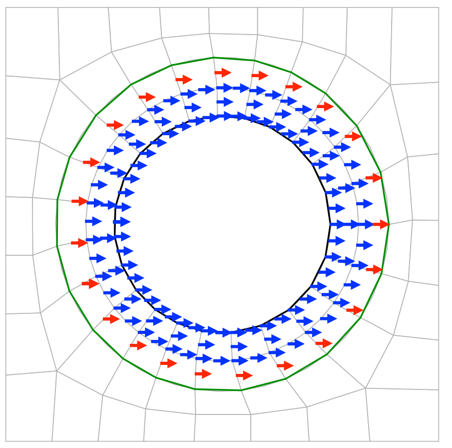
$$\mathbf{J}^* = \begin{bmatrix} N_{1,\xi} & N_{2,\xi} & \cdots \\ N_{1,\eta} & N_{2,\eta} & \cdots \\ N_{1,\gamma} & N_{2,\gamma} & \cdots \end{bmatrix} \begin{bmatrix} x_1^* & y_1^* & z_1^* \\ x_2^* & y_2^* & z_2^* \\ \vdots & \vdots & \vdots \end{bmatrix} \tag{3.41}$$

where the elemental shape functions $(N_1, N_2, ...)$ correspond to the conventional FEM shape functions for the corresponding element interpolation. The quaternion nodal coordinates are defined for the p'th node as:

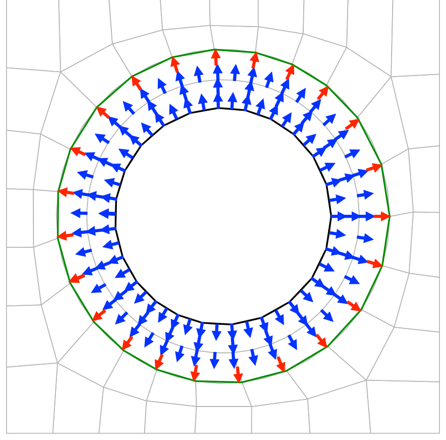
$$x_p^* = x_{p_r} + x_{pi}i + x_{pj}j + x_{pk}k, \quad y_p^* = y_{p_r} + y_{pi}i + y_{pj}j + y_{pk}k, \quad z_p^* = z_{p_r} + z_{pi}i + z_{pj}j + z_{pk}k \tag{3.42}$$

Nodal coordinates are perturbed when shape derivatives are required using the imaginary nodal coordinates [44]. For instance, consider a 2D analysis with a domain that has a circular hole. If the derivative with respect to the location of the hole is required, say in the horizontal direction, a perturbation is applied to the nodes adjacent to the circle boundary in the x coordinates as shown in Figure 3.4a. If the sensitivity with respect to the radius of the circle is required, then a perturbation is applied to all adjacent nodes in the radial direction as shown in Figure 3.4b, representing virtual displacements of the nodes in the imaginary direction.

In QFEM, for example, imaginary directions i and j can be used to compute sensitivities with respect to the location of the hole in the x and y directions respectively. Meanwhile the k imaginary direction can be used to compute the sensitivity with respect to the radius of the hole.



(a) Perturbation to obtain the sensitivity with respect to the horizontal location of the hole.



(b) Perturbation to obtain the sensitivity with respect to the radius of the hole.

Figure 3.4: Examples of the nodal perturbations to obtain shape derivatives of a circular feature in a domain. The green circle marks the perturbed region inside which all nodes are perturbed, while the blue and red arrows determine the magnitude of the perturbation, h and $h/2$ respectively.

In particular to the heat transfer problem, the element system matrix is formed by

$$\mathbf{K}^{e*} = \mathbf{K}_c^{e*} + \mathbf{K}_{h_c}^{e*} \tag{3.43}$$

where $\mathbf{K}_c^{e*}$ is the portion given by heat conduction and $\mathbf{K}_{h_c}^{e*}$ is the contribution of convection to the problem.

The element heat conduction matrix is computed as:

$$\mathbf{K}_c^{e*} = \int_{\Omega_e} \alpha^* \mathbf{B}^{*T}_H \mathbf{B}_H^* \left|\mathbf{J}^*\right| d\Omega \tag{3.44}$$

where α^* is the quaternion-valued thermal conductivity. The temperature gradient matrix $\mathbf{B}_H^*$ also becomes a matrix of quaternion numbers because it depends on the Jacobian $\mathbf{J}$:

$$\mathbf{B}_H^* = \left[\mathbf{J}^*\right]^{-1} \begin{bmatrix} N_{1,\xi} & N_{2,\xi} & \cdots \\ N_{1,\eta} & N_{2,\eta} & \cdots \\ N_{1,\gamma} & N_{2,\gamma} & \cdots \end{bmatrix} \tag{3.45}$$

The determinant of the Jacobian matrix $\mathbf{J}^*$ is computed using the standard matrix determinant operations and thus the result is a scalar quaternion value. As an example, the determinant of a two dimensional analysis with a 2×2 Jacobian matrix is calculated as follows:

$$|\mathbf{J}^*| = \begin{vmatrix} J_{11}^* & J_{12}^* \\ J_{21}^* & J_{22}^* \end{vmatrix} = J_{11}^* J_{22}^* - J_{12}^* J_{21}^* \tag{3.46}$$

In standard applications, Equation (3.44) is carried out using the Gaussian integration scheme for each element type. In the case of a quaternion implementation such integration is carried out as:

$$\mathbf{K}_c^{e*} \approx \sum_r^{n_{gp\gamma}} \sum_l^{n_{gp\eta}} \sum_p^{n_{gp\eta}} W_r W_l W_p \ \alpha^* \mathbf{B}^*{}_H(\xi_p, \eta_l, \gamma_r)^T \mathbf{B}_H^*(\xi_p, \eta_l, \gamma_r) \ |\mathbf{J}^*(\xi_p, \eta_l, \gamma_r)| \tag{3.47}$$

where W_r, W_l, W_p and (ξ_p, η_l, γ_r) are the conventional real-valued weights and evaluation points for the corresponding quadrature.

Similarly, taking into account the quaternion-valued convection coefficient h_c^*, the element convection matrix is formulated as

$$\mathbf{K}_{h_c}^{e*} \ = \ \int_{\partial\Omega_e} h_c^* \mathbf{N}^T \mathbf{N} \ |\mathbf{J}^*| \ d\Gamma \tag{3.48}$$

and is integrated using real-valued Gaussian integration weights and points.

The element force vector is composed by

$$\mathbf{f}^{e*} = \mathbf{f}_b^{e*} + \mathbf{f}_{h_c}^{e*} + \mathbf{f}_s^{e*} \tag{3.49}$$

where $\mathbf{f}_b^{e*}$ is the force vector due to a prescribed heat flux, $\mathbf{f}_{h_c}^{e*}$elemental heat conduction matrix is the force vector due to convection and $\mathbf{f}_s^{e*}$ is the force vector due to a heat source. The force vectors are defined as

$$\mathbf{f}_b^{e*} = \int_{\partial\Omega_e} b^* \mathbf{N}^T \left|\mathbf{J}^*\right| d\Gamma \tag{3.50}$$

$$\mathbf{f}_{h_c}^{e*} = \int_{\partial\Omega_e} h_c^* T_\infty^* \mathbf{N}^T \left|\mathbf{J}^*\right| d\Gamma \tag{3.51}$$

$$\mathbf{f}_s^{e*} = \int_{\Omega_e} s^* \mathbf{N}^T \left|\mathbf{J}^*\right| d\Omega \tag{3.52}$$

For the linear elastic problem, the element system matrix $\mathbf{K}^{e*}$ is formed by

$$\mathbf{K}^{e*} = \int_{\Omega_e} \mathbf{B}^{*T}_M \mathbf{D}^*_M \mathbf{B}^*_M \left|\mathbf{J}^*\right| d\Omega \tag{3.53}$$

where $\mathbf{B}^*_M$ is the strain-displacement matrix. The element force vector is composed by

$$\mathbf{f}_M^{e*} = \mathbf{f}_{\mathbf{t}}^{e*} + \mathbf{f}_{\mathbf{b}}^{e*} \tag{3.54}$$

were $\mathbf{f}_{\mathbf{t}}^{e*}$ is the force vector due to surface tractions and $\mathbf{f}_{\mathbf{b}}^{e*}$ is the force vector due to body forces which are calculated as

$$\mathbf{f}_{\mathbf{t}}^{e*} = \int_{\partial\Omega_e} \mathbf{N}^T \bar{\mathbf{t}}^* \left|\mathbf{J}^*\right| d\Gamma \tag{3.55}$$

$$\mathbf{f}_{\mathbf{b}}^{e*} = \int_{\Omega_e} \mathbf{N}^T \mathbf{b}^* \left|\mathbf{J}^*\right| d\Omega \tag{3.56}$$

3.5.2 Dependency of Intermediate Variables

Until now, this paper has discussed the hypercomplexification of all variables and intermediate variables in the finite element analysis in order to provide the capability of computing sensitivities with respect to any input variable for any analysis. However, only certain elements of the code may need to be hypercomplexified in order to provide the correct capabilities for computing the desired derivatives, and hence, increasing computational performance.

Figure 3.5a summarizes the dependency of the intermediate variables with respect to the input variables for the heat transfer problem and Figure 3.5b analogously summarizes the dependency of the intermediate

variables of the linear elastic finite element problem. The nodal temperatures $\mathbf{u}^*$, solution of the QFEM analysis, will always be hypercomplex. However, if the desired analysis does not require sensitivities with respect to shape variables nor nodal coordinates (x, y, z), then neither the Jacobian matrix $\mathbf{J}^*$ nor the gradient matrix $\mathbf{B}^*$ are required to be hypercomplexified, hence they can be used as standard real-only matrices. Moreover, e.g., if the sensitivity with respect to the heat conduction coefficient k is required, then it is mandatory that the heat conduction matrix $\mathbf{K}_c^*$ and the global system matrix $\mathbf{K}^*$ be defined as hypercomplex-typed arrays.

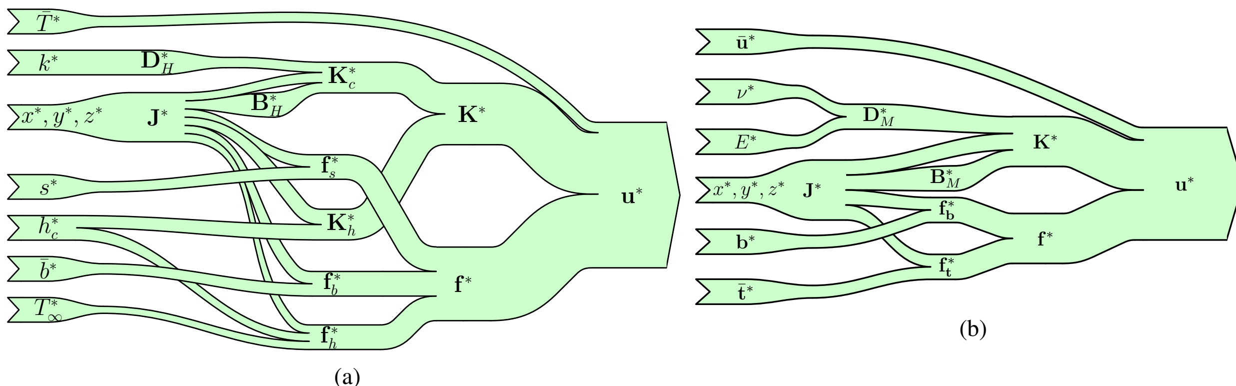


Figure 3.5: Dependency of the intermediate variables of the finite element problem with respect to the input variables for (a) heat transfer and (b) linear elasticity problems.

3.5.3 Assembly and solution of the global system of equations

The quaternion/octonion finite element method produces element matrices and vectors formed by quaternion/octonion numbers. Instead of implementing a quaternion/octonion solver, this paper describes two methods to solve the system of equations using real-based solvers: i) convert the quaternion/octonion system of equations to an equivalent Cauchy-Riemann form of all real numbers (see B.2), and then use a conventional direct solver, or ii) reduce the system of equations to lower triangular then use a block solver method, as discussed below. Both solution methods provide the same result; however, the block solver method is more efficient.

For method i) mentioned above, a single system of equations with all-real coefficients is formed based on the corresponding matrix-vector forms of quaternion/octonion numbers. The system matrix $\mathbf{K}^*$ is transformed into its matrix form and vectors $\mathbf{u}^*$ and $\mathbf{f}^*$ are transformed into vector forms, and thus an equivalent system of equations composed of all real numbers is formulated as:

$$\overbrace{\begin{bmatrix} \mathbf{K}_r & -\mathbf{K}_i & -\mathbf{K}_j & -\mathbf{K}_k \\ \mathbf{K}_i & \mathbf{K}_r & -\mathbf{K}_k & \mathbf{K}_j \\ \mathbf{K}_j & \mathbf{K}_k & \mathbf{K}_r & -\mathbf{K}_i \\ \mathbf{K}_k & -\mathbf{K}_j & \mathbf{K}_i & \mathbf{K}_r \end{bmatrix}}^{\mathbf{T}(\mathbf{K}^*)} \overbrace{\begin{bmatrix} \mathbf{u}_r \\ \mathbf{u}_i \\ \mathbf{u}_j \\ \mathbf{u}_k \end{bmatrix}}^{\mathbf{t}(\mathbf{u}^*)} = \overbrace{\begin{bmatrix} \mathbf{f}_r \\ \mathbf{f}_i \\ \mathbf{f}_j \\ \mathbf{f}_k \end{bmatrix}}^{\mathbf{t}(\mathbf{f}^*)} \tag{3.57}$$

where $\mathbf{T}(\cdot)$ and $\mathbf{t}(\cdot)$ are the matrix and vector transformations of the quaternion algebra (see B.2), respectively. Notice that this conversion can be performed at the local element level before assembly or at the global level after assembly. The resulting system of equations (3.57) is unsymmetric. However, in the case of symmetric problems, e.g. heat transfer and linear elasticity, where $\mathbf{K}_r$ is symmetric; the Cauchy-Riemann matrix of Equation (3.57) can be transformed to symmetric by changing the sign of all the elements of its upper triangular portion, Equation (3.58). This does not affect the overall result of the QFEM/OFEM analysis since the product between imaginary terms, e.g., $\mathbf{K}_i \ \mathbf{u}_j$, are of order h^2 and thus insignificant to the QFEM/OFEM analysis. Therefore, a symmetric solver is used.

$$\begin{bmatrix} \mathbf{K}_r & -\mathbf{K}_i & -\mathbf{K}_j & -\mathbf{K}_k \\ \mathbf{K}_i & \mathbf{K}_r & -\mathbf{K}_k & \mathbf{K}_j \\ \mathbf{K}_j & \mathbf{K}_k & \mathbf{K}_r & -\mathbf{K}_i \\ \mathbf{K}_k & -\mathbf{K}_j & \mathbf{K}_i & \mathbf{K}_r \end{bmatrix} \mapsto \begin{bmatrix} \mathbf{K}_r & \mathbf{K}_i & \mathbf{K}_j & \mathbf{K}_k \\ \mathbf{K}_i & \mathbf{K}_r & \mathbf{K}_k & -\mathbf{K}_j \\ \mathbf{K}_j & \mathbf{K}_k & \mathbf{K}_r & \mathbf{K}_i \\ \mathbf{K}_k & -\mathbf{K}_j & \mathbf{K}_i & \mathbf{K}_r \end{bmatrix} \tag{3.58}$$

The system of equations for a quaternion implementation forms a system of equations that is 16 times larger than the real-only system $\mathbf{K}_r\mathbf{u}_r = \mathbf{f}_r$. In the case of octonions, the equivalent system of equations is 64 times the size of the real-only system.

In the case of method ii) as mentioned above, the system of equations (3.57) can be expanded and subdivided into simpler blocks in order to optimize its solution. First, expand Equation (3.57) to form four systems of equations, then remove the terms of order h^2 as shown below

$$\begin{aligned}
\mathbf{K}_r\ \mathbf{u}_r\ -\cancelto{0}{\mathbf{K}_i}\ \mathbf{u}_i-\cancelto{0}{\mathbf{K}_j}\ \mathbf{u}_j-\cancelto{0}{\mathbf{K}_k}\ \mathbf{u}_k &= \mathbf{f}_r && (3.59)\\
\mathbf{K}_i\ \mathbf{u}_r\ +\mathbf{K}_r\ \mathbf{u}_i-\cancelto{0}{\mathbf{K}_k}\ \mathbf{u}_j+\cancelto{0}{\mathbf{K}_j}\ \mathbf{u}_k &= \mathbf{f}_i && (3.60)\\
\mathbf{K}_j\ \mathbf{u}_r\ +\cancelto{0}{\mathbf{K}_k}\ \mathbf{u}_i+\mathbf{K}_r\mathbf{u}_j-\cancelto{0}{\mathbf{K}_i}\ \mathbf{u}_k &= \mathbf{f}_j && (3.61)\\
\mathbf{K}_k\ \mathbf{u}_r\ -\cancelto{0}{\mathbf{K}_j}\ \mathbf{u}_i+\cancelto{0}{\mathbf{K}_i}\ \mathbf{u}_j+\mathbf{K}_r\mathbf{u}_k &= \mathbf{f}_k && (3.62)
\end{aligned}$$

Using the QTSE method, each coefficient in the imaginary directions i, j and k contains a factor h, e.g. $\mathbf{K}_i = h\mathbf{K}_{,\phi_1}$. Therefore, each multiplication between two imaginary components will become a term of order h^2, e.g. $\mathbf{K}_i\mathbf{u}_i = h^2\mathbf{K}_{,\phi_1}\mathbf{u}_{,\phi_1}$. As a consequence, the terms formed by the multiplication between two imaginary coefficients can be neglected such as $\mathbf{K}_i\mathbf{u}_i$ or $\mathbf{K}_j\mathbf{u}_k$. Using this reduction method, the system of equations can be rewritten as

$$\begin{aligned}
\mathbf{K}_r\mathbf{u}_r &= \mathbf{f}_r && (3.63)\\
\mathbf{K}_r\mathbf{u}_i &= \mathbf{f}_i - \mathbf{K}_i\mathbf{u}_r && (3.64)\\
\mathbf{K}_r\mathbf{u}_j &= \mathbf{f}_j - \mathbf{K}_j\mathbf{u}_r && (3.65)\\
\mathbf{K}_r\mathbf{u}_k &= \mathbf{f}_k - \mathbf{K}_k\mathbf{u}_r && (3.66)
\end{aligned}$$

where the solution of the system (3.63) solves the standard real system of equations, obtaining the conventional finite element solution $\mathbf{u}_r$. The real stiffness matrix $\mathbf{K}_r$, the real solution $\mathbf{u}_r$ and the imaginary components of $\mathbf{f}^*$ and $\mathbf{K}^*$ are used to solve the imaginary coefficients of $\mathbf{u}^*$, Equations (3.64)-(3.66). All systems (3.63)-(3.66) share the same matrix of coefficients $\mathbf{K}_r$, the conventional stiffness matrix of the finite element formulation, and thus Lower-Upper (LU) decomposition or Cholesky decomposition may be used to take advantage of the formulation with multiple right-hand sides. This approach does not require the formation of the full Cauchy-Riemann matrix form for the system. The Cauchy-Riemann form was used only to deduce the method. It only requires the matrices and vectors described in Equations (3.64)-(3.66).

3.6 Abaqus Implementation

The quaternion finite element formulations for heat transfer and linear elasticity analyses of 2D bodies were implemented within the Abaqus commercial finite element software through the implementation of user element subroutines (UELs).

To convert a traditional, real-only Abaqus finite element code into a quaternion-based one, a support library is required to define the quaternion number type and overload its operators. The minimal operations to be supported and overloaded are given in Table 3.5. Although matrix and vector functions are optional, in the general case the code is simplified by writing it in matrix form, thus these operations are included in Table 3.5.

	Operation
Scalar	Addition/Subtraction
	Multiplication/Division
	Power
Vector	Dot product
	Addition/Subtraction
Matrix	Matrix multiplication
	Matrix transpose
	Matrix inverse

Table 3.5: Basic operations required to hypercomplexify a Finite Element code.

Both approaches for solving the system of equations discussed in section 3.5.3, i) the direct Cauchy-Riemann approach and ii) the Block Solver approach were implemented in the Abaqus software.

3.6.1 Direct Cauchy-Riemann approach

The Cauchy-Riemann solution approach requires the least modification of a conventional real UEL code compared to the Block-Solver method. In this paper we extend the strategy proposed previously [44] to quaternions and octonions. To implement this approach, the traditional real Abaqus UEL code is quaternionized/octonionized by defining every intermediate variable (see Figure 3.5) as quaternion/octonion type instead of real. In addition to this, the input and output parameters should also be defined as quaternion/octonion variables. Since Abaqus is restricted to real-only parameters, the Cauchy-Riemann vector and matrix forms are used within the UEL. In this case, the UEL is defined with $4\times$ the number of degrees-of-freedom (DOFs) of the real element to contain the imaginary coefficients of the quaternion algebra and $8\times$ the num-

ber of DOFs for the octonion algebra. For example, an 8-noded quadrilateral element with one DOF per node will become a 32-noded element (see Figure 3.3) with a total of 32-DOFs; each DOF has 4 values at each node, e.g. u, $\partial u/\partial \phi_1$, $\partial u/\partial \phi_2$, $\partial u/\partial \phi_3$ for quaternions. The real coefficients of the nodes and their degrees of freedom correspond to the traditional FEM. Each additional imaginary DOF contains derivative information with respect to the perturbed input parameters. In Abaqus, increasing the number of DOFs automatically generates a system with $4\times$ the number of equations and thus a matrix of $16\times$ the size of that of the real analysis for quaternions and with $8\times$ the number of equations and $64\times$ the matrix size of the real analysis for the case of octonions.

Increasing the number of DOFs requires the definition of additional nodes, which therefore requires the definition of the corresponding coordinates. Since all new nodes correspond to imaginary directions, the coordinate values of the added nodes correspond to the perturbation applied to the real associated node in the corresponding imaginary direction in x, y and z respectively (see Figure 3.4). Other input parameters such as the modulus of elasticity, thermal conductivity, etc., are defined as input parameters to the UEL containing its imaginary coefficients as well. Each input parameter is converted from a set of 4/8 independent values to quaternion/octonion variables at the start of the UEL. Boundary conditions are also expanded to 4/8 inputs per condition each, where consistency is required while associating boundary conditions to the corresponding imaginary nodes.

After the quaternionized/octonionized UEL computes the stiffness matrix and load vectors, the UEL converts the system matrix and load vector to the real-only Cauchy-Riemann corresponding forms following the structure defined in Equation (3.57) with the symmetric matrix as defined in Equation (3.58). Afterwards, no further modification to the Abaqus conventional routines is required, thus the assembly procedure is performed by conventional Abaqus calls and the solution is found using conventional real-only solvers.

After the analysis, post-processing is required to collect the derivative information, which is contained within the nodal solutions of the DOFs associated to the imaginary nodes as described before.

3.6.2 Block-Solver approach

As previously discussed in Section 3.5.3, the quaternion/octonion system of equations can be reduced to lower triangular and then subdivided into multiple subsystems. This approach results in a more efficient implementation than the direct Cauchy-Riemann method.

The implementation of the block solver solution scheme within Abaqus is composed of two steps. First,

an initial step is performed as a conventional finite element analysis to solve the real system of equations, Equation (3.63). Then, a second and final step is used to form and solve the subsequent systems (see Equations (3.64)-(3.66)), using the "multiple load case analysis" in Abaqus, which is used to study the linear response of a structure subjected to distinct sets of loads and boundary conditions. In the second step, the UEL uses the solution from the previous step ($\mathbf{u}_{Re}$) and quaternion/octonion algebra to form the right hand sides of the system of equations. The total number of load cases (right hand sides) corresponds to the number of imaginary directions of the algebra, three for quaternions and seven for octonions. Each load case is uniquely associated to a right hand side and thus to a unique imaginary direction of the algebra. The boundary conditions of a particular load case represent the perturbation of that boundary condition in the corresponding imaginary direction. Perturbations to the nodal coordinates are defined as field variables and the perturbations to the modulus of elasticity, thermal conductivity, etc., are defined as input parameters to the UEL.

The advantage of using the multiple load case approach is that Abaqus shares the stiffness matrix for all right hand sides. This is possible due to the fact that systems (3.64)-(3.66) share the same stiffness matrix, $\mathbf{K}_{\text{Re}}$. Notice that both steps generate systems with the same number of DOFs as the conventional real-only finite element analysis.

This approach results in significant advantages over the direct Cauchy-Riemann approach where the number of DOFs grows $4\times$ (quaternions) and $8\times$ (octonions); and the system of equations grows to $16\times$ (quaternions) and $64\times$ (octonions) the size of the real-only system. Then the system is assembled element by element, in which each element matrix is transformed to its Cauchy-Riemann equivalent system.

3.7 Numerical Examples

3.7.1 Finite Element Abaqus example: Hollow Cylinder with Internal Heat Generation and Convective Surfaces

A two dimensional axisymmetric thermal sensitivity analysis was conducted for a hollow cylinder with internal heat generation in which the inner and outer surfaces are under convective conditions. This example possesses an analytical solution presented by [65]. Also, [34] presented a complex-valued FEM for sensitivity analysis of thermoelastic problems in which the same example was analyzed using CTSE. For each sensitivity, a complex-valued finite element analysis was required. However, using a quaternion/octonion

based FEM, three/seven different sensitivities can be obtained in a single analysis. The quaternion and octonion FEM were implemented using an Abaqus User Element (UEL).

The non-dimensional analytical temperature through the thickness of the cylinder presented by [65] is defined as:

$$\hat{\theta}(R) = \frac{T(R) - T_i}{T_i - T_o} = -\frac{1}{4}\hat{s}R^2 + \frac{\rho \ln(R)}{4} K_c + L_c \tag{3.67}$$

where R is a normalized radius defined as the ratio between the radial coordinate r, and the inner radius r_i, i.e. $R = r/r_i$. R lies within the interval $[1, \rho]$, where ρ is defined as r_o/r_i with r_o as the outer radius. The term $\hat{s}$ in Equation (3.67) is a normalized heat generation parameter, defined as

$$\hat{s} = \frac{s r_i^2}{\alpha(T_i - T_o)} \tag{3.68}$$

where T_i and T_o are the temperature of the fluids at the inner and outer surfaces, respectively. Also, the terms K_c and L_c from Equation (3.67) are given by:

$$K_c = \frac{2Bi_o\hat{s} - Bi_iBi_o\hat{s} + 2Bi_i\hat{s}\rho - 4Bi_iBi_o + Bi_iBi_o\hat{s}\rho^2}{Bi_iBi_o\rho\ln(\rho) + Bi_i + Bi_o\rho} \tag{3.69}$$

$$L_c = \frac{1}{4}\frac{2\hat{s}\rho^2 - 4Bi_o\rho + Bi_o\hat{s}\rho^3 - 2Bi_o\hat{s}\rho\ln(\rho) + Bi_iBi_o\hat{s}\rho\ln(\rho) - 2\hat{s} + Bi_i\hat{s}}{Bi_iBi_o\rho\ln(\rho) + Bi_i + Bi_o\rho} \tag{3.70}$$

where Bi_i and Bi_o are the Biot numbers at the inner and outer surfaces, defined as follows:

$$Bi_i = \frac{h_{ci} r_i}{\alpha} \tag{3.71}$$

$$Bi_o = \frac{h_{co} r_i}{\alpha} \tag{3.72}$$

where h_{ci} and h_{co} are the convection coefficients at the inner and outer surfaces, respectively.

The analyses were computed using Abaqus 6.12, and were performed using a computer containing a compute node with an Intel(R) Xeon(R) CPU E5-2650 v2 processor @ 2.60GHz and 377.80 GB of RAM. User element subroutines were implemented to perform the analyses. For the CFEM case, the user element used the standard complex type of the Fortran language and for the QFEM and OFEM user element subroutines, two libraries based on operator overloading were developed and used as described in section

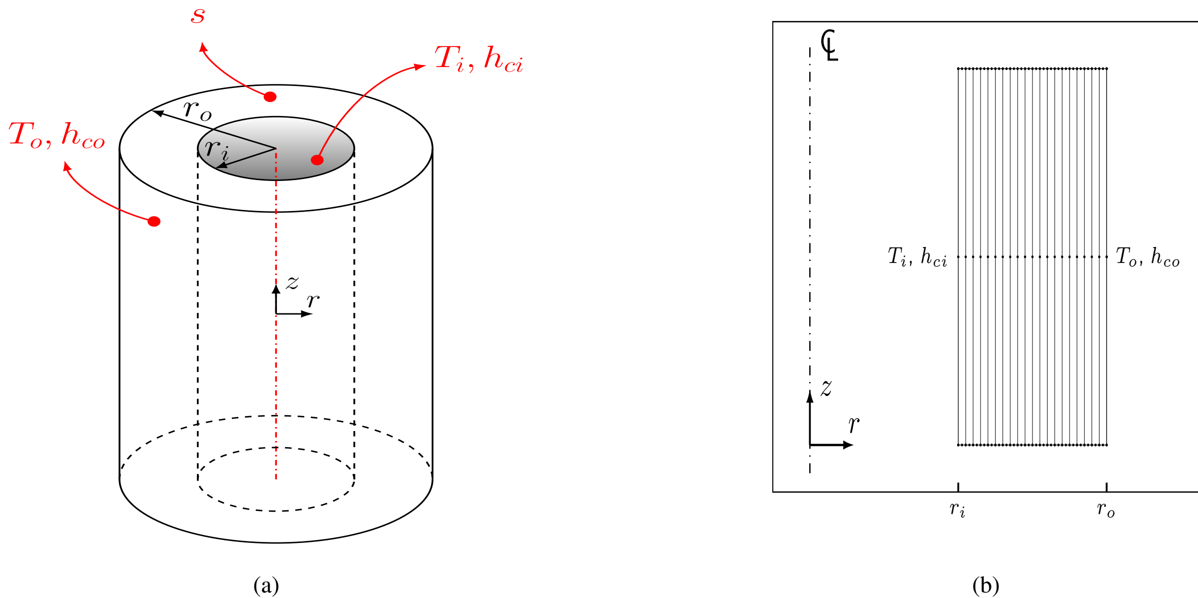


Figure 3.6: (a) Problem schematic (b) Finite element mesh for axysimmetric model

3.6.

3.7.1.1 Results

The cylinder was modeled using a QFEM 8-noded 32-DOF axisymmetric element. The non-dimensional parameters were $\rho = 1.5$, $Bi_i = 0.1$, $Bi_o = 1$, $T_o/T_i = 0.7$ and $\hat{s} = 5$. The finite element model contained a total of 20 32-noded (8 real and 24 imaginary) axisymmetric elements. A schematic of the problem and the finite element mesh are shown in Figure 3.6.

Figure 3.7 shows the comparison of QFEM against the values obtained using the analytical solution for the normalized temperature, $\hat{\theta}$, and the three requested derivatives, $\partial\hat{\theta}/\partial h_{ci}$, $\partial\hat{\theta}/\partial h_{co}$ and $\partial\hat{\theta}/\partial T_i$. When using octonions, in which seven imaginary directions are available, seven sensitivities were computed in a single OFEM analysis. In this case, additional to the QFEM sensitivities, four extra derivatives, $\partial\hat{\theta}/\partial T_o$, $\partial\hat{\theta}/\partial r_o$, $\partial\hat{\theta}/\partial\hat{s}$ and $\partial\hat{\theta}/\partial\alpha$, were obtained. The sensitivity results are the same for complex, quaternion or octonion FE analyses.

3.7.2 Finite Element Abaqus Example: Cantilever Beam

The purpose of this analysis is to compare the performance of the methods for solving the system of equations, direct Cauchy-Riemann versus block-solver approach. Therefore, a two dimensional elastic analysis

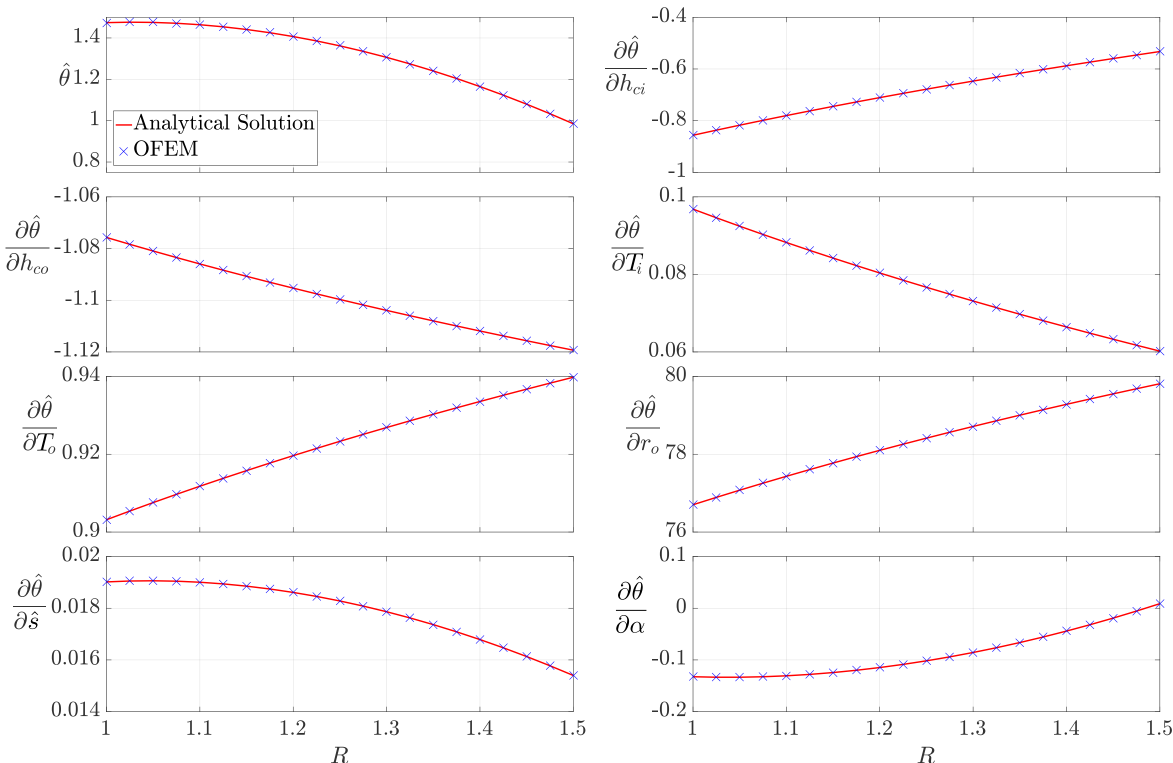


Figure 3.7: CFEM, QFEM and OFEM dimensionless nodal temperature sensitivities with respect to h_{ci}, h_{co}, T_i, T_o, r_0, $\hat{s}$ and α.

was performed for a cantilever beam fixed at the left end and subjected to body forces defined by accelerations in the x and y directions and a contact force at the unsupported end as shown in Figure 3.8.

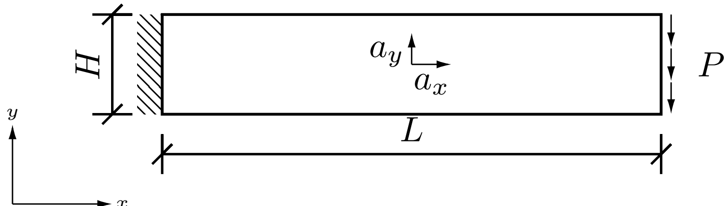


Figure 3.8: Sketch of a cantilever beam anchored at the left end with a vertical load at the right end.

A set of CFEM, QFEM and OFEM analyses was performed in order to compute one, three and seven derivatives respectively, with different meshes ranging from 320 to 1,310,720 elements. The following derivatives were computed: Derivative of the displacement vector field $\mathbf{u}$ (u_x and u_y) with respect to: the modulus of elasticity E, the Poisson ratio ν, the accelerations a_x and a_y, the applied load P, the height H and the length L of the beam. All derivatives were computed using a step size $h = 10^{-17}$ for all algebras.

The analyses were computed using Abaqus 6.12, and were performed using a computer containing a

compute node with an Intel(R) Xeon(R) CPU E5-2650 v2 processor @ 2.60GHz and 377.80 GB of RAM. User element subroutines were implemented to perform the analyses. For the CFEM case, the user element used the standard complex type of the Fortran language and for the QFEM and OFEM user element subroutines, two libraries based on operator overloading were developed and used as described in section 3.6.

3.7.2.1 Results

The performance results for both CPU time and memory consumption are summarized in Figure 3.9. The results are normalized with respect to the CPU and memory consumption of a conventional real analysis for the same mesh, with no derivative computation. The direct Cauchy-Riemann (CR) solution was both the most time and memory consuming approach. The CR approach constantly increased the compute time relative to a real-only analysis as the number of degrees-of-freedom is increased. For example, an OFEM analysis using a CR approach may take longer than $100\times$ the CPU time and $100\times$ the memory of a real-only solution.

The Block Solver (BS) approach, on the other hand, is far more efficient than the CR method. To compute one derivative, CFEM using the BS approach used the least computational resources of up to $2\times$ the CPU time and up to 4% extra memory compared to a real-only analysis. To compute three derivatives, QFEM used up to $2.8\times$ the CPU time and 6% of the memory of a real-only analysis. To compute seven derivatives, OFEM used up to $3.3\times$ the CPU time and 13% of the memory of a real-only analysis (see Figure 3.9c and 3.9d).

Table 3.6 summarizes the computational expense relative to a real-only analysis on a per derivative basis. Using CFEM, computing one derivative takes up to 95% of the time of the real solution. Using QFEM, the derivatives are computed using only 58% the time of the real solution and using OFEM the derivatives are computed using only 32% the time of the real solution. OFEM is most efficient on a per derivative basis because the cost of solving the real system is amortized across a larger number of derivatives.

In Table 3.7, the number of Complex (C), Quaternion (Q) and Octonion (O) finite element analyses required to compute up to fourteen derivatives and the relative computational expense compared to a real-only analysis is shown for each case. The results were obtained for a mesh containing a total of 81920 elements. For multiple derivatives, CFEM is the most inefficient because only one derivative can be computed per complex analysis. However, if only one derivative is required by the user, a CFEM analysis is most efficient.

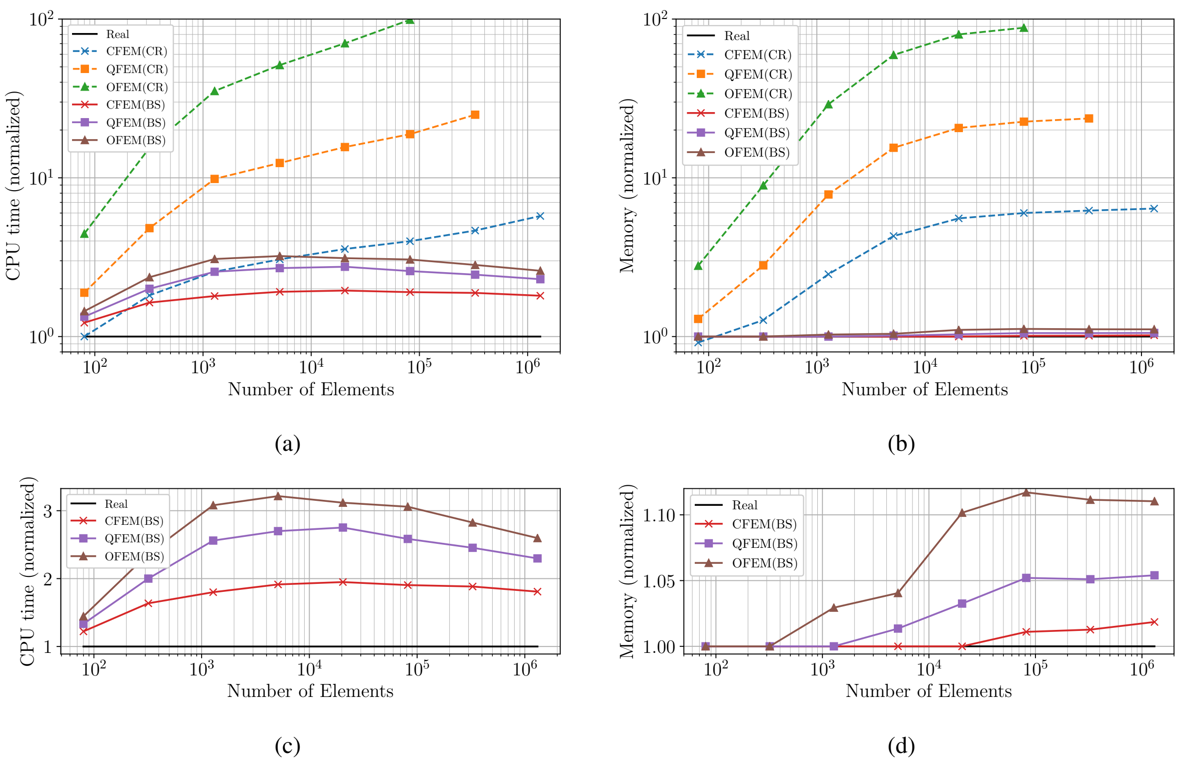


Figure 3.9: Performance results of the linear elastic problem with different mesh densities showing (a) CPU time and (b) memory normalized with respect to real analysis of CFEM (1 derivative), QFEM (3 derivatives) and OFEM (7 derivatives) approaches using both the direct Cauchy Riemann (CR) approach and the block-solver (BS) method. A closeup is presented showing the normalized (c) CPU time and (d) memory of the block-solver approach only.

	Block-Solver			Cauchy-Riemann		
Number of elements	**CFEM**	**QFEM**	**OFEM**	**CFEM**	**QFEM**	**OFEM**
320	0.64	0.33	0.20	0.82	1.27	2.07
1280	0.80	0.52	0.30	1.56	2.94	4.89
5120	0.91	0.56	0.32	2.07	3.80	7.18
20480	0.95	0.58	0.30	2.56	4.87	9.91
81920	0.90	0.53	0.29	2.99	5.94	14.00
327680	0.88	0.49	0.26	3.65	7.99	N/A
1310720	0.81	0.43	0.23	4.75	N/A	N/A

Table 3.6: CPU time overhead per derivative (1 for CFEM, 3 for QFEM and 7 for OFEM) normalized to a real analysis.

QFEM is the most efficient method to compute 2 or 3 derivatives, whereas OFEM is most efficient if 4-7 derivatives are required.

For more than seven derivatives, a combination of analyses is required. For example, to compute ten derivatives, one can use 10 CFEM, 4 QFEM, or 2 OFEM analyses; however, the optimal approach is to use 1 QFEM and 1 OFEM. In that case, the run time will be five times longer than a real-only analysis. To compute eleven derivatives, the optimal approach is to use 2 OFEM analyses whereas only 11 are requested. The column marked optimal shows the combination of analyses that can compute the required number of derivatives in minimum computer time.

Number of	**Number of Analysis Required**				**Relative Computational Cost**			
Derivatives	**CFEM**	**QFEM**	**OFEM**	**Optimal**	**CFEM**	**QFEM**	**OFEM**	**Optimal**
1	1C	1Q	1O	1C	1.90	2.59	3.03	1.90
2	2C	1Q	1O	1Q	3.80	2.59	3.03	2.59
3	3C	1Q	1O	1Q	5.70	2.59	3.03	2.59
4	4C	2Q	1O	1O	7.60	5.18	3.03	3.03
5	5C	2Q	1O	1O	9.50	5.18	3.03	3.03
6	6C	2Q	1O	1O	11.40	5.18	3.03	3.03
7	7C	3Q	1O	1O	13.30	7.77	3.03	3.03
8	8C	3Q	2O	1C+1O	15.20	7.77	6.06	4.93
9	9C	3Q	2O	1Q+1O	17.10	7.77	6.06	5.62
10	10C	4Q	2O	1Q+1O	19.00	10.36	6.06	5.62
11	11C	4Q	2O	2O	20.90	10.36	6.06	6.06
12	12C	4Q	2O	2O	22.80	10.36	6.06	6.06
13	13C	5Q	2O	2O	24.70	12.95	6.06	6.06
14	14C	5Q	2O	2O	26.60	12.95	6.06	6.06

Table 3.7: Computational cost relative to a real-only analysis required to compute derivatives using a block-solver solution scheme for a mesh of 81920 elements.

Figure 3.10 shows the most efficient combination of analyses to compute up to 28 derivatives for the same FE mesh. The most efficient sequence of analyses is the one that uses the most number of full OFEM analyses, this is, computing 7 derivatives per run. Then, the remaining derivatives are computed using an extra CFEM analysis if one more derivative is required, an extra QFEM analysis if 2-3 derivatives are required or an extra OFEM analysis if 4-7 derivatives are required. For example, to compute 23 derivatives, three OFEM analyses compute 21 derivatives and then the remaining 2 derivatives are computed using a QFEM analysis.

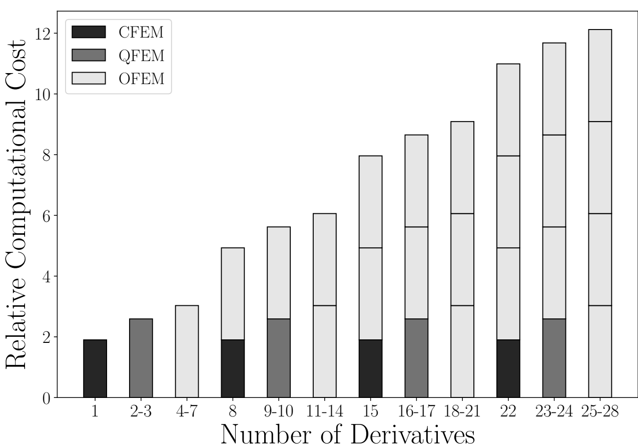


Figure 3.10: Optimal selection of analyses type based on the required number of derivatives for a mesh of 81920 elements.

3.8 Discussion

The complex Taylor series expansion method (CTSE) was extended to Cayley-Dickson algebras (CDTSE) in order to compute multiple first order derivatives in a single analysis. In particular, quaternion (3 derivatives) and octonion (7 derivatives) - based examples were demonstrated. It is well known that a property is lost in each doubling of the algebra, commutativity is lost with quaternion algebra, associativity is lost with octonion, etc. However, the loss of these properties does not affect CDTSE in that for small step sizes the effect of the loss of properties is contained in the higher order terms and these terms are driven to below machine precision through the use of a sufficiently small step size.

Although the method is analogous to CTSE, the truncation error for the derivative is of order h, not h^2, as in CTSE. This is clearly seen in Figure 3.2b. The implications are that a smaller step size is needed for QTSE or OTSE than CTSE. However there are no numerical implications in practice as the truncation error can be driven below machine precision in all cases through the use of a sufficiently small step size.

Figure 3.2b also shows that the relative error for octonions is slightly higher than quaternions, however preserving the same convergence rate. This is explained by the increase in the number of operations octonions will perform in comparison with quaternions. Sedenions will have a larger truncation error than octonions, etc. However, this is an academic topic since the truncation error can be driven below machine precision in practice because reducing the step size h does not generate subtraction cancellation errors.

Both the direct Cauchy-Riemann and block-solver approaches solved for the same value and number of

outputs. The block-solver method is a better approach in terms of its computational performance because it required several orders of magnitude less CPU time and memory than the direct approach. The block solver method lies within the same order of magnitude of CPU performance metrics as the conventional real analysis, even though the number of derivatives increased up to a number of 7 in the OFEM case. The direct method, however, was easier to implement since it was all performed within one step of analysis. However, the paradigm of the implementation in a commercial software such as Abaqus is still open, since the multiple right-hand-side solvers are usually available only to linear analyses.

Using the block-solver method, QFEM shows a better performance per derivative than CFEM. QFEM is at least 1.54$\times$ faster per derivative than CFEM. In the case of OFEM, it is at least 1.71$\times$ faster per derivative than QFEM. However, the total CPU time of QFEM is longer than CFEM. Similarly, the total CPU time of OFEM is slower than QFEM. It is therefore more efficient to use CFEM to compute one derivative, QFEM to compute 2-3 derivatives and OFEM to compute 4-7. It is expected that higher dimensional algebras, e.g. sedenions, will be more efficient for larger number of derivatives, than multiple CFEM, QFEM or OFEM analyses.

The Taylor series method for computing functions of quaternions and octonions is a convenient tool because the required operations are the function and its real-derivative evaluated at the real value. Hence, a formal comprehensive method of computing functions of quaternions and octonions is not required.

The proposed methodology is far superior in run times and memory consumption than equivalent methods using multi-complex or multi-dual implementations. However, CDTSE is limited to first order derivatives, unlike multi-complex or multi-dual implementations which can compute first, second, or higher order derivatives.

3.9 Conclusions

The complex Taylor series expansion (CTSE) method has been extended to Cayley-Dickson algebras (CDTSE) in order to compute multiple first order derivatives. In particular, quaternion algebra can compute 3 first order derivatives, octonion algebra can compute 7, and sedenion algebra can compute 15. This strategy can be extended to any number of first order derivatives in powers of 2.

Similarly to CTSE, CDTSE does not require subtraction of terms in order to compute a derivative. As a result, the truncation error can be driven below machine precision through the use of a very small step size. Contrary to CTSE, the error for CDTSE is of order h whereas for CTSE the error is of order

h^2. However, this difference in error is of no practical importance once a sufficiently small h is used for numerical computations.

Quaternion and octonion algebras were implemented within the commercial finite element code Abaqus as a user element in an analogous manner as CTSE. The key elements were to a) define a set of imaginary nodes added to each real node, b) use the coordinates of the imaginary nodes to define the perturbation in the geometry, c) compute the element stiffness matrix using mathematical operations that support the Cayley-Dickson algebra (quaternion, octonion, etc.), d) convert the stiffness matrix to all real values before returning the results to Abaqus, e) solve the system of equations using the Abaqus built-in direct solver or use the block solver approach, and f) post process the derivatives of the degrees-of-freedom to obtain auxiliary results.

The conversion of a traditional real valued finite element routine to quaternion and octonion algebra was facilitated through the definition of quaternion and octonion types, implementing operator overloading of addition/subtraction, multiplication/division, and the implementation of an approximate but accurate method for computing functions of quaternions/octonions using a truncated Taylor series. The truncated Taylor series approach was shown to have the same accuracy as the traditional quaternion/octonion functions but it was simpler to program and faster to execute.

The accuracy, versatility and performance of CDTSE in the finite element method was shown in two numerical examples implemented in Abaqus through the use of user element subroutines. The use of QFEM and OFEM user elements allowed one to compute accurate three and seven first order derivatives respectively in heat transfer and linear elasticity problems. Two solution methods were discussed and compared: a direct Cauchy Riemann and a block solver approach. The block solver approach was significantly more efficient than the direct approach, although both resulted in the same output values.

Chapter 4: ORDER TRUNCATED IMAGINARY ALGEBRA

Current hypercomplex algebras capable of computing high-order multivariable derivatives, multicomplex [30] and multidual [8] numbers, suffer from exponential increase in their size, while the number of derivatives to be computed does not match such growing rate. Lessons were learned from the development and implementation of the QFEM and OFEM methods (see chapter 3). It is possible to optimize the computation of multiple first order derivatives with respect to traditional multicomplex/multidual analyses, by using every imaginary direction in the algebra to compute first order derivatives with respect to an independent variable. The size of quaternions and octonions, however, may not match the required number of derivatives to be computed, and as shown in Table 3.7, and optimal computational performance may require the use of different algebras as well as multiple evaluations of the computational model. In general, the effective number of derivatives being computed does not scale in the same way the size of the current algebras grow, implying that some computations may not be required and so additional resources are unnecessarily spent.

Analyzing the deficiencies current hypercomplex algebras, the steps toward a more efficient hypercomplex algebra were identified and led to the development of a new hypercomplex algebra called Order Truncated Imaginary (OTI) numbers. This section is dedicated to introduce OTI algebra, which is capable of computing derivatives without repeating computations and can adapt according to the required order and number of variables of the derivatives being analyzed. The result is a step-size independent method to compute derivatives that is capable of computing high order multivariable derivatives with high accuracy (up to machine precision) in a single function evaluation. The equivalent OTI matrix form was developed. It is lower-triangular, sparse and smaller in size compared to equivalent multidual/multicomplex forms. The matrix form allows a natural integration of the algebra with linear algebra procedures. A support library was developed to provide a computational interface to the OTI algebra.

4.1 Motivation

Simplicity of the methodology and easiness of implementation are strong points in favor of hypercomplex differentiation methods. However, large problems where many derivatives are required, e.g. 3-dimensional shape optimization and crack propagation analysis, may not be best suited for these algebras due to the inefficiency given by the exponential growth of the size of their algebras to compute high order multivariable derivatives.

A simple analysis to current algebras shows that there are steps toward a more efficient hypercomplex algebra. Consider $x^* \in \mathbb{D}_2$ a bidual number as follows:

$$x^* = x_0 + \epsilon_1 + \epsilon_2 + 0\epsilon_1\epsilon_2 \tag{4.1}$$

Evaluating functions $f(x^*)$ and $g(x^*)$ will result in bidual number,

$$f(x^*) = f(x_0) + f_{,x}(x_0)\epsilon_1 + f_{,x}(x_0)\epsilon_2 + f_{,x^2}(x_0)\epsilon_1\epsilon_2 \tag{4.2}$$

$$g(x^*) = g(x_0) + g_{,x}(x_0)\epsilon_1 + g_{,x}(x_0)\epsilon_2 + g_{,x^2}(x_0)\epsilon_1\epsilon_2 \tag{4.3}$$

By multiplying $f(x^*)$ and $g(x^*)$ in an equivalent matrix-vector form results in the following:

$$\mathbf{T}\left(f(x^*)\right)\mathbf{t}\left(g(x^*)\right) = \begin{bmatrix} f(x_0) & 0 & 0 & 0 \\ f_{,x}(x_0) & f(x_0) & 0 & 0 \\ f_{,x}(x_0) & 0 & f(x_0) & 0 \\ f_{,x^2}(x_0) & f_{,x}(x_0) & f_{,x}(x_0) & f(x_0) \end{bmatrix} \begin{Bmatrix} g(x_0) \\ g_{,x}(x_0) \\ g_{,x}(x_0) \\ g_{,x^2}(x_0) \end{Bmatrix} \tag{4.4}$$

where the result of the coefficient in the $\epsilon_1\epsilon_2$ direction is

$$\mathrm{Im}_{\epsilon_1\epsilon_2}(f(x^*)g(x^*)) = \underbrace{f_{,x^2}(x_0)}_{(\epsilon_1\epsilon_2)}\ \underbrace{g(x_0)}_{(1)} + \underbrace{f_{,x}(x_0)}_{(\epsilon_2)}\ \underbrace{g_{,x}(x_0)}_{(\epsilon_1)} + \underbrace{f_{,x}(x_0)}_{(\epsilon_1)}\ \underbrace{g_{,x}(x_0)}_{(\epsilon_2)} + \underbrace{f(x_0)}_{(1)}\ \underbrace{g_{,x^2}(x_0)}_{(\epsilon_1\epsilon_2)} \tag{4.5}$$

where each term shows below its corresponding imaginary direction source.

Notice that the term $f_{,x}(x_0)\, g_{,x}(x_0)$ is repeated twice, although it was formed by the multiplication of different imaginary coefficients from $f(x^*)$ and $g(x^*)$. Equation 4.5 can be simplified as

$$\mathrm{Im}_{\epsilon_1\epsilon_2}(f(x^*)g(x^*)) = f_{,x^2}(x_0)\, g(x_0) + \mathbf{2}f_{,x}(x_0)\, g_{,x}(x_0) + f(x_0)g_{,x^2}(x_0) \tag{4.6}$$

This shows the operation can be achieved with a single computation of the term $f_{,x}(x_0)\, g_{,x}(x_0)$, and hence $f(x^*)$ and $g(x^*)$ should provide only one coefficient of the first order derivative. Therefore, two

modifications are necessary:

i) if term $f_{,x}(x_0)\, g_{,x}(x_0)$ is to be computed only once, it needs to be formed by multiplying the coefficient of ϵ_1 by itself (because it comes from multiplying two first order directions). However, current multidual algebra conditions do not allow this because $\epsilon_1^2 = 0$. Therefore, ϵ_1^2 must not be truncated.

ii) a factor must be added to guarantee the correct formulation of equation 4.6 (in this case 2).

If item ii is applied as in equation (4.6), the matrix form becomes

$$\begin{bmatrix} f(x_0) & 0 & 0 \\ f_{,x}(x_0) & f(x_0) & 0 \\ f_{,x^2}(x_0) & \mathbf{2} f_{,x}(x_0) & f(x_0) \end{bmatrix} \begin{Bmatrix} g(x_0) \\ g_{,x}(x_0) \\ g_{,x^2}(x_0) \end{Bmatrix} \tag{4.7}$$

noting that the first order derivatives, $f_{,x}(x_0)$, appear with different factors in the matrix. This requires that the matrix formation and algebra operations track every coefficient and factor for every computation, which is expensive. It is more convenient to avoid this by creating a more homogeneous representation such that every term appears with the same factor in the matrix form. Therefore, dividing equation 4.6 by 2 and reforming the matrix form, all first and second order terms have homogeneous facotrs, with the first order derivatives having a factor of 1 and second order derivatives a factor of $1/2$. The matrix system becomes

$$\begin{bmatrix} f(x_0) & 0 & 0 \\ f_{,x}(x_0) & f(x_0) & 0 \\ f_{,x^2}(x_0)/2 & f_{,x}(x_0) & f(x_0) \end{bmatrix} \begin{Bmatrix} g(x_0) \\ g_{,x}(x_0) \\ g_{,x^2}(x_0)/2 \end{Bmatrix} \tag{4.8}$$

This shows that it is possible to optimize current hypercomplex algebras (multiduals and multicomplex) to compute derivatives avoiding repetition of coefficients and thus avoiding unnecessary calculations.

4.2 Order Truncated Imaginary Numbers

The hypercomplex Order Truncated Imaginary (OTI) algebra is formed by a tuple of coefficients associated to a set of imaginary directions constructed by the multiplication of imaginary basis ϵ_1, ϵ_2, etc. In OTI algebra, the imaginary condition truncates those directions with order greater than a specified value n, instead of considering the basis nilpotent as multiduals do. The order correspond to the sum of basis exponents in the imaginary term. For instance, if truncation order $n = 3$, then $\epsilon_1^2\epsilon_2$ and ϵ_4^3 are a non-zero, but ϵ_5^4 is zero,

whereas multidual conditions would have zeroed these directions.

Formally, an OTI number a^* is a N-tuple of real coefficients $a_{\boldsymbol{\alpha}_s^p}$ associated with a set of imaginary directions $\boldsymbol{\alpha}_s^p$ ($s = 1, \ldots, N_m^p$; $p = 0, \ldots, n$); where p is the order of the imaginary direction, N_m^p is the number of imaginary directions of order p with m bases and s is the unique index of the imaginary direction among directions of order p. Similar to multiduals, a basis ϵ_l is an imaginary unit that has no "divisors" other than 1 and itself, in a sense analogous to the concept of prime numbers. Also similar to multiduals, imaginary directions are formed by multiplication of bases. However, in contrast to multiduals, OTI algebra allows imaginary directions formed by bases with multiplicity higher than 1, e.g. directions ϵ_1^2 and $\epsilon_1^2\epsilon_2^3$ may be different than zero. A general form of an OTI imaginary direction is:

$$\boldsymbol{\alpha}_s^p = \epsilon_1^{\kappa_{1s}^p}\epsilon_2^{\kappa_{2s}^p}\cdots\epsilon_m^{\kappa_{ms}^p} = \prod_{l=1}^{m}\epsilon_l^{\kappa_{ls}^p} \text{ with } \kappa_{ls}^p \in \mathbb{N}, p = \sum_{l=1}^{m}\kappa_{ls}^p \tag{4.9}$$

where m is the number of bases, κ_{ls}^p is the multiplicity of basis ϵ_l for direction $\boldsymbol{\alpha}_s^p$. The order p of an imaginary direction corresponds to the sum of the multiplicities of its bases. The imaginary direction $\boldsymbol{\alpha}_1^0 = 1$ represents the real direction, similar to other hypercomplex algebras (where $N_m^0 = 1$).

Table 4.1 shows a subset of imaginary directions for OTI algebra, where s corresponds to the index of the imaginary direction among all imaginary directions of order p. The table can be easily extrapolated to larger orders and larger indices.

Order p	Index s										
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	$\cdots$
1	ϵ_1	ϵ_2	ϵ_3	ϵ_4	ϵ_5	ϵ_6	ϵ_7	ϵ_8	ϵ_9	ϵ_{10}	$\cdots$
2	ϵ_1^2	$\epsilon_1\epsilon_2$	ϵ_2^2	$\epsilon_1\epsilon_3$	$\epsilon_2\epsilon_3$	ϵ_3^2	$\epsilon_1\epsilon_4$	$\epsilon_2\epsilon_4$	$\epsilon_3\epsilon_4$	ϵ_4^2	$\cdots$
3	ϵ_1^3	$\epsilon_1^2\epsilon_2$	$\epsilon_1\epsilon_2^2$	ϵ_2^3	$\epsilon_1^2\epsilon_3$	$\epsilon_1\epsilon_2\epsilon_3$	$\epsilon_2^2\epsilon_3$	$\epsilon_1\epsilon_3^2$	$\epsilon_2\epsilon_3^2$	ϵ_3^3	$\cdots$
4	ϵ_1^4	$\epsilon_1^3\epsilon_2$	$\epsilon_1^2\epsilon_2^2$	$\epsilon_1\epsilon_2^3$	ϵ_2^4	$\epsilon_1^3\epsilon_3$	$\epsilon_1^2\epsilon_2\epsilon_3$	$\epsilon_1\epsilon_2^2\epsilon_3$	$\epsilon_2^3\epsilon_3$	$\epsilon_1^2\epsilon_3^2$	$\cdots$
5	ϵ_1^5	$\epsilon_1^4\epsilon_2$	$\epsilon_1^3\epsilon_2^2$	$\epsilon_1^2\epsilon_2^3$	$\epsilon_1\epsilon_2^4$	ϵ_2^5	$\epsilon_1^4\epsilon_3$	$\epsilon_1^3\epsilon_2\epsilon_3$	$\epsilon_1^2\epsilon_2^2\epsilon_3$	$\epsilon_1\epsilon_2^3\epsilon_3$	$\cdots$
6	ϵ_1^6	$\epsilon_1^5\epsilon_2$	$\epsilon_1^4\epsilon_2^2$	$\epsilon_1^3\epsilon_2^3$	$\epsilon_1^2\epsilon_2^4$	$\epsilon_1\epsilon_2^5$	ϵ_2^6	$\epsilon_1^5\epsilon_3$	$\epsilon_1^4\epsilon_2\epsilon_3$	$\epsilon_1^3\epsilon_2^2\epsilon_3$	$\cdots$
7	ϵ_1^7	$\epsilon_1^6\epsilon_2$	$\epsilon_1^5\epsilon_2^2$	$\epsilon_1^4\epsilon_2^3$	$\epsilon_1^3\epsilon_2^4$	$\epsilon_1^2\epsilon_2^5$	$\epsilon_1\epsilon_2^6$	ϵ_2^7	$\epsilon_1^6\epsilon_3$	$\epsilon_1^5\epsilon_2\epsilon_3$	$\cdots$
8	ϵ_1^8	$\epsilon_1^7\epsilon_2$	$\epsilon_1^6\epsilon_2^2$	$\epsilon_1^5\epsilon_2^3$	$\epsilon_1^4\epsilon_2^4$	$\epsilon_1^3\epsilon_2^5$	$\epsilon_1^2\epsilon_2^6$	$\epsilon_1\epsilon_2^7$	ϵ_2^8	$\epsilon_1^7\epsilon_3$	$\cdots$
9	ϵ_1^9	$\epsilon_1^8\epsilon_2$	$\epsilon_1^7\epsilon_2^2$	$\epsilon_1^6\epsilon_2^3$	$\epsilon_1^5\epsilon_2^4$	$\epsilon_1^4\epsilon_2^5$	$\epsilon_1^3\epsilon_2^6$	$\epsilon_1^2\epsilon_2^7$	$\epsilon_1\epsilon_2^8$	ϵ_2^9	$\cdots$
10	ϵ_1^{10}	$\epsilon_1^9\epsilon_2$	$\epsilon_1^8\epsilon_2^2$	$\epsilon_1^7\epsilon_2^3$	$\epsilon_1^6\epsilon_2^4$	$\epsilon_1^5\epsilon_2^5$	$\epsilon_1^4\epsilon_2^6$	$\epsilon_1^3\epsilon_2^7$	$\epsilon_1^2\epsilon_2^8$	$\epsilon_1\epsilon_2^9$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Table 4.1: OTI imaginary directions $\boldsymbol{\alpha}_s^p$ in terms of multiplication of imaginary bases. For instance, $\boldsymbol{\alpha}_6^8 = \epsilon_1^3\epsilon_2^5$ and $\boldsymbol{\alpha}_{10}^9 = \epsilon_2^9$.

Multidual algebras truncate the exponent of each basis from 2 onwards, i.e., $\boldsymbol{\alpha}_s^p = 0$ if $\kappa_{ls}^p \geq 2$. In

contrast, OTI algebra truncates imaginary directions whose order exceeds a specified value n:

$$\boldsymbol{\alpha}_s^p : \begin{cases} \neq 0, & \text{if } p \leq n \\ = 0, & \text{otherwise} \end{cases} \tag{4.10}$$

OTI directions are commutative, e.g. $\epsilon_1\epsilon_3^2$, $\epsilon_3\epsilon_1\epsilon_3$ and $\epsilon_3^2\epsilon_1$ are equivalent. In order to form an OTI number, a set of coefficients ($a_{\boldsymbol{\alpha}_s^p}; s = 1, \ldots, N^p; p = 0, \ldots, n$) is associated to the set of imaginary directions. $\mathbb{OTI}_m^n$ denotes the set of OTI numbers of m bases and truncation order n, and is represented as follows:

$$a^* = \sum_{p=0}^{n} \sum_{s=1}^{N_m^p} a_{\boldsymbol{\alpha}_s^p} \, \boldsymbol{\alpha}_s^p, \; a^* \in \mathbb{OTI}_m^n \tag{4.11}$$

where $a_{\boldsymbol{\alpha}_s^p} \in \mathbb{R}$ is a real coefficient associated to $\boldsymbol{\alpha}_s^p$. The total number of coefficients N^p with order p and m bases is given by

$$N^p = \binom{p+m-1}{p} = \binom{p+m-1}{m-1} = \frac{(p+m-1)!}{(m-1)!\,p!} \tag{4.12}$$

In section 4.2.4, it will be shown that the total number of bases m and truncation order n are defined by the number of variables and order of derivatives to be calculated in an analysis, respectively. These two parameters determine the total number of coefficients N of a particular OTI number, given by:

$$N = \binom{m+n}{m} = \frac{(m+n)!}{m!\,n!} = \sum_{p=0}^{n} N^p \tag{4.13}$$

It is convenient to group terms of OTI numbers by order p. Let $[a^*]^p$ contain all terms of order p from OTI number a^*. Thus, a^* can be re-written in the form

$$a^* = \sum_{p=0}^{n} [a^*]^p, \text{ where } [a^*]^p = \sum_{s=1}^{N_m^p} a_{\boldsymbol{\alpha}_s^p} \, \boldsymbol{\alpha}_s^p \tag{4.14}$$

Examples of OTI numbers are shown below.

Example 1. OTI number with truncation order $n = 3$ and number of bases $m = 1$, $\mathbb{OTI}_1^3$:

For this number, the total number of coefficients is found via equation (4.13), which gives $N = 4$. Using equation (4.12), the number of coefficients per order is $N^0 = 1$, $N^1 = 1$, $N^2 = 1$ and $N^3 = 1$. Considering

the infinite ways to form imaginary directions from a given number of bases, order truncation limits the available imaginary directions to only 4: those that have order less than 3.

$$a^* = a_r + a_{\epsilon_1}\epsilon_1 + a_{\epsilon_1^2}\epsilon_1^2 + a_{\epsilon_1^3}\epsilon_1^3 + \cancel{a_{\epsilon_1^4}\epsilon_1^4 + a_{\epsilon_1^5}\epsilon_1^5 + ...} \tag{4.15}$$

$$\therefore a^* = a_r + a_{\epsilon_1}\underbrace{\epsilon_1}_{\boldsymbol{\alpha}_1^1} + a_{\epsilon_1^2}\underbrace{\epsilon_1^2}_{\boldsymbol{\alpha}_1^2} + a_{\epsilon_1^3}\underbrace{\epsilon_1^3}_{\boldsymbol{\alpha}_1^3} \tag{4.16}$$

Notice that the terms organized by order are given by:

$$[a^*]^0 = a_r, \quad [a^*]^1 = a_{\epsilon_1}\epsilon_1, \quad [a^*]^2 = a_{\epsilon_1^2}\epsilon_1^2, \quad [a^*]^3 = a_{\epsilon_1^3}\epsilon_1^3 \tag{4.17}$$

Example 2. OTI number with truncation order $n = 2$ and number of bases $m = 3$, $\mathbb{OTI}_3^2$:

The total number of coefficients is $N = 10$, with $N^0 = 1$, $N^1 = 3$ and $N^2 = 6$:

$$a^* = a_r + a_{\epsilon_1}\underbrace{\epsilon_1}_{\boldsymbol{\alpha}_1^1} + a_{\epsilon_2}\underbrace{\epsilon_2}_{\boldsymbol{\alpha}_2^1} + a_{\epsilon_3}\underbrace{\epsilon_3}_{\boldsymbol{\alpha}_3^1} + a_{\epsilon_1^2}\underbrace{\epsilon_1^2}_{\boldsymbol{\alpha}_1^2} + a_{\epsilon_1\epsilon_2}\underbrace{\epsilon_1\epsilon_2}_{\boldsymbol{\alpha}_2^2} + a_{\epsilon_2^2}\underbrace{\epsilon_2^2}_{\boldsymbol{\alpha}_3^2}$$
$$+ a_{\epsilon_1\epsilon_3}\underbrace{\epsilon_1\epsilon_3}_{\boldsymbol{\alpha}_4^2} + a_{\epsilon_2\epsilon_3}\underbrace{\epsilon_2\epsilon_3}_{\boldsymbol{\alpha}_5^2} + a_{\epsilon_3^2}\underbrace{\epsilon_3^2}_{\boldsymbol{\alpha}_6^2} \tag{4.18}$$

where by the notation from equation 4.14, the number can be written as

$$[a^*]^0 = a_r \tag{4.19}$$

$$[a^*]^1 = a_{\epsilon_1}\epsilon_1 + a_{\epsilon_2}\epsilon_2 + a_{\epsilon_3}\epsilon_3 \tag{4.20}$$

$$[a^*]^2 = a_{\epsilon_1^2}\epsilon_1^2 + a_{\epsilon_1\epsilon_2}\epsilon_1\epsilon_2 + a_{\epsilon_2^2}\epsilon_2^2 + a_{\epsilon_1\epsilon_3}\epsilon_1\epsilon_3 + a_{\epsilon_2\epsilon_3}\epsilon_2\epsilon_3 + a_{\epsilon_2^2}\epsilon_2^2 \tag{4.21}$$

The following sections will define the algebraic operations of OTI numbers: scalar addition and scalar multiplication. It can be shown that with these operations OTI numbers form a vector space.

4.2.1 Addition

Addition of two OTI numbers a^* and b^* is operated by adding the coefficients of identical imaginary directions, that is:

$$a^* + b^* = \sum_{p=0}^{n} \sum_{s=1}^{N_m^p} \left(a_{\boldsymbol{\alpha}_s^p} + b_{\boldsymbol{\alpha}_s^p} \right) \boldsymbol{\alpha}_s^p \tag{4.22}$$

As it can be seen, OTI addition is commutative.

Example 3. Addition of two OTI numbers.

Consider two OTI numbers as follows

$$a^* = a_r + a_{\epsilon_1}\epsilon_1 + a_{\epsilon_1^2}\epsilon_1^2 + a_{\epsilon_1^3}\epsilon_1^3,\ a^* \in \mathbb{OTI}_1^3 \tag{4.23}$$

$$b^* = b_r + b_{\epsilon_1}\epsilon_1 + b_{\epsilon_1^2}\epsilon_1^2 + b_{\epsilon_1^3}\epsilon_1^3,\ b^* \in \mathbb{OTI}_1^3 \tag{4.24}$$

then the addition operation $c^* = a^* + b^*$, the result is given below

$$c^* = a^* + b^* = (a_r + b_r) + (a_{\epsilon_1} + b_{\epsilon_1})\,\epsilon_1 + \left(a_{\epsilon_1^2} + b_{\epsilon_1^2} \right) \epsilon_1^2 + \left(a_{\epsilon_1^3} + b_{\epsilon_1^3} \right) \epsilon_1^3,\ a^*, b^*, c^* \in \mathbb{OTI}_1^3 \tag{4.25}$$

4.2.2 Multiplication

Given two arbitrary imaginary directions, $\boldsymbol{\alpha}_s^p$ and $\boldsymbol{\alpha}_r^q$, the multiplication is defined by adding the exponents of the same basis. As a result, multiplication of $\boldsymbol{\alpha}_s^p$ and $\boldsymbol{\alpha}_r^q$ result in an OTI direction $\boldsymbol{\alpha}_t^o$ defined as follows:

$$\boldsymbol{\alpha}_t^o = \boldsymbol{\alpha}_s^p \boldsymbol{\alpha}_r^q = \left(\prod_{l=1}^{m} \epsilon_l^{\kappa_{ls}^p} \right) \left(\prod_{l=1}^{m} \epsilon_l^{\kappa_{lr}^q} \right) = \prod_{l=1}^{m} \epsilon_l^{(\kappa_{ls}^p + \kappa_{lr}^q)} = \prod_{l=1}^{m} \epsilon_l^{\kappa_{lt}^o} \tag{4.26}$$

The order of the resulting imaginary direction (o) corresponds to the addition of the orders of the multiplicands. The index t of the resulting imaginary is extracted from the corresponding sequence as in Table 4.1 for order o. Notice that multiplying the real direction $\boldsymbol{\alpha}_1^0 = 1$ times any other imaginary direction $\boldsymbol{\alpha}_s^p$ results in $\boldsymbol{\alpha}_s^p$. Also note that the multiplication of two imaginary directions $\boldsymbol{\alpha}_r^q \boldsymbol{\alpha}_s^p$ always result in an imaginary direction of order greater than any of the orders of the multiplicands when $p \neq 0$ and $q \neq 0$. Consequently,

if the resulting order $o = q + p$ is greater than the truncation order n of the algebra, then the result will be truncated. For that case, multiplication can be avoided explicitly. Examples of multiplication between OTI directions are shown below.

$$\left(\epsilon_1^3\right)\left(\epsilon_1^2\right) = \epsilon_1^{3+2} = \epsilon_1^5 \tag{4.27}$$

$$\left(\epsilon_1\right)\left(\epsilon_3^3\right) = \epsilon_1^{1+0}\,\epsilon_3^{0+3} = \epsilon_1\epsilon_3^3 \tag{4.28}$$

$$\left(\epsilon_1\epsilon_2^2\right)\left(\epsilon_2\epsilon_3\right) = \epsilon_1^{1+0}\,\epsilon_2^{2+1}\,\epsilon_3^{0+1} = \epsilon_1\epsilon_2^3\epsilon_3 \tag{4.29}$$

Multiplication of two OTI numbers is defined as

$$c^* = a^*b^* = \sum_{q=0}^{n}\sum_{r=1}^{N_m^q}\left(\sum_{p=0}^{n-q}\sum_{s=1}^{N_m^p} a_{\boldsymbol{\alpha}_r^q} b_{\boldsymbol{\alpha}_s^p} \boldsymbol{\alpha}_r^q \boldsymbol{\alpha}_s^p\right),\ \text{for } a^*, b^*, c^* \in \mathbb{OTI}_m^n \tag{4.30}$$

Multiplication of OTI numbers is commutative and distributive:

$$a^*(b^* + c^*) = (b^* + c^*)a^* = a^*b^* + a^*c^* \tag{4.31}$$

As a particular case, the multiplication of a real number times an OTI number reduces to

$$ab^* = \sum_{p=0}^{n}\sum_{s=1}^{N_m^p} a b_{\boldsymbol{\alpha}_s^p} \boldsymbol{\alpha}_s^p,\ \text{for } a \in \mathbb{R},\ b^* \in \mathbb{OTI}_m^n \tag{4.32}$$

Using the notation as in equation (4.14), the multiplication between OTI numbers can be equivalently written as

$$c^* = a^*b^* = \sum_{q=0}^{n}\sum_{p=0}^{n-q}[a^*]^q[b^*]^p \tag{4.33}$$

Example 4. Multiplication of two OTI numbers.

As an example of multiplication between two OTI numbers, consider numbers a^* and b^* as

$$a^* = a_r + a_{\epsilon_1}\epsilon_1 + a_{\epsilon_2}\epsilon_2 + a_{\epsilon_1^2}\epsilon_1^2 + a_{\epsilon_1\epsilon_2}\epsilon_1\epsilon_2 + a_{\epsilon_2^2}\epsilon_2^2,\ a^* \in \mathbb{OTI}_2^2 \tag{4.34}$$

$$b^* = b_r + b_{\epsilon_1}\epsilon_1 + b_{\epsilon_2}\epsilon_2 + b_{\epsilon_1^2}\epsilon_1^2 + b_{\epsilon_1\epsilon_2}\epsilon_1\epsilon_2 + b_{\epsilon_2^2}\epsilon_2^2,\ b^* \in \mathbb{OTI}_2^2 \tag{4.35}$$

the multiplication results in an OTI number $c^* \in \mathbb{OTI}_2^2$ as follows.

$$c^* = a^*b^* = a_rb_r + (a_rb_{\epsilon_1} + a_{\epsilon_1}b_r)\,\epsilon_1 + (a_rb_{\epsilon_2} + a_{\epsilon_2}b_r)\,\epsilon_2 + \left(a_rb_{\epsilon_1^2} + a_{\epsilon_1}b_{\epsilon_1} + a_{\epsilon_1^2}b_r\right)\epsilon_1^2$$
$$+ (a_rb_{\epsilon_1\epsilon_2} + a_{\epsilon_1}b_{\epsilon_2} + a_{\epsilon_2}b_{\epsilon_1} + a_{\epsilon_1\epsilon_2}b_r)\,\epsilon_1\epsilon_2 + \left(a_rb_{\epsilon_2^2} + a_{\epsilon_2}b_{\epsilon_2} + a_{\epsilon_2^2}b_r\right)\epsilon_2^2 \tag{4.36}$$

4.2.3 Evaluation of elementary functions

Elementary functions of OTI variables such as power, sine, cosine, logarithm, exponential function, etc; are evaluated following Taylor Series Expansion (TSE), extending the method developed in section 3.4 to OTI algebra. A similar methodology was presented in [53] that generalized the evaluation of multidual and multicomplex elementary functions for the particular case of computation of derivatives using a truncated TSE of the function.

For the case of single-variable functions such as sine, logarithm, etc., the TSE is as follows.

$$f(z) = f(a) + f_{,z}(a)\,(z-a) + \frac{1}{2!}f_{,z^2}(a)\,(z-a)^2 + \frac{1}{3!}f_{,z^3}(a)\,(z-a)^3 + \text{H.O.T} \tag{4.37}$$

Replacing the evaluation points z and a, by a^* and a_r respectively, the equivalent function is given by:

$$f(a^*) = f(a_r) + f_{,z}(a_r)(a^* - a_r) + \frac{1}{2!}f_{,z^2}(a_r)(a^* - a_r)^2 + ... + \frac{1}{n!}f_{,z^n}(a_r)(a^* - a_r)^n \tag{4.38}$$

where the result is an OTI number with the same number of basis and truncation order as the input OTI number.

Notice that the value $a^* - a_r$ is an OTI number formed by only the imaginary elements of a^*. Hence, $a^* - a_r$ is nilpotent to order $n+1$, naturally truncating the TSE of the function. Also note that derivatives $f_{,z}(a_r)$, $f_{,z^2}(a_r)$, etc; are the conventional real derivatives of the function evaluated at the real coefficient

a_r.

This method can be extended to two or more variable functions, such as the generalized power function x^y, following a multivariable TSE.

Example 5. Power function.

Consider the power function $f(x) = x^{100}$, with its p'th derivative given by:

$$f_{,x^p}(x) = \begin{cases} \frac{100!}{(100-p)!}\, x^{100-p} & p \leq 100 \\ 0 & \text{otherwise} \end{cases} \tag{4.39}$$

Consider an OTI number $a^* \in \mathbb{OTI}_2^2$, with truncation order $n = 2$ and $m = 2$ bases,

$$a^* = a_r + a_{\epsilon_1}\epsilon_1 + a_{\epsilon_2}\epsilon_2 + a_{\epsilon_1^2}\epsilon_1^2 + a_{\epsilon_1\epsilon_2}\epsilon_1\epsilon_2 + a_{\epsilon_2^2}\epsilon_2^2 \tag{4.40}$$

The evaluation of $f(a^*) = (a^*)^{100}$ following equation (4.38) gives the following expression:

$$a^{*100} = {a_r}^{100} + \overbrace{100{a_r}^{99}}^{f_{,x}} \left(a_{\epsilon_1}\epsilon_1 + a_{\epsilon_2}\epsilon_2 + a_{\epsilon_1^2}\epsilon_1^2 + a_{\epsilon_1\epsilon_2}\epsilon_1\epsilon_2 + a_{\epsilon_2^2}\epsilon_2^2\right) + \overbrace{4950{a_r}^{98}}^{f_{,x^2}/\,2!} (a_{\epsilon_1}\epsilon_1 + a_{\epsilon_2}\epsilon_2)^2 \tag{4.41}$$

Note that the term next to the second order derivative only has the terms of a^* with order 1. This is because terms with order 2, e.g. ϵ_1^2, when multiplied by any non-real imaginary direction will result in an imaginary direction with order 3 or higher, naturally truncated for $a^* \in \mathbb{OTI}_2^2$ and can be avoided explicitly. The operation $(a_{\epsilon_1}\epsilon_1 + a_{\epsilon_2}\epsilon_2)^2$ is performed using conventional multiplication rules defined in section 4.2.2. The final result is

$$\begin{aligned} a^{*100} = {a_r}^{100} + 100{a_r}^{99}a_{\epsilon_1}\epsilon_1 + 100{a_r}^{99}a_{\epsilon_2}\epsilon_2 + \left(100{a_r}^{99}a_{\epsilon_1^2} + 4950{a_r}^{98}a_{\epsilon_1}^2\right)\epsilon_1^2 + \\ \left(100{a_r}^{99}a_{\epsilon_1\epsilon_2} + 9900{a_r}^{98}a_{\epsilon_1}a_{\epsilon_2}\right)\epsilon_1\epsilon_2 + \left(100{a_r}^{99}a_{\epsilon_2^2} + 4950{a_r}^{98}a_{\epsilon_2}\right)\epsilon_2^2 \end{aligned} \tag{4.42}$$

Example 6. Sine function.

Consider the sine function $f(x) = \sin(x)$. Its p'th order derivative, $f_{,x^p}$, is given by

$$f_{,x^p} = \sin\left(x + p\frac{\pi}{2}\right) \tag{4.43}$$

In order to evaluate $f(a^*)$, for $a^* \in \mathbb{OTI}_2^2$ as in equation (4.40), equation (4.38) results in:

$$\sin(a^*) = \sin(a_r) + \overbrace{\cos(a_r)}^{f_{,x}}\left(a_{\epsilon_1}\epsilon_1 + a_{\epsilon_2}\epsilon_2 + a_{\epsilon_1^2}\epsilon_1^2 + a_{\epsilon_1\epsilon_2}\epsilon_1\epsilon_2 + a_{\epsilon_2^2}\epsilon_2^2\right) \overbrace{-\frac{\sin(a_r)}{2}}^{f_{,x^2}/2!}(a_{\epsilon_1}\epsilon_1 + a_{\epsilon_2}\epsilon_2)^2 \tag{4.44}$$

After reorganizing the terms in equation (4.44), the result is

$$\sin(a^*) = \sin(a_r) + \cos(a_r)a_{\epsilon_1}\epsilon_1 + \cos(a_r)a_{\epsilon_2}\epsilon_2 + \left(\cos(a_r)a_{\epsilon_1^2} - \frac{\sin(a_r)}{2}a_{\epsilon_1}^2\right)\epsilon_1^2 + \\ \left(\cos(a_r)a_{\epsilon_1\epsilon_2} - \sin(a_r)a_{\epsilon_1}a_{\epsilon_2}\right)\epsilon_1\epsilon_2 + \left(\cos(a_r)a_{\epsilon_2^2} - \frac{\sin(a_r)}{2}a_{\epsilon_2}^2\right)\epsilon_2^2 \tag{4.45}$$

4.2.4 Computation of Derivatives with OTI Numbers

The process to compute derivatives of real functions using OTI numbers is similar to that of dual and Complex Taylor Series Expansion methods. The method consist on hypercomplexifying the input parameters of the function whose derivatives are wanted with OTI numbers. The hypercomplexification process consists in perturbing the real value of the input variable of intetrest in an OTI imaginary direction. The resulting input OTI value is selected to have truncation order equal to the maximum order of derivative required.

Consider a univariate real function $f : \mathbb{R} \to \mathbb{R}$ and its Taylor Series Expansion (TSE):

$$f(x) = f(a) + f_{,x}(a)(x-a) + \frac{1}{2!}f_{,x^2}(a)(x-a)^2 + \frac{1}{3!}f_{,x^3}(a)(x-a)^3 + \text{H.O.T} \tag{4.46}$$

The first step is to hypercomplexify the input x of the function by perturbing its real value x_0 with an OTI imaginary direction,

$$x^* = x_0 + \epsilon_1,\ x^* \in \mathbb{OTI}_1^n \tag{4.47}$$

The value x_0 corresponds to the value at which derivatives will be evaluated. Then, selecting the center point $a = x_0$, the following expression is obtained:

$$f(x^*) = f(x_0 + \epsilon_1) = f(x_0) + f_{,x}(x_0)\epsilon_1 + \frac{1}{2!} f_{,x^2}(x_0)\epsilon_1^2 + \cdots + \frac{1}{n!} f_{,x^n}(x_0)\epsilon_1^n \tag{4.48}$$

Notice that the resulting expression is truncated for terms $n + 1$ due to the truncation order n of the number.

As a consequence, evaluating any function at the OTI number as perturbed according to equation (4.47), will result in an OTI number whose coefficients contain the derivatives of $f(x)$ evaluated at the real coefficient of the input OTI number. For this case, ϵ_1 is directly associated to variable x. The truncation order of x^* matches the maximum order of derivative to be obtained.

In contrast to equation (4.46), equation (4.48) has finite terms due to the OTI truncation order n. Since the result is an OTI number containing function derivatives, they can be recovered as shown below.

$$f_{,x^p}(x_0) = p! \, \mathrm{Im}_{\epsilon_1^p}[\, f(x^*) \,] \tag{4.49}$$

note that the order p of the OTI imaginary direction ϵ_1^p matches the corresponding order of derivative obtained $f_{,x^p}$. Note that no subtractions are performed to compute the p'th order derivative, thus the result is free of subtraction cancellation error.

This concept is extensible to multivariable functions. Consider the m variable function $f : \mathbb{R}^{(m)} \to \mathbb{R}$, $f(\mathbf{x})$ with $\mathbf{x} = \{x_1, x_2, ..., x_m\}^T$. Analogous to the single variable case, the function is expanded using its TSE and its input variables are perturbed as $x_l^* = x_l + \epsilon_l,\ x_l^* \in \mathbb{OTI}_m^n,\ l = 1, ..., m$, as follows

$$\mathbf{x}^* = \begin{Bmatrix} x_1^* \\ x_2^* \\ \vdots \\ x_m^* \end{Bmatrix} = \begin{Bmatrix} x_1 + \epsilon_1 \\ x_2 + \epsilon_2 \\ \vdots \\ x_m + \epsilon_m \end{Bmatrix} = \begin{Bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{Bmatrix} + \begin{Bmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_m \end{Bmatrix} = \mathbf{x} + \boldsymbol{\epsilon} \tag{4.50}$$

Notice that each variable is perturbed in one imaginary direction of order 1, different for each variable. That is, each variable is associated to one basis. Evaluating $f(\mathbf{x}^*)$ produces the following OTI number:

$$f(\mathbf{x}^*) = f(\mathbf{x}) + \frac{1}{1!}\nabla f(\mathbf{x}) : \boldsymbol{\epsilon} + \frac{1}{2!}\nabla^2 f(\mathbf{x}) : \boldsymbol{\epsilon}^2 + \frac{1}{3!}\nabla^3 f(\mathbf{x}) : \boldsymbol{\epsilon}^3 + \ldots + \frac{1}{n!}\nabla^n f(\mathbf{x})\boldsymbol{\epsilon}^n \tag{4.51}$$

Since the chosen representation is compact it requires a brief description: $\nabla f(\mathbf{x})$ is the well known gradient of the function f evaluated at $\mathbf{x}$, i.e. $(\nabla \otimes f)(\mathbf{x})$. The symbol $\otimes$ represents the tensor outer product. Expression $\nabla f(\mathbf{x}) : \boldsymbol{\epsilon}$ represents the Frobenius inner product between tensors. In this case. $\nabla f(\mathbf{x})$ and $\boldsymbol{\epsilon}$ are first order tensors, therefore the product is equivalent to the dot product:

$$\nabla f(\mathbf{x}) : \boldsymbol{\epsilon} = \begin{Bmatrix} f_{,x_1} \\ f_{,x_2} \\ \vdots \\ f_{,x_m} \end{Bmatrix} : \begin{Bmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_m \end{Bmatrix} = f_{,x_1}\epsilon_1 + f_{,x_2}\epsilon_2 + ... + f_{,x_m}\epsilon_m \tag{4.52}$$

In addition, $\nabla^2 f(\mathbf{x})$ is the Hessian matrix of f evaluated at $\mathbf{x}$, i.e. $(\nabla \otimes (\nabla \otimes f))(\mathbf{x})$[1]. The term $\boldsymbol{\epsilon}^2$ summarizes the tensor outer product $\boldsymbol{\epsilon}^2 = \boldsymbol{\epsilon} \otimes \boldsymbol{\epsilon} = \boldsymbol{\epsilon}\boldsymbol{\epsilon}^T$. Therefore, the term $\nabla^2 f(\mathbf{x}) : \boldsymbol{\epsilon}^2$ becomes

$$\nabla^2 f(\mathbf{x}) : \boldsymbol{\epsilon}^2 = \begin{bmatrix} f_{,x_1^2} & f_{,x_1x_2} & \cdots & f_{,x_1x_m} \\ f_{,x_2x_1} & f_{,x_2^2} & \cdots & f_{,x_2x_m} \\ \vdots & \vdots & \ddots & \vdots \\ f_{,x_mx_1} & f_{,x_mx_2} & \cdots & f_{,x_m^2} \end{bmatrix} : \begin{bmatrix} \epsilon_1^2 & \epsilon_1\epsilon_2 & \cdots & \epsilon_1\epsilon_m \\ \epsilon_2\epsilon_1 & \epsilon_2^2 & \cdots & \epsilon_2\epsilon_m \\ \vdots & \vdots & \ddots & \vdots \\ \epsilon_m\epsilon_1 & \epsilon_m\epsilon_2 & \cdots & \epsilon_m^2 \end{bmatrix} \tag{4.53}$$

$$= f_{,x_1^2}\epsilon_1^2 + 2f_{,x_1x_2}\epsilon_1\epsilon_2 + \ldots + 2f_{,x_1x_m}\epsilon_1\epsilon_m + f_{,x_2^2}\epsilon_2^2 + \ldots + 2f_{,x_2x_m}\epsilon_2\epsilon_m + \ldots + f_{,x_m^2}\epsilon_m^2 \tag{4.54}$$

This is extensible to the other terms of Equation (4.51). Therefore $\nabla^3 f(\mathbf{x})$ is the tensor of third order derivatives, i.e. $(\nabla \otimes (\nabla \otimes (\nabla \otimes f)))(\mathbf{x})$; and $\boldsymbol{\epsilon}^3$ is equivalent to $\boldsymbol{\epsilon}^3 = \boldsymbol{\epsilon} \otimes \boldsymbol{\epsilon} \otimes \boldsymbol{\epsilon}$, a third order tensor containing all imaginary directions of order 3.

For example, consider a two variable function $f(x, y)$ with perturbation as in equation (4.50).

$$x^* = x_0 + \epsilon_1,\ x^* \in \mathbb{OTI}_2^3 \tag{4.55}$$

$$y^* = y_0 + \epsilon_2,\ y^* \in \mathbb{OTI}_2^3 \tag{4.56}$$

[1] In this document, expression $\nabla^2 f(\mathbf{x})$ does not represent the Laplacian of f. The Laplacian operator is represented as $\Delta f(\mathbf{x}) = \nabla \cdot (\nabla f(\mathbf{x}))$

both OTI numbers with truncation order $n = 3$. This perturbation generates the following expansion according to equation (4.51)

$$
\begin{aligned}
f(x^*, y^*) = f(x_0, y_0) + \frac{1}{1!} f_{,x}(x_0, y_0)\epsilon_1 + \frac{1}{1!} f_{,y}(x_0, y_0)\epsilon_2 + \\
\frac{1}{2!} f_{,x^2}(x_0, y_0)\epsilon_1^2 + \frac{1}{1!1!} f_{,xy}(x_0, y_0)\epsilon_1\epsilon_2 + \frac{1}{2!} f_{,y^2}(x_0, y_0)\epsilon_2^2 + \frac{1}{3!} f_{,x^3}(x_0, y_0)\epsilon_1^3 + \\
\frac{1}{2!1!} f_{,x^2y}(x_0, y_0)\epsilon_1^2\epsilon_2 + \frac{1}{1!2!} f_{,xy^2}(x_0, y_0)\epsilon_1\epsilon_2^2 + \frac{1}{3!} f_{,y^2}(x_0, y_0)\epsilon_2^3
\end{aligned}
\quad (4.57)
$$

Notice that equation (4.57) generates all derivatives of f up to order 3 for all two variables, with no repeated derivatives. In order to get each derivative, the coefficient of the imaginary direction with the basis multiplicity that matches the number of times the function was differentiated with respect to its associated variable is extracted and then multiplied by the corresponding factor in the TSE.

$$f_{,xy}(x_0, y_0) = (1!1!)\ \text{Im}_{\epsilon_1\epsilon_2}[\, f(x^*, y^*)\,] \quad (4.58)$$

$$f_{,x^2y}(x_0, y_0) = (2!1!)\ \text{Im}_{\epsilon_1^2\epsilon_2}[\, f(x^*, y^*)\,] \quad (4.59)$$

$$f_{,y^3}(x_0, y_0) = (3!)\ \text{Im}_{\epsilon_2^3}[\, f(x^*, y^*)\,] \quad (4.60)$$

In general, any p'th order derivative of the function f with respect to variables x_1, x_2, ..., x_m; is extracted in the following manner:

$$\frac{\partial^p f}{\partial {x_1}^{\kappa_{1s}^p} \partial {x_2}^{\kappa_{2s}^p} \ldots \partial {x_m}^{\kappa_{ms}^p}} = f_{,{x_1}^{\kappa_{1s}^p} {x_2}^{\kappa_{2s}^p} \ldots {x_m}^{\kappa_{ms}^p}} = \left(\prod_{l=1}^{m} \kappa_{ls}^p ! \right) \text{Im}_{\boldsymbol{\alpha}_s^p}[\, f(x_1^*, x_2^*, ..., x_m^*)\,] \quad (4.61)$$

where $\boldsymbol{\alpha}_s^p$ is the imaginary direction with the basis multiplicity that matches the number of times the function was differentiated with respect to its associated variable, i.e., $\boldsymbol{\alpha}_s^p = \epsilon_1^{\kappa_{1s}^p} \epsilon_2^{\kappa_{2s}^p} ... \epsilon_m^{\kappa_{ms}^p}$.

4.2.5 OTI Reduced Order Models

As seen in section 4.2.4, OTI numbers can be used to compute derivatives of functions when evaluated at perturbed input variables following the described methodology. The derivatives are contained within the resulting OTI number multiplied by factors following the Taylor Series Expansion of the function, as seen

for example, in equation (4.57). If one assumes that any OTI number is the result of evaluating a function following perturbations as in equation (4.50), then the TSE of the function truncated to order n can be formed naturally from the coefficients in the OTI number.

Therefore, this expansion can be used to approximate the function at a different evaluation point $\mathbf{x}' = \mathbf{x}_0 + \Delta\mathbf{x}$. That is, the OTI number performs as a reduced order model of the evaluated function by using all derivatives up to order n to generate a polynomial approximation to function f. This forms the OTI reduced order model (OTIROM) of function f. Consider for example the function as in equation 4.57. An approximation of the function evaluated at the new points $x' = x_0 + \Delta x$ and $y' = y_0 + \Delta y$ can be achieved by replacing the imaginary bases ϵ_1 and ϵ_2 by Δx and Δy, respectively:

$$\begin{aligned} f(x', y') = f(x_0 + \Delta x, y_0 + \Delta y) \approx f(x_0, y_0) + \frac{1}{1!} f_{,x}(x_0, y_0)\Delta x + \frac{1}{1!} f_{,y}(x_0, y_0)\Delta y + \\ \frac{1}{2!} f_{,x^2}(x_0, y_0)\Delta x^2 + \frac{1}{1!1!} f_{,xy}(x_0, y_0)\Delta x \Delta y + \frac{1}{2!} f_{,y^2}(x_0, y_0)\Delta y^2 + \frac{1}{3!} f_{,x^3}(x_0, y_0)\Delta x^3 + \\ \frac{1}{2!1!} f_{,x^2 y}(x_0, y_0)\Delta x^2 \Delta y + \frac{1}{1!2!} f_{,xy^2}(x_0, y_0)\Delta x \Delta y^2 + \frac{1}{3!} f_{,y^2}(x_0, y_0)\Delta y^3 \end{aligned} \quad (4.62)$$

which forms the TSE of the function centered at (x_0, y_0) and truncated to order 3, with the coefficients of the OTI number.

In general, the OTIROM is formed by replacing every imaginary basis by the desired change in value of its associated input variable. That is, if perturbation to the input variables occurred as in equation (4.50), then ϵ_l is replaced by Δx_l, for all $l = 1, ..., m$. Finally, the reduced order model is obtained by adding every imaginary coefficient multiplied by its corresponding product of delta values, as follows:

$$\text{OTIROM}\left(a^*, \{\Delta\mathbf{x}\}\right) = \sum_{p=0}^{n} \sum_{s=1}^{N_m^p} \left(a_{\boldsymbol{\alpha}_s^p} \prod_{l=1}^{m} \Delta x_l^{\kappa_{ls}^p} \right), \; a^* \in \mathbb{OTI}_m^n \quad (4.63)$$

where OTIROM$(\cdot)$ is the n'th order reduced order model generated from the OTI number a^* and the input variations $\{\Delta\mathbf{x}\}$.

The approximation error of the OTIROM is a function of the truncation order n of the OTI number, explored in section 4.4.5 and in chapter 5. An example is provided next to further clarify the concept.

Example 7. Reduced order model of an analytic function using OTI numbers.

As an academic example, consider the bivariate function $f : \mathbb{R}^{(2)} \to \mathbb{R}$, defined as follows

$$f(x, y) = \sin(xy) \tag{4.64}$$

The derivatives of the function up to order $n = 2$ are sought evaluated at point $x_0 = 1/2$ and $y_0 = \pi/3$. Derivatives are computed using OTI-perturbed input values as follows

$$x^* = \frac{1}{2} + \epsilon_1, \; x^* \in \mathbb{OTI}_2^2 \tag{4.65}$$
$$y^* = \frac{\pi}{3} + \epsilon_2, \; y^* \in \mathbb{OTI}_2^2 \tag{4.66}$$

The function is evaluated using standard multiplication as defined in section 4.2.2 and the sine function is evaluated following example 6.

$$f(x^*, y^*) = \sin\left(\left(\frac{1}{2} + \epsilon_1\right)\left(\frac{\pi}{3} + \epsilon_2\right)\right) \tag{4.67}$$
$$f(x^*, y^*) = \sin\left(\frac{\pi}{6} + \frac{\pi}{3}\epsilon_1 + \frac{1}{2}\epsilon_2 + \epsilon_1\epsilon_2\right) \tag{4.68}$$
$$f(x^*, y^*) = \frac{1}{2} + \frac{\sqrt{3}}{6}\pi\epsilon_1 + \frac{\sqrt{3}}{4}\epsilon_2 - \frac{1}{36}\pi^2\epsilon_1^2 + \left(\frac{\sqrt{3}}{2} - \frac{1}{12}\pi\right)\epsilon_1\epsilon_2 - \frac{1}{8}\epsilon_2^2 \tag{4.69}$$

A second order OTIROM is generated to approximate the value of the function evaluated at $x' = 1/2 + 0.01$ and $y' = \pi/3 - .01$, with $\Delta x = 0.01$ and $\Delta y = -0.01$. Applying it in equation (4.69), the OTIROM becomes

$$\text{OTIROM}\left(f(x^*, y^*), \{0.01, -0.01\}\right) = \frac{1}{2} + \frac{\sqrt{3}}{6}\pi(\Delta x) + \frac{\sqrt{3}}{4}(\Delta y) - \frac{1}{36}\pi^2(\Delta x)^2 + \left(\frac{\sqrt{3}}{2} - \frac{1}{12}\pi\right)(\Delta x)(\Delta y) - \frac{1}{8}(\Delta y)^2 \tag{4.70}$$
$$\text{OTIROM}\left(f(x^*, y^*), \{0.01, -0.01\}\right) = 0.5046447816328694 \tag{4.71}$$

The exact function evaluation at the new values $f(x', y')$ is 0.5046450303730787. The relative error between the approximation using the OTIROM and the exact result is $\varepsilon = 4.92 \times 10^{-6}$. The method is simple and can expand to more complicated functions. This will be shown later in section 4.4.5 and in chapter 5.

4.2.6 Matrix representation

Similar to multicomplex [66] and multidual [8] numbers, OTI numbers can also be represented and operated in matrix form. One of the uses of a matrix form is that arithmetic operations between OTI numbers can be analogously performed using its corresponding matrix form and matrix algebra. Therefore, operations such as addition, multiplication, evaluation of analytic functions, etc.; can be performed using the matrix form. Another advantage of the matrix form is that it allows to use conventional real-algebra solvers when trying to solve linear systems of OTI equations.

The vector form of an OTI number a^* is represented by the expression $\mathbf{t}(a^*)$. It is formed by the tuple of coefficients of the OTI number. The location of coefficients in the vector is given by associating each imaginary direction to a single element of the vector. The vector form of a^* is not unique. However, in this document it will follow the same arrangement of OTI number directions as described before. For example, given an OTI number $a^* \in \mathbb{OTI}_2^3$ with $m = 2$ bases and order $n = 3$,

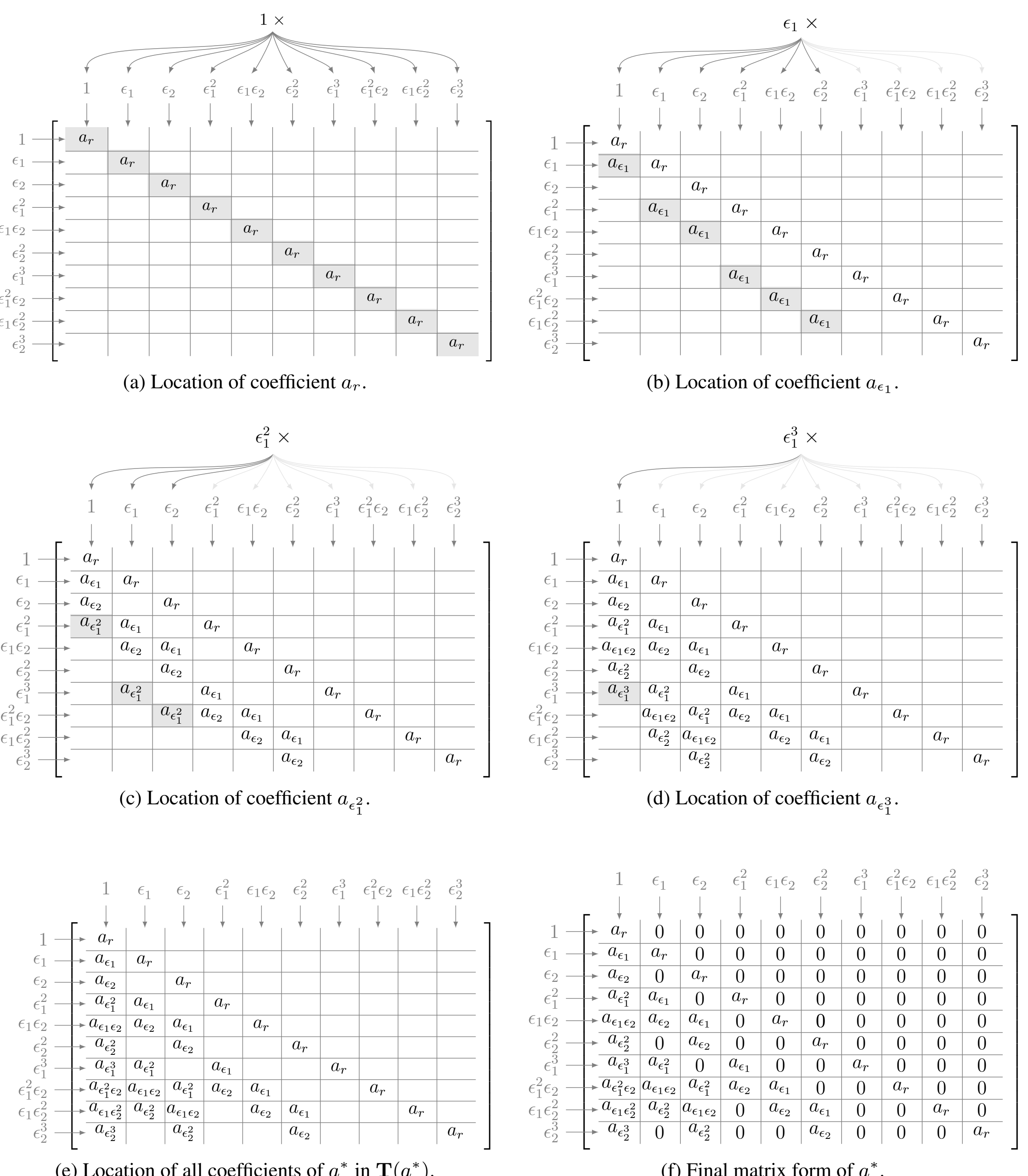
$$a^* = a_r + a_{\epsilon_1}\epsilon_1 + a_{\epsilon_2}\epsilon_2 + a_{\epsilon_1^2}\epsilon_1^2 + a_{\epsilon_1\epsilon_2}\epsilon_1\epsilon_2 + a_{\epsilon_2^2}\epsilon_2^2 + a_{\epsilon_1^3}\epsilon_1^3 + a_{\epsilon_1^2\epsilon_2}\epsilon_1^2\epsilon_2 + a_{\epsilon_1\epsilon_2^2}\epsilon_1\epsilon_2^2 + a_{\epsilon_2^3}\epsilon_2^3 \tag{4.72}$$

its corresponding vector form is:

$$\mathbf{t}(a^*) = \begin{bmatrix} a_r & a_{\epsilon_1} & a_{\epsilon_2} & a_{\epsilon_1^2} & a_{\epsilon_1\epsilon_2} & a_{\epsilon_2^2} & a_{\epsilon_1^3} & a_{\epsilon_1^2\epsilon_2} & a_{\epsilon_1\epsilon_2^2} & a_{\epsilon_2^3} \end{bmatrix}^T \tag{4.73}$$

The matrix form of an OTI number a^* is represented by $\mathbf{T}(a^*)$. The shape of the matrix form is $N \times N$, given by the number of coefficients of the OTI number. In order to define the position of the coefficients of a^* within the matrix, a simple procedure has been developed. First create an empty $N \times N$ matrix and associate each row and column to an imaginary direction in the same sequence as in the vector form (see Figure 4.1a). It follows to determine the position of the coefficients of a^* within $\mathbf{T}(a^*)$. The position of each coefficient is determined by multiplying the coefficient's imaginary direction times the imaginary direction associated to each column. For a specific column, the resulting imaginary direction, if not truncated, gives the row at which the coefficient is located.

As an example, the procedure is applied to a^* from equation 4.72. In order to locate the real coefficient a_r inside the matrix, its direction (real direction $\boldsymbol{\alpha}_1^0 = 1$) is multiplied by the imaginary directions associated



(a) Location of coefficient a_r.

(b) Location of coefficient a_{ϵ_1}.

(c) Location of coefficient $a_{\epsilon_1^2}$.

(d) Location of coefficient $a_{\epsilon_1^3}$.

(e) Location of all coefficients of a^* in $\mathbf{T}(a^*)$.

(f) Final matrix form of a^*.

Figure 4.1: General distribution of coefficients in OTI matrix form.

to each column, as illustrated on top of Figure 4.1a. Since multiplying 1 times any other imaginary direction results in the same imaginary direction, then the real coefficient is always located at the diagonal of the matrix.

Now consider coefficient a_{ϵ_1}, whose imaginary direction is ϵ_1 (see Figure 4.1b). For the first column, multiplication is $\epsilon_1 \cdot 1 = \epsilon_1$ as illustrated on top of Figure 4.1b, which means that a_{ϵ_1} is placed at row 2 column 1, or (2,1). For column 2, multiplication $\epsilon_1 \cdot \epsilon_1 = \epsilon_1^2$ which is located at row 4, hence a_{ϵ_1} is placed at (4,2). The same follows for column 3, multiplication $\epsilon_1 \cdot \epsilon_2 = \epsilon_1\epsilon_2$ results in location (5,3). For column 4, the multiplication results in $\epsilon_1 \cdot \epsilon_1^2 = \epsilon_1^3$, thus location (7,4), and the same is applied to the following columns. Notice that from column 6, the associated directions have order 3. Thus multiplying them by ϵ_1 will result in directions with order 4, which are truncated because the truncation order of a^* is $n = 3$. As a consequence, it is known in advance that a_{ϵ_1} and other coefficients of order 1 and above will not be located at matrix positions beyond column 6. See the full locations of a_{ϵ_1} in Figure 4.1b. The same is applied to a_{ϵ_2}, locating the coefficient at (3,1), (5,2), (6,3), (8,4), (9,5) and (10,6).

The same procedure is applied to $a_{\epsilon_1^2}$ with imaginary direction ϵ_1^2 of order 2. Multiplying with the first column results in $\epsilon_1^2 \cdot 1 = \epsilon_1^2$, location (4,1). Multiplying with column 2 and column 3 results in $\epsilon_1^2 \cdot \epsilon_1 = \epsilon_1^3$ (7,2) and $\epsilon_1^2 \cdot \epsilon_2 = \epsilon_1^2\epsilon_2$ (8,3) respectively (see Figure 4.1c). Notice that every imaginary direction from column 4 onwards has order 2 or greater, thus after multiplying by ϵ_1^2 the order will be greater than 4, therefore truncating the imaginary directions. Similarly, applying the procedure to $a_{\epsilon_1\epsilon_2}$ results in locations (5,1), (8,2) and (9,3). Coefficient $a_{\epsilon_2^2}$ is located at (6,1), (9,2) and (10,3). Applying the same procedure to coefficients with order 3 directions such as $a_{\epsilon_1^3}$, these coefficients will only be located at column 1 since multiplication with any other imaginary direction will truncate the result. Therefore, coefficients $a_{\epsilon_1^3}$, $a_{\epsilon_1^2\epsilon_2}$, $a_{\epsilon_1\epsilon_2^2}$, and $a_{\epsilon_2^3}$ are located at (7,1), (8,1), (9,1) and (10,1) respectively, as shown in Figure 4.1d.

The careful reader may have noticed that because of sorting the OTI number by order, the coefficients in the matrix form are placed in blocks of coefficients with the same order. This can be seen more clearly in Figure 4.2. For example, notice the block made from row 4 to row 6 and column 2 to column 3. All elements within this block are either zero or coefficients of order 1 imaginary directions. Also, the block below (rows 7-10 and cols 2-3) contains coefficients of order 2. This arrangement is most useful for solving linear systems of OTI equations, as will be discussed later in section 5.2.

In general, an OTI matrix representation has the following characteristics:

- The elements are all real and its shape is $N \times N$, i.e. $\mathbf{T}(a^*) \in \mathbb{R}^{(N \times N)}$.

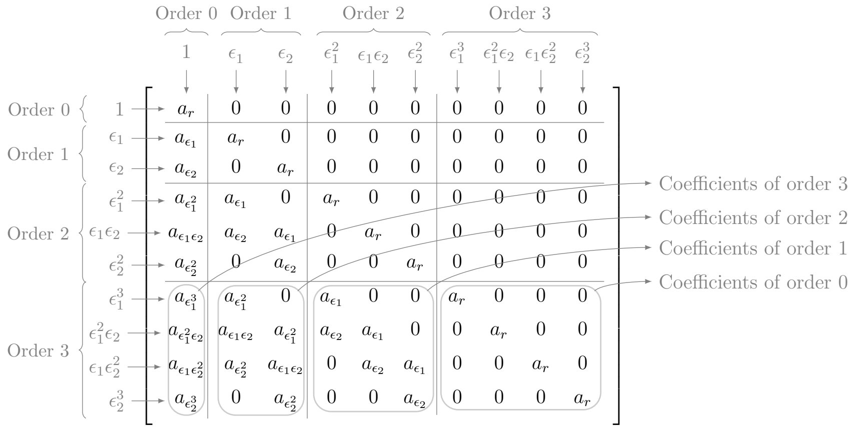


Figure 4.2: Blocks determined by order separation of the OTI matrix form for an OTI number $\mathbb{OTI}_2^3$.

- All diagonal elements are equal to the real coefficient of the OTI number.
- The first column of the matrix is formed by the vector form of the OTI number.
- An imaginary direction coefficient is not repeated in a column or in a row.
- The matrix is formed by rectangular blocks containing coefficients sharing the same order, as shown in Figure 4.2.
- Every element above the diagonal is zero. Hence, the matrix is always lower triangular.
- In general, a large portion of the lower triangle is zero. The total number of non-zero coefficients $N_\mathbf{T}$ in the matrix representation depends on the number of imaginary bases m and truncation order n of the OTI number as follows:

$$N_\mathbf{T} = \binom{2m+n}{2m} = \frac{(2m+n)!}{(2m)!\, n!} \tag{4.74}$$

Matrix and vector forms have equivalent arithmetic operations to OTI algebra. Consider OTI numbers a^*, b^* and c^*. Its corresponding vector forms $\mathbf{t}(a^*)$, $\mathbf{t}(b^*)$ and $\mathbf{t}(c^*)$; and matrix forms $\mathbf{T}(a^*)$, $\mathbf{T}(b^*)$ and $\mathbf{T}(c^*)$; have the equivalent operations as described in Table 4.2.

	Addition/Subtraction	**Multiplication**	**Functions**
OTI	$a^* \pm b^* = c^*$	$a^* \cdot b^* = c^*$	$f(a^*) = c^*$
Matrix Equivalence	$\mathbf{T}(a^*) \pm \mathbf{T}(b^*) = \mathbf{T}(c^*)$	$\mathbf{T}(a^*)\mathbf{T}(b^*) = \mathbf{T}(c^*)$	$F(\mathbf{T}(a^*)) = \mathbf{T}(c^*)$
Vector Equivalence	$\mathbf{t}(a^*) \pm \mathbf{t}(b^*) = \mathbf{t}(c^*)$	$\mathbf{T}(a^*)\mathbf{t}(b^*) = \mathbf{t}(c^*)$	N/A

Table 4.2: Corresponding operations of OTI matrix and vector forms.

The advantage of matrix and vector forms is that there is no need of specific libraries to implement the OTI algebra, only linear algebra support. However, the computational cost (both in memory and CPU time) increases either by the square of the number of coefficients N (matrices instead of vectors) or by use of sparse matrices. The matrix form is, however, an important step to deduce an optimal procedure to solve hypercomplex linear systems of equations as it will be shown in the section 5.2.

4.2.7 Relationship with other hypercomplex algebras.

The $\mathbb{OTI}$ set contains other hypercomplex algebras. The first, and most obvious case, is that OTI numbers contain $\mathbb{R}$, the real number set, because if truncation order n is set to zero all imaginary units are truncated, then the only remaining coefficient is the real one. The same applies if the number of bases is $m = 0$. As a consequence $\mathbb{OTI}_0^0 = \mathbb{R}$.

It can be shown that OTI numbers contain Multidual numbers. For example, consider a bidual number $x^* \in \mathbb{D}_2$ and an OTI number $y^* \in \mathbb{OTI}_2^2$ as follows

$$x^* = x_r + x_{\epsilon_1}\epsilon_1 + x_{\epsilon_2}\epsilon_2 + x_{\epsilon_1\epsilon_2}\epsilon_1\epsilon_2 \quad , \; x^* \in \mathbb{D}_2 \tag{4.75}$$

$$y^* = y_r + y_{\epsilon_1}\epsilon_1 + y_{\epsilon_2}\epsilon_2 + y_{\epsilon_1^2}\epsilon_1^2 + y_{\epsilon_1\epsilon_2}\epsilon_1\epsilon_2 + y_{\epsilon_2^2}\epsilon_2^2, \; y^* \in \mathbb{OTI}_2^2 \tag{4.76}$$

Note that every imaginary direction of x^* is contained in the imaginary directions of y^*. The particular, imaginary directions ϵ_1^2 and ϵ_2^2 are eliminated from the bidual number by the nilpotent condition, namely $\epsilon_1^2 = \epsilon_2^2 = 0$.

In general, the multidual set $\mathbb{D}_m$ with m bases is contained in the OTI set of m bases and truncation order $n = m$, i.e. $\mathbb{D}_m \subset \mathbb{OTI}_m^m$.

4.3 Support Library

A library, named OTIlib, has been developed in C programming language with the goal to provide support for core OTI algebra algorithms with the better performance associated with a compiled language. C was selected since it allows the programmer to control most low level aspects of the implementation. Also, a wrapper to Python 3 has been developed using Cython in order to add operator overload support and hence make it easier to interface with the algebra and algorithms. Python language was selected as it is widely

adopted in the scientific community. The current state of the implementation does not implement parallelism for any of the core algorithms.

The library has two main data representations: static dense and sparse numbers. The coefficient type "`coeff_t`" used for all representations is the 8-byte, double precision real type provided by the C-language. The static dense data representation of OTI numbers provides better performance in terms of computational time for a given algebra $\mathbb{OTI}_m^n$, while restricting the implementation to specific truncation order n and number of bases m, and lacking versatility to increase or decrease the number of imaginary directions or truncation order, as its memory implementation is static. On the other hand, the sparse data representation of the number allows dynamic memory allocation, permitting both variable number of imaginary directions and variable truncation order with the added benefit of only storing the non-zero coefficients in order to reduce the overall number of operations and memory usage. Both approaches have computational advantages, justifying its coexistence in the library. A brief description is provided in the following sections.

4.3.1 Static dense implementation

The static dense representation is implemented to provide a fast implementation for OTI operations in the sense of three factors: (i) a static memory implementation such that the memory for the OTI number is preserved constant through the execution of the program, avoiding explicit memory allocation calls which may negatively impact the performance of the implementation; (ii) a full representation of the OTI number such that all coefficients are always available and easily accessed and (iii) arithmetic operations are explicitly defined in the code such that no computational resources are spent to "find" the way the imaginary coefficients must be operated. It is important to note that the code to support static dense OTI numbers, as shown here, is generated automatically by using a helper function for that matter.

Figure 4.3 shows two examples of the data structures used to define static dense OTI numbers. Figure 4.3 left, shows the representation of an OTI number for a total of $m = 2$ bases and truncation order $n = 3$. Figure 4.3 right shows the structure of an OTI number for a total of $m = 3$ bases and truncation order $n = 2$ (note that for both cases, the total number of coefficients is $N = 10$).

Static dense $\mathbb{OTI}_2^3$ scalar	Static dense $\mathbb{OTI}_3^2$ scalar

```
typedef struct {
  coeff_t     r;
  coeff_t    e1;
  coeff_t    e2;
  coeff_t   e11;
  coeff_t   e12;
  coeff_t   e22;
  coeff_t  e111;
  coeff_t  e112;
  coeff_t  e122;
  coeff_t  e222;
} onumm2n3_t;
```

```
typedef struct {
  coeff_t     r;
  coeff_t    e1;
  coeff_t    e2;
  coeff_t    e3;
  coeff_t   e11;
  coeff_t   e12;
  coeff_t   e22;
  coeff_t   e13;
  coeff_t   e23;
  coeff_t   e33;
} onumm3n2_t;
```

Figure 4.3: Static dense data types for an OTI number with truncation order $n = 3$ and $m = 2$ bases $\mathbb{OTI}_2^3$ (left) and truncation order $n = 2$ and $m = 3$ bases $\mathbb{OTI}_3^2$ (right).

Static dense implementations of multidual numbers is also supported and automatically generated.

4.3.1.1 Arithmetic operations

Arithmetic operations are provided via C-functions. An example of the multiplication implementation between two $\mathbb{OTI}_3^2$ numbers is shown in Figure 4.4. Note that the advantage of this implementation is that manipulation of coefficients is explicitly programmed in the code, that is, it does not require additional computational resources to find how the imaginary coefficients are operated.

Static dense multiplication

```
onumm3n2_t onumm3n2_mul_oo(  onumm3n2_t* lhs,  onumm3n2_t* rhs){
  onumm3n2_t res;

  //  Multiplication like function 'lhs * rhs'
  // Real;
  res.r = lhs->r * rhs->r;
  // Order 1;
  res.e1 = lhs->r * rhs->e1 + lhs->e1 * rhs->r;
  res.e2 = lhs->r * rhs->e2 + lhs->e2 * rhs->r;
  res.e3 = lhs->r * rhs->e3 + lhs->e3 * rhs->r;
  // Order 2;
  res.e11 = lhs->r * rhs->e11 + lhs->e11 * rhs->r + lhs->e1 * rhs->e1;
  res.e12 = lhs->r * rhs->e12 + lhs->e12 * rhs->r + lhs->e1 * rhs->e2
          + lhs->e2 * rhs->e1;
  res.e22 = lhs->r * rhs->e22 + lhs->e22 * rhs->r + lhs->e2 * rhs->e2;
  res.e13 = lhs->r * rhs->e13 + lhs->e13 * rhs->r + lhs->e1 * rhs->e3
          + lhs->e3 * rhs->e1;
  res.e23 = lhs->r * rhs->e23 + lhs->e23 * rhs->r + lhs->e2 * rhs->e3
          + lhs->e3 * rhs->e2;
  res.e33 = lhs->r * rhs->e33 + lhs->e33 * rhs->r + lhs->e3 * rhs->e3;

  return res;

}
```

Figure 4.4: Sample of a static dense multiplication between two static dense OTI numbers with $m = 3$ bases and truncation order $n = 2$, $\mathbb{OTI}_3^2$.

An important characteristic of the implementation is that all OTI input data is passed by reference to avoid excessive memory copying. Unary arithmetic operations such as negation is supported, and binary arithmetic operations: addition, subtraction and multiplication, are implemented analogously to the operation shown in Figure 4.4 for OTI-OTI, real-OTI and OTI-real (the latter only for subtraction, as addition and multiplication are commutative operators). Elementary functions are implemented in C following the method described in section 4.2.3. Finally, division is implemented as $x^*/y^* = x^*y^{*-1}$, where y^{*-1} is computed using the single variable power function, similar to the one presented in example 5.

4.3.2 Sparse implementation

A sparse implementation is provided in OTIlib because it is common to find applications that do not utilize all available imaginary coefficients of the algebra, i.e. must coefficients may be zero. As such, by knowing in advance that some coefficients are zero, operations with these zero coefficients can be avoided directly reducing computational costs. For this matter, the data structure defined for the sparse OTI representation is shown in Figure 4.5.

Sparse OTI

```
typedef struct {
  coeff_t        re; //<-- Real coefficient.
  coeff_t**    p_im; //<-- Array with all imaginary coeffs.
  imdir_t**   p_idx; //<-- Array of indices of directions.
  ndir_t*     p_nnz; //<-- Number of non zero coefficients per order.
  ndir_t*    p_size; //<-- Size per array.
  ord_t       order; //<-- Truncation order of the number.
} sotinum_t;         // Sparse OTI number type.
```

Figure 4.5: Data structure used for the sparse OTI implementation in the C portion of OTIlib.

The truncation order of the number is stored in `order`, where `ord_t` is a 1-byte unsigned integer. The real coefficient of the number is stored in `re`. It is separated from the other imaginary coefficients to allow faster access in the case the object represent a real-only number, i.e. $\mathbb{OTI}_0^0 = \mathbb{R}$.

The attribute `p_im` represents an n-element list of arrays with the imaginary coefficients of the number. The first component of `p_im`, `p_im[0]`, is an array with all non-zero coefficients of order 1 in the OTI number. The second element of the list, `p_im[1]`, is an array with all the non-zero coefficients of order 2. In general the p'th element of the list contains all non-zero coefficients of order p. Similarly, `p_idx` is a n-element list of arrays containing the indices of the imaginary directions for every coefficient in `p_im`. The type `imdir_t` used in the current version of the library is 8-byte unsigned integer. There are two more lists, `p_nnz` and `p_size`. The p'th element in each list indicate the number of active non-zero coefficients and the size of the p'th array, respectively, for both `p_im` and `p_idx`.

For more clarity, a visual depiction of the sparse data structures is sketched in Figure 4.6. Note that, for example, imaginary coefficient $a_{\boldsymbol{\alpha}_{s_1}^1}$ is located in the first element of the first array of `p_im`. Its index s_1 is located in the first element of the first array of `p_idx`. Notice that the p'th number of active non-zero elements may be lower or equal to the size of the p'th array, namely $\mathrm{nz}^p \leq \mathrm{sz}^p$, because the structure may reserve more memory if the current object is expected to increase the number of non-zero coefficients later in the program.

An example for both static dense and sparse implementations is presented considering OTI number $a^* = 3 + \epsilon_2 + 4\epsilon_3 + 2\epsilon_1\epsilon_2$, $\mathbb{OTI}_3^2$, of order 2 and three bases. The explicit OTI number representation is

$$a^* = 3 + 0\underbrace{\epsilon_1}_{\boldsymbol{\alpha}_1^1} + 1\underbrace{\epsilon_2}_{\boldsymbol{\alpha}_2^1} + 4\underbrace{\epsilon_3}_{\boldsymbol{\alpha}_3^1} + 0\underbrace{\epsilon_1^2}_{\boldsymbol{\alpha}_1^2} + 2\underbrace{\epsilon_1\epsilon_2}_{\boldsymbol{\alpha}_2^2} + 0\underbrace{\epsilon_2^2}_{\boldsymbol{\alpha}_3^2} + 0\underbrace{\epsilon_1\epsilon_3}_{\boldsymbol{\alpha}_4^2} + 0\underbrace{\epsilon_2\epsilon_3}_{\boldsymbol{\alpha}_5^2} + 0\underbrace{\epsilon_3^2}_{\boldsymbol{\alpha}_6^2}$$

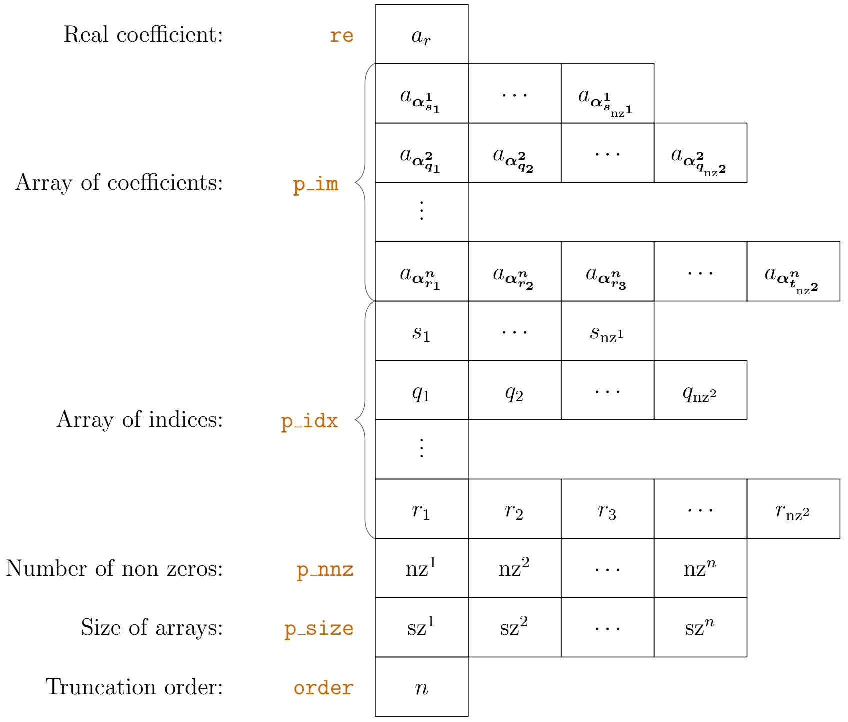


Figure 4.6: Visual representation of the sparse data structures defined in OTIlib.

Figure 4.7 shows how static dense and sparse implementations will set the data in the respective memory structure. Note that the sparse implementation only stores 4 coefficients, and only use those when it is operated by another number. In contrast, the dense implementation stores all 10 coefficients even though 6 of them are equal to zero.

4.3.2.1 Arithmetic operations

Addition and subtraction are trivial implementations because imaginary directions with the same order and index can be added/subtracted together. Every binary operation implemented in OTIlib for sparse numbers preserves the maximum truncation order from the two given operands.

Multiplication, on the other hand, requires a brief explanation. The result of multiplying directions $\boldsymbol{\alpha}_t^o = \boldsymbol{\alpha}_r^q \cdot \boldsymbol{\alpha}_s^p = \boldsymbol{\alpha}_s^p \cdot \boldsymbol{\alpha}_r^q$ is stored in a precomputed multiplication table $\text{MT}_m^{o,q}$, where $q \leq p$ and m is the number of bases supported for order o. Tables are generated for every order o supported in the library, for all orders $q = 1, ..., \lfloor o/2 \rfloor$[2] such that $o = q + p$. The size of every multiplication table is given by $(N_m^q \times N_m^p)$. In general, the element (r, s) in the table contains the index t of the resulting imaginary direction of order

[2]The symbol $\lfloor \cdot \rfloor$ represents the floor operator.

Static dense a^*:

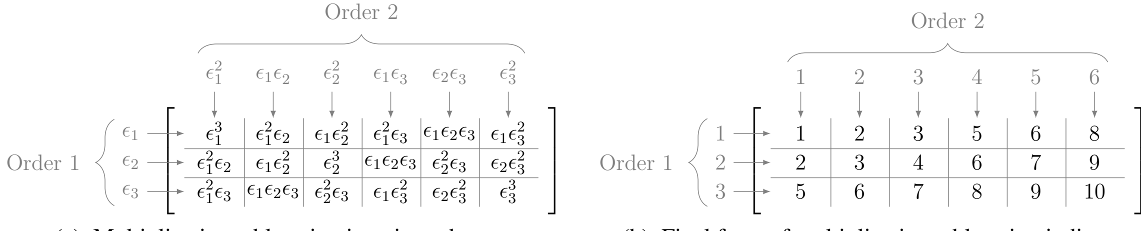
r	3.0
e1	0.0
e2	1.0
e3	4.0
e11	0.0
e12	2.0
e22	0.0
e13	0.0
e23	0.0
e33	0.0

Sparse a^*:

re	3.0	
p_im	1.0	4.0
	2.0	
p_idx	2	3
	2	
p_nnz	2	1
p_size	2	1
order	2	

Figure 4.7: Comparison between sparse and dense implementations

o, thus $\mathrm{MT}_m^{o;q}(r,s) = t$. As an example, consider multiplication of imaginary directions of order 1 times imaginary directions of order 2 for a total of $m = 2$ bases. All imaginary directions of order 1 are associated to the rows of the table, and similarly order 2 directions are associated to the columns as shown in Figure 4.8a. The resulting imaginary directions of order 3 are contained in the elements of the table. The indices of the resulting imaginary directions are extracted as in Table 4.1, and the final form of the multiplication table $\mathrm{MT}_3^{3,1}$ is shown in Figure 4.8b.



(a) Multiplication table using imaginary bases. (b) Final form of multiplication table using indices.

Figure 4.8: Example of multiplication table for imaginary directions with order 1 times directions with order 2 and 3 bases, using (a) a representation using imaginary basis and (b) the final form of the multiplication table using indices.

Multiplication tables are generated automatically using helper functions provided with OTIlib. The standard distribution of the library automatically generates data to support the order and number of bases shown in Table 4.3. According to Table 4.3, the library supports operations with OTIs of order 1 and up to 65000 bases, however for directions of order 2 it only supports operations with up to 1000 bases. Directions

with order 3 are supported with up to 100 imaginary bases. Directions of order from 4 to 10 support up to 10 imaginary bases. Directions with order 10 to 20 support up to 5 bases. Directions with order 20 to 50 support up to 3 imaginary bases. Finally, directions with order 50 up to 150 support up to 2 bases. In other words, the standard distribution of OTIlib can be used to compute first order derivatives of up 65000 variables as well as derivatives of order 150 for up to 2 variables. These limitations arise from the available precomputed data, so if greater number of variables or order is required, the provided helper function can generate the required data. Precomputed multiplication tables for the standard capabilities of OTIlib as shown in Table 4.3 occupy a total of 497.1 MB, for a total of 5626 multiplication tables.

OTI Order	Maximum number of bases
1	65000
2	1000
3	100
4 to 10	10
10 to 20	5
20 to 50	3
50 to 150	2

Table 4.3: Capabilities of the standard distribution of OTIlib.

Other precomputed data are the arrays forming the full imaginary directions for each order. These arrays are 2 dimensional, in which the s'th row of the p'th array correspond to the expanded list of basis indices for imaginary direction $\boldsymbol{\alpha}_s^p$. For example, imaginary direction $\epsilon_1^2\epsilon_3$ has an expanded list of basis indices $[1, 1, 3]$, since $\epsilon_1^2\epsilon_3 = \epsilon_1\epsilon_1\epsilon_3$. This list is the 5th component of the full imaginary direction array of order 3, as the index of $\epsilon_1^2\epsilon_3$ is 5 (see Table 4.1). The full imaginary direction arrays occupy a total 12.2 MB for the standard distribution of OTIlib with a total of 150 arrays. The total memory cost of precomputed data for the standard capabilities is 509.3 MB.

4.3.3 Python OTI Library (pyOTI)

A Python 3 wrapper was developed to provide a user friendly interface to OTI number algebra. Cython 0.29 [67] was used to easily integrate C programmed functions with Python specific functions. Operator overloading allows easier-to-read code than if implemented using function calls.

The library implements overloads to basic algebraic operations such as addition, subtraction, negation and multiplication. . Table 4.4 shows currently supported operators in pyOTI.

The library implements the method described in section 4.2.3 to support elementary function evaluation.

Operation	pyOTI operator
Addition	`a + b`
Subtraction	`a - b`
Negation	`- a`
Multiplication	`a * b`
Division	`a / b`
Power	`a ** b`

Table 4.4: Overloaded operators supported in the current version of the pyOTI.

Power function and division are implemented as elementary functions. The library, in its current form, supports the functions listed in Table 4.5.

pyOTI call	Mathematical symbol
`cos(x)`	$\cos(x)$
`sin(x)`	$\sin(x)$
`tan(x)`	$\tan(x)$
`sinh(x)`	$\sinh(x)$
`cosh(x)`	$\cosh(x)$
`tanh(x)`	$\tanh(x)$
`power(x,n)`	x^n
`log(x)`	$\ln(x)$
`log10(x)`	$\log_{10}(x)$
`atan2(x,y)`	$\tan^{-1}\left(\frac{x}{y}\right)$

pyOTI call	Mathematical symbol
`acos(x)`	$\cos^{-1}(x)$
`asin(x)`	$\sin^{-1}(x)$
`atan(x)`	$\tan^{-1}(x)$
`asinh(x)`	$\sinh^{-1}(x)$
`acosh(x)`	$\cosh^{-1}(x)$
`atanh(x)`	$\tanh^{-1}(x)$
`sqrt(x)`	$\sqrt{x}$
`exp(x)`	e^x
`logb(x,b)`	$\log_b(x)$

Table 4.5: pyOTI support for elementary functions (for both sparse and static dense versions).

To implement an algorithm that uses the OTI library, the crucial step is to hypercomplexify the input variables. This is done in the library by adding an imaginary direction of choice to the variable whose sensitivity is required. Operator overloads carry on the basic algebraic computations, however it is necessary to point to the pyOTI's elementary function calls in order to support evaluation of functions correctly.

In order to do so, the next steps must be followed:

i) Import the library,

ii) perturb the input variables using OTI imaginary directions

iii) evaluate your algorithm with minimum to none syntax modifications, and

iv) retrieve/use derivatives with the built-in methods.

As an illustrative example, Figure 4.9, shows both an example of how to evaluate function $f(x, y, z) = \cos\left(x^3 y^2 z^4\right)$ using the pyOTI and how the code compares with an equivalent real-valued code in Python 3 that does not get derivatives.

Standard real code	OTI code

```python
# Import math function cosine
from math import *


# Define the variables
x = 0.2
y = 0.5
z = 1.4

# Evaluate the desired function.
f_eval = cos( x**3 * y**2 * z**4 )
```

```python
# Make OTI library available
from pyoti.sparse import *

# Define the variables by perturbing them
# in OTI directions.
x = 0.2 + e( 1, order = 4)
y = 0.5 + e( 2, order = 4)
z = 1.4 + e( 3, order = 4)

# Evaluate the desired function.
f_eval = cos( x**3 * y**2 * z**4 )

# Get derivatives.
df_dx      = f_eval.get_deriv([1])
d2f_dxd    = f_eval.get_deriv([1,2])
#           ...
d4f_dxdydz2 = f_eval.get_deriv([1,2,[3,2]])
```

Figure 4.9: Sample codes for evaluation of function $f(x, y, z) = \cos\left(x^3 y^2 z^4\right)$ using real algebra (left) and OTI numbers for computation of fourth order derivatives with pyOTI (right).

The OTI direction generator function, i.e. `e( 3, order = 4)`, returns a sparse OTI number with a coefficient of 1 in the given direction and truncation order $n = 4$. If `order` is not specified in the call, the order of the imaginary direction given is used as truncation order. The function takes as input the "human readable" version of the OTI imaginary direction. In this case, 3 represents the imaginary basis ϵ_3. A general imaginary direction, e.g. $\epsilon_4\epsilon_5\epsilon_7$ is represented by `[ 4, 5, 7]`. In the case of more complex imaginary directions, e.g. $\epsilon_4^2\epsilon_5^3\epsilon_7^4$, the direction can be given in the following manners: `[ 4, 4, 5, 5, 5, 7, 7, 7, 7]` by explicitly listing all imaginary basis or more compactly `[ [4,2], [5,3], [7,4] ]`, associating each basis with its corresponding exponent.

The variable `f_eval` for the example case is an OTI sparse scalar containing the derivatives of the evaluated function. The method `f_eval.get_deriv( imdir )` extracts the derivative associated to the imaginary direction `imdir`, which must follow the format described previously.

The method `f_eval.rom_eval( bases_list, deltas )` evaluates the OTIROM given in the OTI number by assigning each given basis the specified variation. The input `bases_list` is a list of integers and `deltas` is a list of real numbers such that `deltas[i]` is the variation of the input variable

associated to basis `bases_list[i]`.

4.4 Results

This section describes some numerical and analytical examples with OTI numbers. The numerical examples were run in a machine with an Intel® Core™ i7 4770 at 3.6 GHz, 32 GB ram at 1600 MHz and the operating system was Ubuntu 18.04. The compiler used for C and Cython [67] was the GNU C Compiler GCC 7.5.0 (the version of Cython used was 0.29.20). Python 3.7.4 [68] from the Anaconda distribution 2019.10 [69] was used. Scipy 1.5.2 [70], Sympy 1.6.2 [71] and Numpy1.19.1 [72] were used.

4.4.1 OTI numbers vs Multicomplex and Multidual Algebras

The complexity and memory consumption of an algebra can be quantified by its size, i.e, by the total number of coefficients each the algebra has. As explained in Appendix A.2, the total number of coefficients in multicomplex and multidual algebras is given by $N = 2^{mn}$ when used to compute n order derivatives with respect to m variables in a single analysis. These number of coefficients correspond to the vector form of the numbers. In contrast, as explained in Section 4.2, the total number of coefficients of OTI numbers is given by $N = \binom{m+n}{m}$. Instead, the matrix forms of the numbers will contain: 2^{2mn} non zero coefficients in the multicomplex case; 3^{mn} non zero coefficients in the multidual case and in the OTI case $\binom{2m+n}{2m}$. Figure 4.10 shows how the number of coefficients escalate in both cases for different values of order of derivative n and number of variables m.

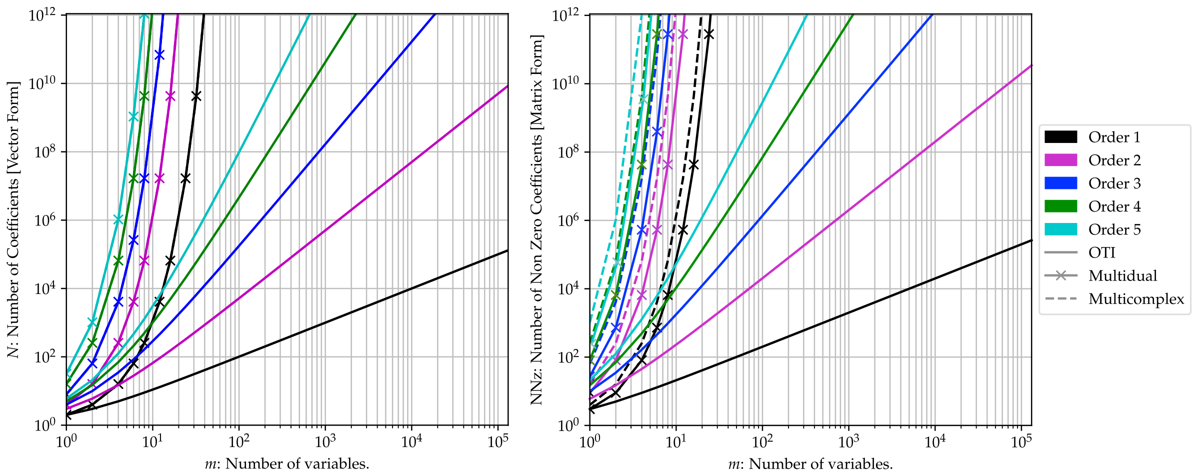


Figure 4.10: Comparison of the number of coefficients of multicomplex, multidual and OTI numbers forms. The number of coefficients of the corresponding vector forms (left) and number of non-zero coefficients of the corresponding matrix forms (right) are displayed.

For computation of 10 first order derivatives, multicomplex and multduals need 1024 coefficients and OTI numbers require 11 coefficients, a difference of two orders of magnitude. For the case of 10 second order derivatives, multiduals and multicomplex are 4 orders of magnitude larger than OTI numebers. This clearly shows the advantages of using the OTI algebra over other hypercomplex algebras if *one* function evaluation is sought. The memory consumption of multicumplex and multidual algebras and the execution time multicomplex/multiduals grow exponentially compared to equivalent OTI evaluation. The repetitive scheme can be used to reduce the computational complexity when multivariable derivatives are required, see Appendix A.2.2.2. This scheme requires, for multidual and multicomplex algebras, a total number of evaluations of the function given by $\binom{m+n-1}{m-1}$.

Figure 4.11 shows how the number of repetitions grow with respect to the number of variables and order of derivatives. OTI number only require a single evaluation. Figure 4.11 is not a reference of the complexity of time usage. This will be addressed later in a computational comparison.

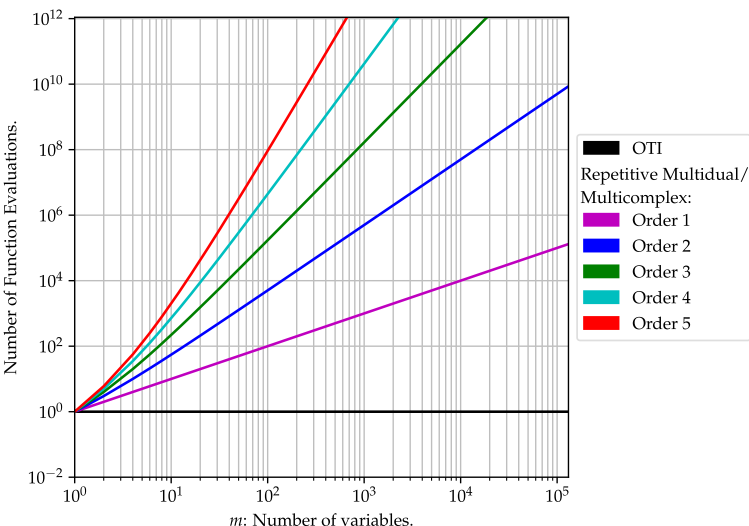


Figure 4.11: Number of function evaluations required by multicomplex and multidual algebras for computing multivariable derivatives. OTI numbers in contrast only requires 1 evaluation.

4.4.1.1 Analytic comparison

To appreciate the difference between OTI and multidual numbers, an example is shown for the single variable function $f : \mathbb{R} \to \mathbb{R}$,

$$f(x) = x^2 \tag{4.77}$$

and derivatives up to second order are computed at $x = 2$. To do so, an OTI of order 2 with only one basis is required. A bidual is required in the multidual case. The OTI perturbation required is

$$x^* \quad = \quad 2 + \epsilon_1 \tag{4.78}$$

while the bidual perturbation is

$$x^* \quad = \quad 2 + \epsilon_1 + \epsilon_2 \tag{4.79}$$

Table 4.6 shows the required steps both OTI and bidual algebra perform in order to obtain the results. It can be observed that in general, the OTI algebra performs less steps, and hence it is easier to execute. Also, it shows that bidual algebra obtains two repeated results ($f_{,x}$) in its imaginary directions ϵ_1 and ϵ_2, and it also shows that many intermediate operations are repeated, making the overall computation more complex.

4.4.2 Computational Comparison

In order to compare the performance of OTI numbers against multidual numbers, an implementation using matrix forms has been developed in Python 3 and Scipy [70]. The goal of using matrix forms is to provide an implementation independent comparison, i.e. to avoid performance differences due to implementation specific optimizations. Particularly, sparse matrices are used to define the equivalent hypercomplex number matrix form, using Compressed Sparse Row (CSR) matrix format. This provides fast matrix-vector multiplication.

The function to compare the performance is the following 60 variable function $f : \mathbb{R}^{60} \to \mathbb{R}$,

$$f(x_1, x_2, ..., x_{60}) = \prod_{l=1}^{60} \sin(x_l) \tag{4.80}$$

Implementation was performed in Python using Scipy [70] library and its built in sparse matrix sine function. Multiplication between two elements is done using matrix-vector multiplication. All derivatives were computed using Multidual, repetitive Multidual and OTI algebras.

Figure 4.12 shows the execution time for the computation of up to 70 variables and OTI matrix form

OTI algebra	Bidual algebra
$\begin{aligned} f(x^*) &= (x^*)^2 \\ &= (2+\epsilon_1)^2 \\ &= (2+\epsilon_1)(2+\epsilon_1) \\ &= 2(2+\epsilon_1)+\epsilon_1(2+\epsilon_1) \\ &= 4+2\epsilon_1+2\epsilon_1+\epsilon_1^2 \\ &= 4+4\epsilon_1+\epsilon_1^2 \end{aligned}$	$\begin{aligned} f(x^*) &= (x^*)^2 \\ &= (2+\epsilon_1+\epsilon_2)^2 \\ &= (2+\epsilon_1+\epsilon_2)(2+\epsilon_1+\epsilon_2) \\ &= 2(2+\epsilon_1+\epsilon_2)+\epsilon_1(2+\epsilon_1+\epsilon_2)+\epsilon_2(2+\epsilon_1+\epsilon_2) \\ &= 4+2\epsilon_1+2\epsilon_2+2\epsilon_1+\cancel{\epsilon_1^2}+\epsilon_1\epsilon_2+2\epsilon_2+\cancel{\epsilon_2^2}+\epsilon_1\epsilon_2 \\ &= 4+4\epsilon_1+4\epsilon_2+2\epsilon_1\epsilon_2 \end{aligned}$
$\therefore$	$\therefore$
$\begin{aligned} f(2) &= \text{Re}\ [f(x^*)] \\ &= 4 \end{aligned}$	$\begin{aligned} f(2) &= \text{Re}\ [f(x^*)] \\ &= 4 \end{aligned}$
$\begin{aligned} \frac{df}{dx}(2) &= \text{Im}_{\epsilon_1}[f(x^*)] \\ &= 4 \end{aligned}$	$\begin{aligned} \frac{df}{dx}(2) &= \text{Im}_{\epsilon_1}\ [f(x^*)] \\ &= 4 \end{aligned}$
	$\begin{aligned} \frac{df}{dx}(2) &= \text{Im}_{\epsilon_2}\ [f(x^*)] \\ &= 4 \end{aligned}$
$\begin{aligned} \frac{d^2f}{dx^2}(2) &= 2!\ \text{Im}_{\epsilon_1^2}[f(x^*)] = (2)1 \\ &= 2 \end{aligned}$	$\begin{aligned} \frac{d^2f}{dx^2}(2) &= \text{Im}_{\epsilon_1\epsilon_2}[f(x^*)] \\ &= 2 \end{aligned}$

Table 4.6: Comparative example to evaluate $f(x) = x^2$ using OTI and bidual numbers in order to compute all up to second order derivatives.

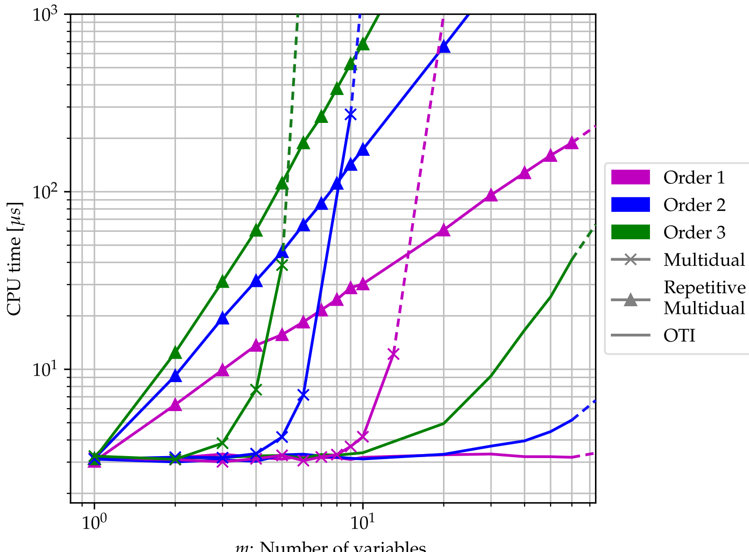


Figure 4.12: Computation time of function OTI comparison with two multidual implementations.

implementation is, for almost every case, below the CPU times of the other methods. While OTI order 1 computation time remained almost constant for all variables, the standard implementation grows exponentially and repetitive multidual grows polynomically. OTI order 2 and order 3 computation time increases after 10 variable analysis. The CPU time of 3μs is due to the overhead of the sparse matrix algebra. The repetitive multidual implementation is driven by this overhead as the number of repetitions require the function to be re-evaluated at a different perturbation scheme.

4.4.3 Numerical Example using pyOTI

As an academic example, consider the three variable function $f : \mathbb{R}^3 \to \mathbb{R}$,

$$f(x, y, z) = \sin\left(\log(x^2 e^{yz^3}) \cos\left(x^3 y^2 z^4\right)\right) \tag{4.81}$$

The goal is to illustrate the use of the method to compute all derivatives up to 7'th order evaluated at $x = 1.5$, $y = 2.0$ and $z = 3.0$ and assess its accuracy. The total number of derivatives, including the function evaluation, is 120. In order to compute the derivatives, the variables are perturbed using OTI imaginary directions following the methodology described in section 4.2.4. Each variable is associated and perturbed with an independent basis as follows and the truncation order is set to $n = 7$ as 7'th order derivatives are sought:

$$x^* = 1.5 + \epsilon_1,\ x^* \in \mathbb{OTI}_3^7 \tag{4.82}$$

$$y^* = 2.0 + \epsilon_2,\ y^* \in \mathbb{OTI}_3^7 \tag{4.83}$$

$$z^* = 3.0 + \epsilon_3,\ z^* \in \mathbb{OTI}_3^7 \tag{4.84}$$

The Python code used to evaluate this example is shown in Figure 4.13. The sparse algebra implementation included in pyOTI is used. Note that compared with a real only implementation, the difference is localized in the perturbation of the input variables in the respective imaginary directions.

```
# Make OTI library available
from pyoti.sparse import e, sin, log, exp, cos

# Define the variables and its OTI perturbations.
x = 1.5 + e( 1, order = 7)
y = 2.0 + e( 2, order = 7)
z = 3.0 + e( 3, order = 7)

# Define and evaluate the  desired function.
f_eval = sin( log( x**2 * exp( y * z**3 ) ) * cos( x**3 * y**2 * z**4 ) )
```

Figure 4.13: Implementation of the analytic function $f(x, y, z) = \sin\left(\log(x^2 e^{yz^3})\ \cos\left(x^3 y^2 z^4\right)\right)$ to compute all 7'th order derivatives.

Results obtained with the previous implementation are compared to derivatives computed with the symbolic library Sympy [73]. A total of 120 coefficients containing the derivatives up to 7'th order are retrieved. The relative error of the result is computed as

$$\varepsilon_p = \max\left(\frac{\partial^p f_{\text{Sympy}} - \partial^p f_{\text{pyOTI}}}{\partial^p f_{\text{Sympy}}}\right) \tag{4.85}$$

where ε_p represents the maximum relative error obtained in any p'th order derivative. The maximum relative errors are shown in Table 4.7.

4.4.4 Comparison of OTI implementation with an automatic differentiation library.

OTI Python implementation pyOTI was compared to pyAuDi [6], an automatic differentiation tool that implements Truncated Taylor Polynomials in a sparse manner. pyAuDi's core is implemented in C++ and wrapped to Python. The version used for this comparison was pyAuDi 1.6.5 which supports threaded parallelism and it is claimed to provide fast overall performance compared to other Truncated Taylor Polynomial

Order of Derivative (p)	ε_p
1	1.3585e-16
2	1.3585e-16
3	6.1473e-16
4	6.1473e-16
5	1.7288e-15
6	1.7288e-15
7	5.0329e-15

Table 4.7: Maximum relative error per order of derivatives obtained after evaluating function $f(x, y, z) = \sin\left(\log(x^2 e^{yz^3})\cos\left(x^3 y^2 z^4\right)\right)$. Relative error compared with the derivatives obtained using Sympy.

implementations [6].

The comparison was performed using the m variable function $f : \mathbb{R}^m \to \mathbb{R}$ shown in equation 4.86. Derivatives of up to 4th order and up to 50 variables were computed.

$$f(\mathbf{x}) = \frac{\sin\left(\prod_{l=1}^{m}\left(x_l^{10}\right)\right)\cos\left(\prod_{l=1}^{m} x_l\right)}{\prod_{l=1}^{m} x_l} \tag{4.86}$$

The computational performance was compared by computing the CPU time ratio r_m^n of the CPU time spent by pyAuDi divided by the CPU time spent by pyOTI to compute derivatives of m variables of n'th order.

$$r_m^n = \frac{\text{CPU}_{\text{pyAuDi}}}{\text{CPU}_{\text{pyOTI}}} \tag{4.87}$$

Results are shown in Table 4.8. It can be observed that the performance ratio in every computation case evaluated is greater than one, meaning pyOTI was faster in every case. The minimum ratio is obtained when computing fourth order derivatrives of single variable, where pyOTI performed $7.3\times$ faster than pyAuDi. Moreover, the largest ratio is observed when computing second order derivatives of 50 variables to the function in equation (4.86), when pyAuDi used $187.1\times$ the time pyOTI used for the same computation.

4.4.5 Numerical Example of OTI Reduced Order Model.

This example is intended to evaluated the performance of pyOTI when used to generate reduced order models of functions, as described in section 4.2.5. Consider the two variable function $f : \mathbb{R}^2 \to \mathbb{R}$,

Number of Variables (m)	Order (n) 1	2	3	4
1	11.8	11.2	8.8	7.3
2	15.3	13.4	10.5	10.1
5	19.6	21.9	24.8	24.2
10	28.6	48.7	65.4	30.9
20	48.5	106.1	63.7	16.6
30	66.3	133.9	65.2	22.8
50	125.6	187.1	160.9	25.4

Table 4.8: CPU time comparison of current implementation of pyOTI vs pyAuDi runtime for evaluation of function $f(\mathbf{x}) = \sin\left(\prod_{i=1}^{m}\left(x_i^{10}\right)\right)\cos\left(\prod_{i=1}^{m} x_i\right)/\prod_{i=1}^{m} x_i$ computing all derivatives of m variables up to order n.

$$f(x, y) = \log(xy) \tag{4.88}$$

The goal is to generate an 50'th order OTIROM centered at $x = 0.5$ $y = 0.8$, and analyze the behavior in this numerically challenging function. This function is chosen because the exact high order mixed derivatives of $f(x, y)$ are zero. This can be seen by expanding the logarithm as follows

$$f(x, y) = \log(x) + \log(y) \tag{4.89}$$

which shows more clearly that the there are no mixed terms. If evaluated in the form of equation (4.88), referred here as the compact form of the function, OTI algebra is not 'aware' that the mixed derivatives are zero, thus its best efforts will lead to computing the mixed derivatives with machine precision (which induces an error). The question that arises is whether the values obtained with machine precision will propagate the error to the high order derivatives and make the results of the OTIROM unusable.

In order to generate an OTIROM, OTI perturbations are applied to the evaluation point number representing the function is found by perturbing the input variables in OTI imaginary directions and truncation order is set to the desired order, in this case $n = 50$,

$$x^* = 0.5 + \epsilon_1,\ x^* \in \mathbb{OTI}_2^{50} \tag{4.90}$$

$$y^* = 0.8 + \epsilon_2,\ y^* \in \mathbb{OTI}_2^{50} \tag{4.91}$$

then an OTI number is generated by evaluating the function at the perturbed input values

$$g^* = f(x^*, y^*) \tag{4.92}$$

Finally, the OTIROM is evaluated at the required input variations Δx and Δy, as

$$f(x + \Delta x, y + \Delta y) \approx \text{OTIROM}\left(g^*, \{\Delta x, \Delta y\}\right) \tag{4.93}$$

Figure 4.14 sketches the code used to generate the 50'th order OTIROM of the compact form of the function using pyOTI sparse implementation. In particular, line 12 performs the OTIROM evaluation at specific variable deltas $\Delta x = 0.5$, $\Delta y = 0.8$. Evaluation of the expanded function led to a sparse OTI with 100 non-zero imaginary coefficients while evaluation of the compact function led to 1325 imaginary coefficients stored.

```
1  # Make OTI library available
2  from otilib import e, log
3
4  # Define the variables by perturbing them
5  # in OTI directions.
6  x = 0.5 + e( 1, order = 50)
7  y = 0.8 + e( 2, order = 50)
8
9  # Compute the desired function.
10 g = log( x * y )
11
12 f_new = g.rom_eval([1,2],[0.5,0.8])
```

Figure 4.14: Sample code for the generation and evaluation of an 50th order OTIRO for function $f(x, y) = \log(xy)$ using pyOTI. The evaluation point, as implemented in this code example, is $x' = 0.5$, $y' = 0.8$ and the approximated evaluation point is $x' = 1.0$, $y' = 1.6$.

The behavior of the even order mixed derivatives $f_{x^p y^p}$ is shown in Figure (4.15) for evaluations of the function in compact and expanded forms, equations (4.88-4.89). It can be observed that the second order mixed derivatives are exactly zero for both evaluations. However, the fourth order mixed derivative of the initial function is not zero but a value that is 'numerically' zero, i.e. meaning machine error. The propagation of this error is exponential with respect to the order of derivative. The maximum error achieved was 10^{60} at the maximum evaluated order of 50. In contrast, evaluation of the expanded function (equation (4.89)) leads to mixed order derivatives that are exactly zero.

The behavior of the OTIROM approximation to the function is assessed by evaluating the relative error of the OTIROM result with respect to the function evaluated at the corresponding point, using truncation orders ranging from 1 to 50. Figure 4.16a shows the contours delimiting the regions where the relative

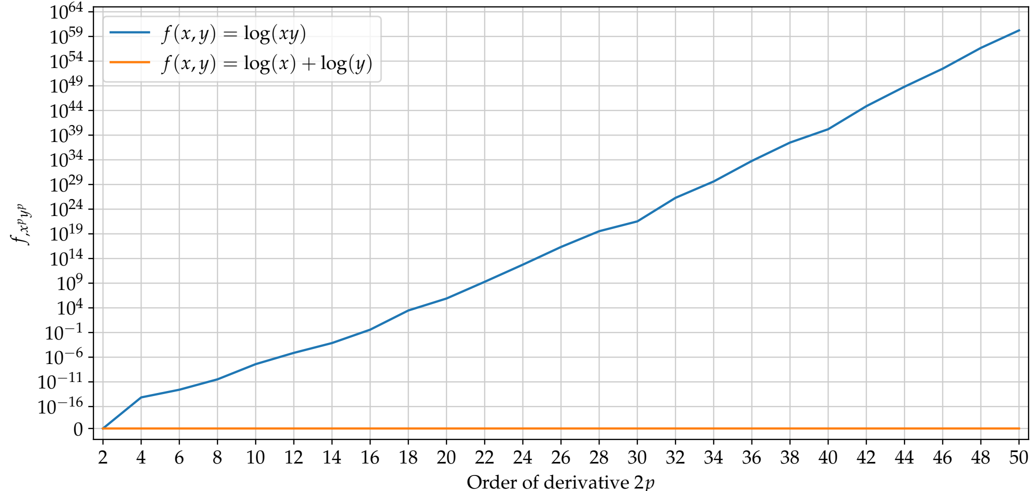


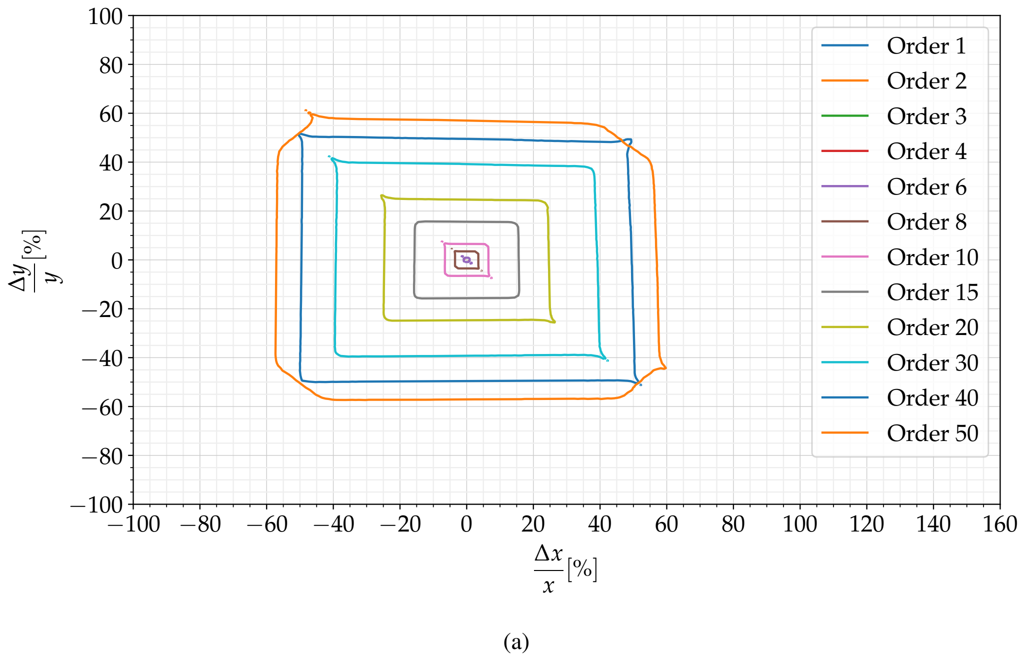
Figure 4.15: Values of the even $2p$'th mixed derivatives of function $f(x, y) = \log(xy)$ by evaluating it using the traditional form and the expanded form using pyOTI.

error of the reduced order model is below 10^{-14} for p'th order OTIROM using the expanded form of the function, equation (4.88). On the other hand, Figure 4.16b shows the contours delimiting the region where the relative error of the p'th order OTIROM is below 10^{-14} for the expanded function in equation (4.89). For instance, the 40'th order OTIROM can be evaluated with confidence of obtaining errors below 10^{-14} for $x' = x \pm 50\%x$ and $y' = y \pm 50\%y$.

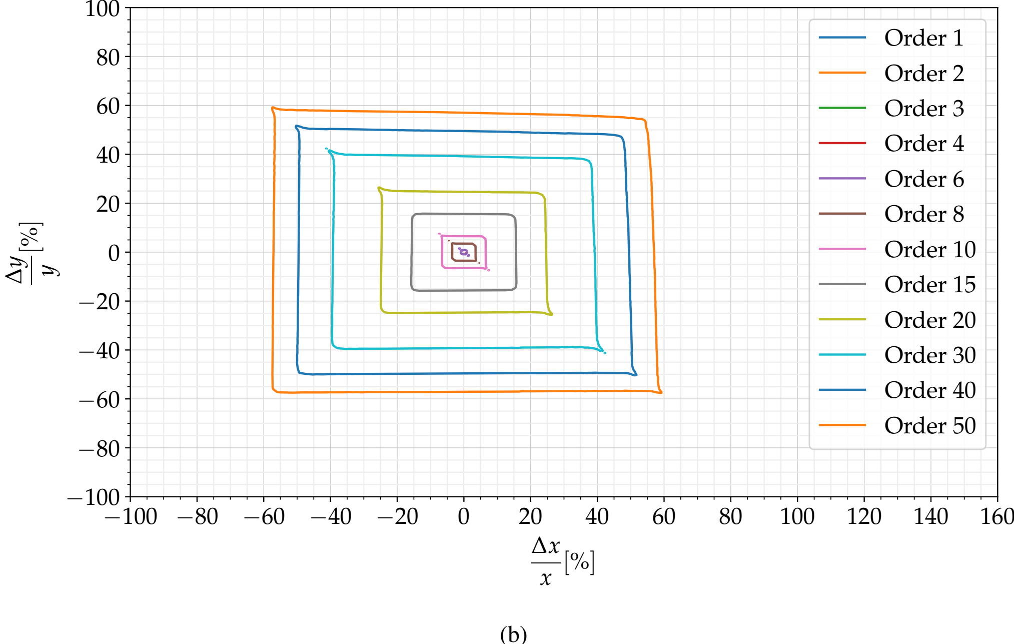
Although the mixed order derivatives deviates from the exact result when evaluating the compact form of the function, it does not affect in a significant manner the approximation accuracy for OTI evaluations with $n = 40$ or lower. The case of the 50th order model is affected mostly in the 'edges' of the region, more specifically the positive diagonal direction (~45º angle) of the input variable variations, which are values that require better accuracy of the mixed derivatives because they are more sensitive to its values as it variates both variables simultaneously.

4.5 Discussion

The Order Truncated Imaginary numbers were developed to compute high order multivariable derivatives efficiently because the size of the algebra adapts to the number of derivatives required. As the order of required derivatives increases, the number of required hypercomplex directions grows. The grown rate of OTI directions is significantly lower when compared with other algebras. The capacity to evaluate the complete set of m variable n'th order derivatives in a single evaluation also reduces the implementation



(a)



(b)

Figure 4.16: Contour lines delimiting the regions where the relative error of the OTI reduced order model is below 10^{-14} of the function (a) $f(x, y) = \log(xy)$ and (b) $f(x, y) = \log(x) + \log(y)$ using pyOTI.

complexity of computing higher order derivatives, as minimal code is required to set-up the evaluation and post-process the code.

As observed in section 4.4.5, a correct selection of the evaluation function may lead to better accuracy for high order derivatives. The accumulation of machine error through all derivatives, specially if derivatives are 'machine-error' zero, is exponential with respect to the order of derivative. It may affect the approximation of OTIROMs for high order approximations. Multiduals may theoretically be capable of computing 50th order derivatives, but practically a multidual with 50 imaginary basis will have a total number of coefficients of 2^{50}, which requires 8 PB (Peta Bytes) to store all its coefficients. By contrast, a sparse OTI number with 1325 imaginary directions can compute and store all two variable 50'th order derivatives, requiring approximately 27 kB. This shows that OTI enable computations of derivatives that were not possible before with other hypercomplex algebras.

Although the implementation of the algebra in pyOTI is serial, the sparse implementation of the algebra is faster in all situations studied compared to automatic differentiation parallel implementations. Better performance is expected after the parallelization of core algorithms is performed in future updates.

Chapter 5: ORDER TRUNCATED IMAGINARY FINITE ELEMENT METHOD

This chapter describes the Order Truncated Imaginary Finite Element Method (OTIFEM). The integration of OTI algebra into the Finite Element operations is highly based on the general Hypercomplex Finite Element Method (ZFEM), described in section 3.5.1. However, differences arise in terms of the capabilities of the algebra, interface with real linear algebra solvers and some performance considerations are discussed in this chapter.

Section 5.2 presents the block solver method where instead of using the full matrix form of OTI numbers to interface with real linear systems of equations solvers, the resulting hypercomplex system of equations is reformulated into a set of smaller subsystems that share the same matrix of coefficients. This allows to use LU or Cholesky decomposition methods to solve linear system of equations.

OTIFEM allows not only to compute high order derivatives of the solution the finite element method with respect to shape, material, boundary condition inputs, etc; but also to generate reduced order models of the solution with the naturally occurring Taylor series expansion contained in the OTI result, applying the method described in section 4.2.5.

Section 5.4, presents some numerical examples using different physics. Heat transfer, linear elasticity and fluid dynamics problems are solved using OTI numbers and derivatives up to 30th order are calculated be computed using the developed method and that . The be applied to various physics.

Also, OTI algebra is used to compute the derivatives required for the elemental computations of FEM, in order to simplify programming. For example, the Jacobian of the element transformation for isoparametric elements is formed by perturbing the basis functions of the element at OTI-perturbed natural coordinates.

5.1 OTI Finite Element Method

The integration of OTI numbers in the Finite Element Method (FEM) is heavily associated to the Hypercomplex Finite Element Method (ZFEM), described in section 3.5.1. Additional to support perturbations of the input variables of the FEM problem, OTI numbers are also used to compute the derivatives of the elemental basis functions, required in finite element computations.

The hypercomplexification of the FEM allows computation of derivatives by uplifting every input variable of interest ϕ_i to the hypercomplex algebra of choice. OTI Finite Element Method (OTIFEM) allows computation of high order multivariable derivatives, by applying OTI perturbations to the input variables of

the problem, in the form

$$\phi_1^* = \phi_1 + \epsilon_1, \quad \phi_1^* \in \mathbb{OTI}_m^n \tag{5.1}$$

$$\phi_2^* = \phi_2 + \epsilon_2, \quad \phi_2^* \in \mathbb{OTI}_m^n \tag{5.2}$$

$$\vdots \tag{5.3}$$

$$\phi_m^* = \phi_m + \epsilon_m, \quad \phi_m^* \in \mathbb{OTI}_m^n \tag{5.4}$$

where the truncation order n corresponds to the maximum order of derivative required in the computation and the number of basis m is the number of variables to compute derivatives, see chapter 4. The input variable ϕ_i depends on the problem, see appendix D, and can represent a shape, material, boundary condition input variable of the problem. Note that the real part of the input variable perturbed correspond to the real input value for the traditional finite element method.

Due to OTI perturbations, the OTIFEM analysis generates an OTI system of equations

$$\mathbf{K}^*\mathbf{u}^* = \mathbf{f}^* \tag{5.5}$$

where every element in $\mathbf{K}^*$, $\mathbf{u}^*$ and $\mathbf{f}^*$ is an OTI number. The solution of equation 5.5 is discussed in section 5.2.

Vector $\mathbf{u}^*$ is the hypercomplexified nodal solution vector which represents either the nodal temperature if solving a heat transfer analysis, the nodal displacements if solving a linear elastic problem or the nodal velocity and pressure in fluid dynamics analysis. It contains the derivatives with respect to the perturbed input parameters.

In the case that two input variables were perturbed as $\phi_1^* = \phi_1 + \epsilon_1$ and $\phi_2^* = \phi_2 + \epsilon_2$ both with truncation order 2. The solution $\mathbf{u}^*$ can be used to generate a reduced order model of the function $\mathbf{u}$ for the input variables perturbed, in this case ϕ_1 and ϕ_2, as described in section 4.2.5.

$$\mathbf{u}^* = \mathbf{u}_r + \mathbf{u}_{\epsilon_1}\epsilon_1 + \mathbf{u}_{\epsilon_2}\epsilon_2 + \mathbf{u}_{\epsilon_1^2}\epsilon_1^2 + \mathbf{u}_{\epsilon_1\epsilon_2}\epsilon_1\epsilon_2 + \mathbf{u}_{\epsilon_2^2}\epsilon_2^2 \tag{5.6}$$

where the imaginary coefficients from the result are additional degrees of freedom added to the problem, as

sketched in Figure 5.1, containing the derivatives of $\mathbf{u}$ with respect to the perturbed inputs.

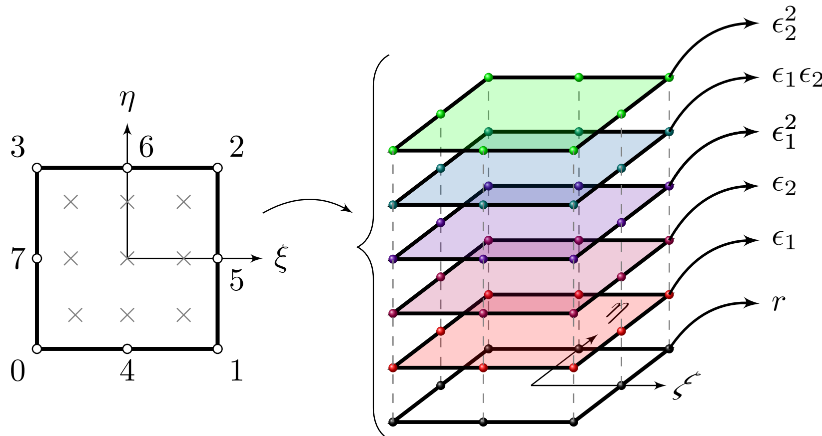


Figure 5.1: 2D 8-noded OTI quadrilateral element with the corresponding duplicates given by the imaginary directions ϵ_1, ϵ_2, etc, in natural coordinates and the 3×3 full integration scheme. Other elements for 1D and 3D analyses can be similarly constructed.

The OTI perturbed finite element solution $\mathbf{u}^*$ can be used to create an OTIROM of the traditional finite element solution with respect to the perturbed input parameters. As a consequence, evaluation of the OTIROM becomes

$$\mathbf{u}|_{\phi_1+\Delta\phi_1,...,\phi_m+\Delta\phi_m} \approx \text{OTIROM}(\mathbf{u}^*, \{\Delta\phi_1, ..., \Delta\phi_m\}) \tag{5.7}$$

Additionally, the resulting $\mathbf{K}^*$ and $\mathbf{f}^*$ are expressed as follows:

$$\mathbf{K}^* = \mathbf{K}_r + \mathbf{K}_{\epsilon_1}\epsilon_1 + \mathbf{K}_{\epsilon_2}\epsilon_2 + \mathbf{K}_{\epsilon_1^2}\epsilon_1^2 + \mathbf{K}_{\epsilon_1\epsilon_2}\epsilon_1\epsilon_2 + \mathbf{K}_{\epsilon_2^2}\epsilon_2^2 \tag{5.8}$$

$$\mathbf{f}^* = \mathbf{f}_r + \mathbf{f}_{\epsilon_1}\epsilon_1 + \mathbf{f}_{\epsilon_2}\epsilon_2 + \mathbf{f}_{\epsilon_1^2}\epsilon_1^2 + \mathbf{f}_{\epsilon_1\epsilon_2}\epsilon_1\epsilon_2 + \mathbf{f}_{\epsilon_2^2}\epsilon_2^2 \tag{5.9}$$

where $\mathbf{K}_r$ is the matrix with all the real coefficients of the elements in $\mathbf{K}^*$; $\mathbf{K}_{\epsilon_1}$ is the matrix containing all coefficients in direction ϵ_1, etc. For example, matrix $\mathbf{K}_{\epsilon_1^2}$ and vector $\mathbf{f}_{\epsilon_1^2}$ contain the second order derivatives of $\mathbf{K}$ and $\mathbf{f}$ with respect to ϕ_1,

$$\mathbf{K}_{,\phi_1^2} = 2!\text{Im}_{\epsilon_1^2}[\mathbf{K}^*], \quad \mathbf{f}_{,\phi_1^2} = 2!\text{Im}_{\epsilon_1^2}[\mathbf{f}^*] \tag{5.10}$$

In particular, shape perturbations that deform the domain boundary limits are performed in the way depicted in Figure 5.2. To compute the derivative of the state function $\mathbf{u}$ with respect to the horizontal dimension of the domain L_x, then a perturbation function is applied in the normal direction of the boundary, in this case in x direction, as shown in Figure 5.2a. The dimension of the region l_x of elements to which the perturbation is applied is selected by the user. The perturbation can be seen as a virtual displacement of the domain in the imaginary direction. Analogously, perturbation can be applied to compute derivatives with respect to the height L_y of the domain, in y direction, as shown in Figure 5.2b. If both perturbations discussed are applied in two independent imaginary directions, derivatives with respect to both boundary movements are obtained, including the mixed derivatives if the truncation order of the algebra greater than 2.

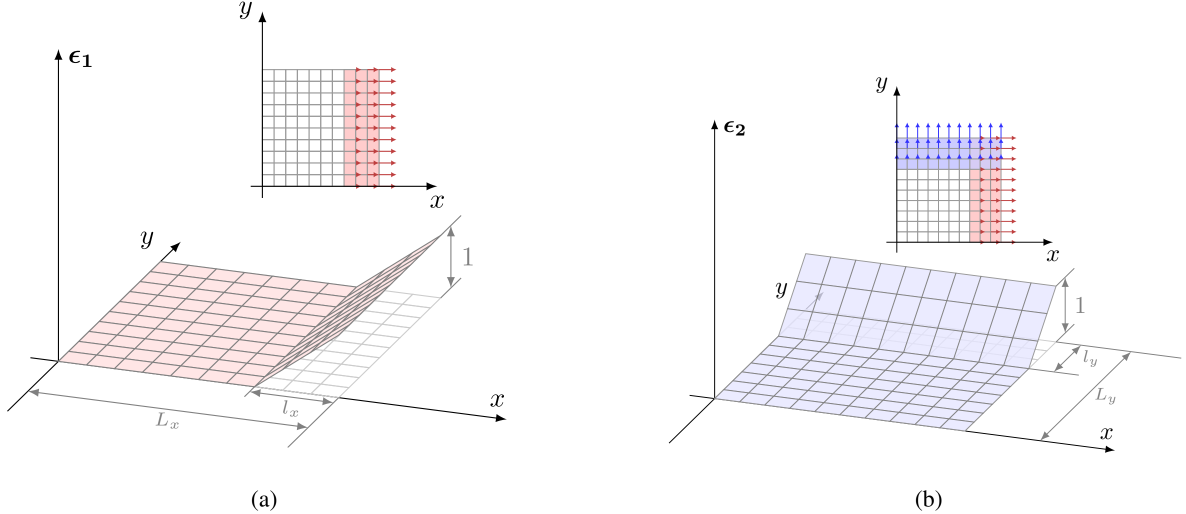


Figure 5.2: Sketch of the OTI nodal perturbations of a rectangular mesh for computation of shape derivatives with respect to (a) the base L_x and (b) the hight L_y of the domain.

In addition to the traditional ZFEM computations, OTI numbers are used to compute the derivatives of the basis functions. The computation of derivatives of the basis functions with respect to the natural coordinates is common to isoparametric and affine formulations. For instance, the isoparametric definition of the coordinate transformation from the natural coordinate space to the global coordinate space as

$$\mathbf{x}^*(\boldsymbol{\xi}_l) = \sum_i N_i(\boldsymbol{\xi}_l)\mathbf{x}_i^* \tag{5.11}$$

where N_i is a real valued basis function associated to node i and $\boldsymbol{\xi}_l = [\xi_l, \eta_l, \zeta_l]$ is the l'th integration point natural coordinate and $\mathbf{x}_i^* = [x_i^*, y_i^*, z_i^*]$ is the i'th OTI perturbed nodal coordinate in the global cartesian space.

The computation of the gradient of $\mathbf{x}^*$ with respect to the natural coordinates, e.g. $\mathbf{x}_{,\xi}(\boldsymbol{\xi}_l)$, is performed as follows

$$\mathbf{x}_{,\xi}^*(\boldsymbol{\xi}_l) = \sum_i N_{i,\xi}(\boldsymbol{\xi}_l)\, \mathbf{x}_i^*, \quad \mathbf{x}_{,\eta}^*(\boldsymbol{\xi}_l) = \sum_i N_{i,\eta}(\boldsymbol{\xi}_l)\, \mathbf{x}_i^*, \quad \mathbf{x}_{,\zeta}^*(\boldsymbol{\xi}_l) = \sum_i N_{i,\zeta}(\boldsymbol{\xi}_l)\, \mathbf{x}_i^* \tag{5.12}$$

where $N_{i,\xi}$ is the derivative of the i'th basis function with respect to ξ, etc. Traditional FEM and ZFEM methods require an explicit definition of the derivatives of the basis functions. However, taking advantage of OTI capabilities and since the basis function N_i is explicitly defined as a function of the natural coordinates, OTI numbers are used to compute the derivatives as

$$N_i^* = N_i(\boldsymbol{\xi}_l^*) \tag{5.13}$$

where $\boldsymbol{\xi}^*$ is the OTI-perturbed nodal coordinate values, defined as

$$\boldsymbol{\xi}_l^* = [\xi_l^*, \eta_l^*, \zeta_l^*]^T = [\xi_l + \epsilon_1, \eta_l + \epsilon_2, \zeta_l + \epsilon_3]^T \tag{5.14}$$

where $\xi_l^*, \eta_l^*, \zeta_l^* \in \mathbb{OTI}_3^1$. Higher order derivatives with respect to the natural coordinates can be achieved if the truncation order is increased, but for most finite element formulations higher order derivatives are not necessary. Derivatives are retrieved following equation 4.61,

$$N_i(\boldsymbol{\xi}_l) = \text{Re}\left[N_i(\boldsymbol{\xi}_l^*)\right] \tag{5.15}$$

$$N_{i,\xi}(\boldsymbol{\xi}_l) = \text{Im}_{\epsilon_1}\left[N_i(\boldsymbol{\xi}_l^*)\right] \tag{5.16}$$

$$N_{i,\eta}(\boldsymbol{\xi}_l) = \text{Im}_{\epsilon_2}\left[N_i(\boldsymbol{\xi}_l^*)\right] \tag{5.17}$$

$$N_{i,\zeta}(\boldsymbol{\xi}_l) = \text{Im}_{\epsilon_3}\left[N_i(\boldsymbol{\xi}_l^*)\right] \tag{5.18}$$

Note that the computation of the i'th basis function and its derivatives is performed in a single evalu-

ation of the basis function, as in equation 5.13, for every evaluation point $\boldsymbol{\xi}_l$ required. This simplifies the implementation of higher order elements and new elements, as only the nodal basis functions need to be defined. Its derivatives are obtained by evaluating them using OTI numbers.

Computation of the Jacobian follows as in the standard FEM, equation 3.41. The inverse of the Jacobian $\mathbf{J}^{*-1}$ is required to transform the derivatives of the basis functions in terms of the global coordinates x, y, z, as follows

$$\begin{Bmatrix} N_{i,x} \\ N_{i,y} \\ N_{i,z} \end{Bmatrix} = [\mathbf{J}^*]^{-1} \begin{Bmatrix} N_{i,\xi} \\ N_{i,\eta} \\ N_{i,\zeta} \end{Bmatrix} \tag{5.19}$$

where $\mathbf{J}^{*-1}$ can be computed with conventional algorithms, using OTI algebra. Equation 5.20 illustrates the computation of the inverse Jacobian for a two dimensional problem with a 2×2 Jacobian.

$$\mathbf{J}^{*-1} = \begin{bmatrix} J^*_{11} & J^*_{12} \\ J^*_{21} & J^*_{22} \end{bmatrix} = \frac{1}{|\mathbf{J}^*|} \begin{bmatrix} J^*_{22} & -J^*_{12} \\ -J^*_{21} & J^*_{11} \end{bmatrix} \tag{5.20}$$

where the determinant $|\mathbf{J}^*|$ can be computed as in equation 3.46. An alternative methodology to compute the inverse of OTI matrices is proposed in section 5.2, that allows the use of conventional real matrix inverse programs.

5.2 Solution of Linear Systems of OTI Equations

Many problems are solved by means of solving Linear System of Equations (LSE), see equation (5.21). A popular example is the Finite Element Method, as seen in section . The buildup of the matrix of coefficients $\mathbf{K} \in \mathbb{R}^{(d \times d)}$ or vector of independent variables $\mathbf{f} \in \mathbb{R}^{(d)}$ may depend on certain variables, making the solution $\mathbf{u} \in \mathbb{R}^{(d)}$ sensitive to them (d is the number of degrees of freedom of the linear system of equations). Thus, in some applications, it is of interest to compute the derivatives of the solution vector $\mathbf{u}$ with respect to the input variables. This means that derivatives of the solution vector $\mathbf{u}$ can potentially be computed by hypercomplexifying the input variables, transforming the standard system into a hypercomplex linear system of equations (HLSE), where every element of $\mathbf{K}$, $\mathbf{u}$ and $\mathbf{f}$ are OTI numbers, as in equation (5.22).

$$\mathbf{K}\mathbf{u} = \mathbf{f} \tag{5.21}$$

$$\mathbf{K}^*\mathbf{u}^* = \mathbf{f}^* \tag{5.22}$$

where $\mathbf{K}^* \in \mathbb{OTI}_m^{n\,(d\times d)}$, $\mathbf{f}^* \in \mathbb{OTI}_m^{n\,(d)}$ and $\mathbf{u}^* \in \mathbb{OTI}_m^{n\,(d)}$.

The hypercomplex linear system of equations (5.22) can not be solved as is using conventional real linear algebra libraries because the elements of matrix $\mathbf{K}^*$ and vectors $\mathbf{u}^*$ and $\mathbf{f}^*$ are OTI numbers. However, as described in section 4.2.6, an OTI number has equivalent matrix and vector forms with all real coefficients. A real only system of equations can be obtained by transforming each member of $\mathbf{K}^*$ into its matrix form and every member of $\mathbf{u}^*$ and $\mathbf{f}^*$ into its vector form. If input variables consist of OTI numbers in $\mathbb{OTI}_2^2$, then the system of equations can be reorganized as:

$$\overbrace{\begin{bmatrix} \mathbf{K}_r & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{K}_{\epsilon_1} & \mathbf{K}_r & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{K}_{\epsilon_2} & \mathbf{0} & \mathbf{K}_r & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{K}_{\epsilon_1^2} & \mathbf{K}_{\epsilon_1} & \mathbf{0} & \mathbf{K}_r & \mathbf{0} & \mathbf{0} \\ \mathbf{K}_{\epsilon_1\epsilon_2} & \mathbf{K}_{\epsilon_2} & \mathbf{K}_{\epsilon_1} & \mathbf{0} & \mathbf{K}_r & \mathbf{0} \\ \mathbf{K}_{\epsilon_2^2} & \mathbf{0} & \mathbf{K}_{\epsilon_2} & \mathbf{0} & \mathbf{0} & \mathbf{K}_r \end{bmatrix}}^{\mathbf{T}(\mathbf{K}^*)} \overbrace{\begin{Bmatrix} \mathbf{u}_r \\ \mathbf{u}_{\epsilon_1} \\ \mathbf{u}_{\epsilon_2} \\ \mathbf{u}_{\epsilon_1^2} \\ \mathbf{u}_{\epsilon_1\epsilon_2} \\ \mathbf{u}_{\epsilon_2^2} \end{Bmatrix}}^{\mathbf{t}(\mathbf{u}^*)} = \overbrace{\begin{Bmatrix} \mathbf{f}_r \\ \mathbf{f}_{\epsilon_1} \\ \mathbf{f}_{\epsilon_2} \\ \mathbf{f}_{\epsilon_1^2} \\ \mathbf{f}_{\epsilon_1\epsilon_2} \\ \mathbf{f}_{\epsilon_2^2} \end{Bmatrix}}^{\mathbf{t}(\mathbf{f}^*)} \tag{5.23}$$

where $\mathbf{K}_r$ is the conventional real matrix of coefficients, $\mathbf{K}_{\epsilon_1}$ is the matrix of coefficients of imaginary direction ϵ_1, $\mathbf{0}$ is a matrix of zeros of same shape as $\mathbf{K}_r$, etc. For this example the transformation $\mathbf{T}(\mathbf{K}^*) \in \mathbb{R}^{(6d\times 6d)}$ has shape 6 times larger than the real LSE, meaning it has 6 times the number of degrees of freedom. This size increases as the truncation order and number of bases increase in the algebra, which would make the solution of the system of equations for large systems computationally expensive. In general, the number of degrees of freedom becomes $d\,N$, and the transformations become $\mathbf{T}(\mathbf{K}^*) \in \mathbb{R}^{(Nd\times Nd)}$ and $\mathbf{t}(\mathbf{u}^*) \in \mathbb{R}^{(Nd)}$.

Notice, however, that the new system of equations is *lower triangular* due to the nature of the matrix form of OTI numbers. Therefore, it is possible to subdivide the system into *blocks* and generate a set of equations of smaller size, using a forward substitution scheme.

$$\mathbf{K}_r \mathbf{u}_r = \mathbf{f}_r \quad (5.24)$$

$$\mathbf{K}_r \mathbf{u}_{\epsilon_1} = \mathbf{f}_{\epsilon_1} - \mathbf{K}_{\epsilon_1} \mathbf{u}_r \quad (5.25)$$

$$\mathbf{K}_r \mathbf{u}_{\epsilon_2} = \mathbf{f}_{\epsilon_2} - \mathbf{K}_{\epsilon_2} \mathbf{u}_r \quad (5.26)$$

$$\mathbf{K}_r \mathbf{u}_{\epsilon_1^2} = \mathbf{f}_{\epsilon_1^2} - \mathbf{K}_{\epsilon_1} \mathbf{u}_{\epsilon_1} - \mathbf{K}_{\epsilon_1^2} \mathbf{u}_r \quad (5.27)$$

$$\mathbf{K}_r \mathbf{u}_{\epsilon_1 \epsilon_2} = \mathbf{f}_{\epsilon_1 \epsilon_2} - \mathbf{K}_{\epsilon_2} \mathbf{u}_{\epsilon_1} - \mathbf{K}_{\epsilon_1} \mathbf{u}_{\epsilon_2} - \mathbf{K}_{\epsilon_2^2} \mathbf{u}_r \quad (5.28)$$

$$\mathbf{K}_r \mathbf{u}_{\epsilon_2^2} = \mathbf{f}_{\epsilon_2^2} - \mathbf{K}_{\epsilon_2} \mathbf{u}_{\epsilon_2} - \mathbf{K}_{\epsilon_2^2} \mathbf{u}_r \quad (5.29)$$

Notice that the system of equations (5.23) is now transformed into 6 systems of equations all sharing the same matrix of coefficients $\mathbf{K}_r$ with the same shape as the original real only system in equation (5.21). Also, notice that system 5.24 is equivalent to equation (5.21) as the result is the same real LSE. It can be shown that the systems of equations (5.24-5.29) are equivalent to the result of direct differentiating the system of equations (5.21), but were obtained naturally from the OTI matrix form. Before, expressions to generate the computations had to be derived analytically, but with OTI matrix form it can be generated automatically for any order and number of variables.

All systems in equation (5.23) share the same matrix of coefficients $\mathbf{K}_r$, the standard real matrix problem. Because of that, the solution of the system of equations can be done using standard LU decomposition for non symmetric systems or Cholesky decomposition for symmetric ones. Therefore, a single decomposition of K_r is required to solve all 6 systems, which can significantly reduce the computational complexity to solve the system (5.22), compared with using the full system (5.23).

Additionally, notice that systems (5.25-5.29) all depend on $\mathbf{u}_r$. Hence, system (5.24) should be solved first. Systems (5.25) and (5.27) only depend on $\mathbf{u}_r$ to solve for $\mathbf{u}_{\epsilon_1}$ and $\mathbf{u}_{\epsilon_2}$ respectively. Therefore, they are independent of each other and thus both can be solved concurrently after solving $\mathbf{u}_r$. These two systems share the fact that both correspond to the solution of the directions of order one of the OTI algebra. Systems (5.26), (5.28) and (5.29) solve for $\mathbf{u}_{\epsilon_1^2}$, $\mathbf{u}_{\epsilon_1 \epsilon_2}$ and $\mathbf{u}_{\epsilon_2^2}$ respectively; and depend on the solution of the real and ϵ_1 solutions of $\mathbf{u}^*$. Since they are independent from each other, they can be solved concurrently after finding $\mathbf{u}_{\epsilon_1}$ and $\mathbf{u}_{\epsilon_2}$.

In general, the number of systems to be solved is given by the total number N of imaginary directions of the OTI algebra (see equation 4.13). The sequence to solve the problem is determined by the order of the

imaginary directions of the system. That is, first solve directions with order 0 (real solution), then solve all directions with order 1 (first order derivatives), then directions with order 2 (all second order derivatives), etc. A convenient way to reorganize the solution of the HLSE is to re-write the system of equations by order of imaginary directions as follows

$$\left[\mathbf{K}^*\right]^0\left[\mathbf{u}^*\right]^0 = \left[\mathbf{f}^*\right]^0 \tag{5.30}$$

$$\left[\mathbf{K}^*\right]^0\left[\mathbf{u}^*\right]^1 = \left[\mathbf{f}^*\right]^1 - \left[\mathbf{K}^*\right]^1\left[\mathbf{u}^*\right]^0 \tag{5.31}$$

$$\left[\mathbf{K}^*\right]^0\left[\mathbf{u}^*\right]^2 = \left[\mathbf{f}^*\right]^2 - \left[\mathbf{K}^*\right]^1\left[\mathbf{u}^*\right]^1 - \left[\mathbf{K}^*\right]^2\left[\mathbf{u}^*\right]^0 \tag{5.32}$$

$$\left[\mathbf{K}^*\right]^0\left[\mathbf{u}^*\right]^3 = \left[\mathbf{f}^*\right]^3 - \left[\mathbf{K}^*\right]^1\left[\mathbf{u}^*\right]^2 - \left[\mathbf{K}^*\right]^2\left[\mathbf{u}^*\right]^1 - \left[\mathbf{K}^*\right]^3\left[\mathbf{u}^*\right]^0 \tag{5.33}$$

$$\vdots$$

$$\left[\mathbf{K}^*\right]^0\left[\mathbf{u}^*\right]^n = \left[\mathbf{f}^*\right]^n - \sum_{i=1}^{n}\left[\mathbf{K}^*\right]^i\left[\mathbf{u}^*\right]^{n-i} \tag{5.34}$$

where $\left[\mathbf{K}^*\right]^0 = \mathrm{K}_r$ and the operation $[\cdot]^p$ extracts all p'th order imaginary directions of the given OTI array, as shown in equation (4.14). The system for imaginary directions of order p,

$$\left[\mathbf{K}^*\right]^0\left[\mathbf{u}^*\right]^p = \underbrace{\left[\mathbf{f}^*\right]^p - \sum_{i=1}^{p}\left[\mathbf{K}^*\right]^i\left[\mathbf{u}^*\right]^{p-i}}_{\left[\mathbf{rhs}^*\right]^p} \tag{5.35}$$

is solved by forming a real only system concatenating all imaginary directions of $[\mathbf{u}^*]^p$ horizontally, generating a matrix of shape $(d \times N_m^p)$. The same is applied to the right hand side $[\mathbf{rhs}^*]^p$, forming the following system of equations

$$\mathbf{K}_r\left[\mathbf{u}_{\boldsymbol{\alpha}_1^p}, \mathbf{u}_{\boldsymbol{\alpha}_2^p}, \ldots, \mathbf{u}_{\boldsymbol{\alpha}_{N_m^p}^p}\right] = \left[\mathbf{rhs}_{\boldsymbol{\alpha}_1^p}, \mathbf{rhs}_{\boldsymbol{\alpha}_2^p}, \ldots, \mathbf{rhs}_{\boldsymbol{\alpha}_{N_m^p}^p}\right] \tag{5.36}$$

which can be solved using the factorized $\mathbf{K}_r$. Modern real solvers can solve the system of equations (5.36) concurrently using optimized parallel procedures.

The inverse of a hypercomplex matrix can be computed as a particular case of this solution scheme, becoming:

$$\begin{aligned}
\left[\mathbf{K}^*\right]^0 \left[\mathbf{K}^{*\,-1}\right]^0 &= \mathbf{I} && (5.37)\\
\left[\mathbf{K}^*\right]^0 \left[\mathbf{K}^{*\,-1}\right]^1 &= -\left[\mathbf{K}^*\right]^1 \left[\mathbf{K}^{*\,-1}\right]^0 && (5.38)\\
&\vdots\\
\left[\mathbf{K}^*\right]^0 \left[\mathbf{K}^{*\,-1}\right]^n &= -\sum_{i=1}^{n} \left[\mathbf{K}^*\right]^i \left[\mathbf{K}^{*\,-1}\right]^{n-i} && (5.39)
\end{aligned}$$

where equation (5.37) solves the conventional inverse of the real part of K^*, K_r^{-1}, and then equations (5.38-5.39) solve the imaginary directions of $\mathbf{K}^{*-1}$. Since $\left[\mathbf{K}^{*\,-1}\right]^0 = \mathrm{K}_r^{-1}$ is computed explicitly, the imaginary coefficients can be computed as follows,

$$\left[\mathbf{K}^{*\,-1}\right]^n = -\left[\mathbf{K}^{*-\mathbf{1}}\right]^0 \left(\sum_{i=1}^{n} \left[\mathbf{K}^*\right]^i \left[\mathbf{K}^{*\,-1}\right]^{n-i}\right) \tag{5.40}$$

meaning that the imaginary coefficients of the inverse of matrix K^* are solved by means of matrix-matrix multiplications. This method can be used to accelerate computations of the Jacobian inverse in finite element computations (equation 5.19).

Since multidual numbers are a subset of OTI numbers, see section 4.2.7, the methods presented in this section can be applied also to multiduals. The derivation is trivial.

Example 8. Matrix inverse.

In this example the computation of a matrix inverse is performed. The matrix to be inverted is

$$\mathbf{A} = \begin{bmatrix} x^2 & y \\ 0 & xy \end{bmatrix} \tag{5.41}$$

The goal for this example is to compute all first and second order derivatives of the inverse matrix with respect to x and y for $x = 2$ and $y = 1$ using OTI algebra. The analytical solution for the inverse of this matrix is:

$$\mathbf{A}^{-1} = \begin{bmatrix} \frac{1}{x^2} & -\frac{1}{x^3} \\ 0 & \frac{1}{xy} \end{bmatrix} \tag{5.42}$$

In order to compute the derivatives of the inverse with respect to x and y, then OTI algebra is used. The

input variables x and y are perturbed in OTI bases ϵ_1 and ϵ_2, respectively, selecting OTI truncation order $n = 2$ as derivatives up to second order are required:

$$x^* = 2 + \epsilon_1,\ x^* \in \mathbb{OTI}_2^2 \tag{5.43}$$

$$y^* = 1 + \epsilon_2,\ y^* \in \mathbb{OTI}_2^2 \tag{5.44}$$

The hypercomplex form of A is therefore:

$$\mathbf{A}^* = \begin{bmatrix} 4 + 4\epsilon_1 + \epsilon_1^2 & 1 + \epsilon_2 \\ 0 & 2 + \epsilon_1 + 2\epsilon_2 + \epsilon_1\epsilon_2 \end{bmatrix} \tag{5.45}$$

Note that $[\mathbf{A}^*]^0$, $[\mathbf{A}^*]^1$ and $[\mathbf{A}^*]^2$ are:

$$[\mathbf{A}^*]^0 = \begin{bmatrix} 4 & 1 \\ 0 & 2 \end{bmatrix},\quad [\mathbf{A}^*]^1 = \begin{bmatrix} 4\epsilon_1 & \epsilon_2 \\ 0 & \epsilon_1 + 2\epsilon_2 \end{bmatrix},\quad [\mathbf{A}^*]^2 = \begin{bmatrix} \epsilon_1^2 & 0 \\ 0 & \epsilon_1\epsilon_2 \end{bmatrix} \tag{5.46}$$

The real part of $\mathbf{A}^{*-1}$ is computed using the conventional real procedure as it correspond to the inverse of the real part of $\mathbf{A}_r^{-1}$, yielding:

$$\left[\mathbf{A}^{*-1}\right]^0 = \mathbf{A}_r^{-1} = \begin{bmatrix} 0.25 & -0.125 \\ 0 & 0.5 \end{bmatrix} \tag{5.47}$$

Order 1 imaginary coefficients are obtained by

$$\left[\mathbf{A}^{*-1}\right]^1 = -\left[\mathbf{A}^{*-1}\right]^0 \left[\mathbf{A}^*\right]^1 \left[\mathbf{A}^{*-1}\right]^0 \tag{5.48}$$

producing the following result:

$$\left[\mathbf{A}^{*-1}\right]^{1} = -\begin{bmatrix} 0.25\epsilon_1 & -0.1875\epsilon_1 \\ 0 & 0.25\epsilon_1 + 0.5\epsilon_2 \end{bmatrix} \tag{5.49}$$

The order 2 imaginary coefficients of the inverse matrix are obtained

$$\left[\mathbf{A}^{*-1}\right]^{2} = -\left[\mathbf{A}^{*-1}\right]_{0}\left(\left[\mathbf{A}^{*}\right]^{1}\left[\mathbf{A}^{*-1}\right]^{1} + \left[\mathbf{A}^{*}\right]^{2}\left[\mathbf{A}^{*-1}\right]^{0}\right) \tag{5.50}$$

yielding the following result

$$\left[\mathbf{A}^{*-1}\right]^{2} = -\begin{bmatrix} -0.1875\epsilon_1^2 & 0.1875\epsilon_1^2 \\ 0 & -0.125\epsilon_1^2 - 0.25\epsilon_1\epsilon_2 - 0.5\epsilon_2^2 \end{bmatrix} \tag{5.51}$$

The final OTI form of the result is obtained as

$$\mathbf{A}^{*-1} = \left[\mathbf{A}^{*-1}\right]_{0} + \left[\mathbf{A}^{*-1}\right]_{1} + \left[\mathbf{A}^{*-1}\right]_{2} \tag{5.52}$$

$$\mathbf{A}^{*-1} = \begin{bmatrix} 0.25 - 0.25\epsilon_1 + 0.1875\epsilon_1^2 & -0.125 + 0.1875\epsilon_1 - 0.1875\epsilon_1^2 \\ 0 & 0.5 - 0.25\epsilon_1 - 0.5\epsilon_2 + 0.125\epsilon_1^2 + 0.25\epsilon_1\epsilon_2 + 0.5\epsilon_2^2 \end{bmatrix} \tag{5.53}$$

In order to retrieve the derivatives of $\mathbf{A}^{-1}$, the imaginary components of each element is extracted according to equation (4.61). Only the second order derivatives are shown below. The second order analytical derivatives of $\mathbf{A}^{-1}$ and its corresponding values for $x = 2$ and $y = 1$ are

$$\mathbf{A}^{-1}_{,x^2} = \quad (2!)\ \text{Im}_{\epsilon_1^2} \quad \left[\mathbf{A}^{*-1}\right] \quad = \begin{bmatrix} \dfrac{6}{x^4} & -\dfrac{12}{x^5} \\ 0 & \dfrac{2}{x^3 y} \end{bmatrix} = \begin{bmatrix} 0.375 & -0.375 \\ 0 & 0.25 \end{bmatrix} \tag{5.54}$$

$$\mathbf{A}^{-1}_{,xy} = (1!1!)\ \text{Im}_{\epsilon_1\epsilon_2} \left[\mathbf{A}^{*-1}\right] \quad = \begin{bmatrix} 0 & 0 \\ 0 & \dfrac{1}{x^2 y^2} \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 0.5 \end{bmatrix} \tag{5.55}$$

$$\mathbf{A}^{-1}_{,y^2} = \quad (2!)\ \text{Im}_{\epsilon_2^2} \quad \left[\mathbf{A}^{*-1}\right] \quad = \begin{bmatrix} 0 & 0 \\ 0 & \dfrac{2}{x\ y^3} \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \tag{5.56}$$

Using OTI, the derivatives are retrieved using equation (4.61) and correspond to the exact values of the evaluation analytical solution.

5.3 Computational Implementation

Linear algebra and finite element modules were added to the OTI library. This section briefly describes the capabilities added to the pyOTI library for computation and solution of finite element problems.

5.3.1 Linear algebra support

Linear algebra routines are supported for both static dense and sparse data structures at C-level, see section 4.3. An interface is provided for Python use, following a similar syntax to Numpy [72]. Element-wise operations such as addition, subtraction, multiplication and division are supported for matrix and vector structures. Also, element-wise evaluation of elementary functions is supported for all functions in Table 4.5. Select linear algebra operations are also supported for matrices/vectors whose elements are OTI numbers, which include matrix-matrix multiplication, dot product, determinant, p-norm, matrix inverse and transpose. Solution of linear systems of equations and matrix inverse are performed using the block-solver approach described in section 5.2.

Sparse matrices of OTI numbers are also supported in two representations: linked list (LIL) format [74], useful for matrix creation; and compressed sparse row (CSR) format [75], useful for matrix-vector multiplication. Support of sparse linear algebra operations is limited in the current version of pyOTI, but minimal support for CSR matrix - dense matrix multiplication is provided.

All linear algebra operations, to the exception of the external matrix factorization algorithms, are currently implemented as serial routines. It is expected to add support for parallelism in the near future. The factorization algorithms for solution of sparse linear systems of equations supported are: for LU decomposition, SciPy's distribution of SuperLU [76] and UMFPACK [77]. Cholesky factorization is supported through the scikit-sparse package [78]. All SuperLU, UMFPACK and Cholesky factorization algorithms are incorporated in the standard distribution. All three decomposition programs support parallelism with multithreading.

A real only linear algebra module is integrated to pyOTI with the same function and operator overloading support as for the OTI array support, used for comparison purposes. It is highly based in C double precision arithmetics and is implemented with serial routines. This was developed for comparison purposes because other linear algebra libraries (in particular Numpy), present a high overhead for matrix operations, highly significant for the CPU performance small matrices (e.g. matrices smaller than 20x20). This overhead is driven by, e.g. data type checks, rather than by the actual matrix operation. This overhead becomes predominant for applications such as elemental computations in finite element analysis. The developed real only routines minimizes the computational overhead.

5.3.2 Finite Element support

A finite element module was integrated into pyOTI. The module uses OTI algebra routines in the hypercompex finite element procedures. The finite element routines are defined in Python 3, using linear algebra operations associated to the selected algebra. Additionally, problems are defined using a syntax strongly associated to the mathematical formulation of the problem, similar to the syntax of FreeFem++ [79] and FEniCS [80]. Multi-physics problems can be simulated because problems are defined using the weak form of the partial differential equation. Currently, only partial differential equations of up to second order and up to three dimensions are supported.

Mesh creation is supported using the GMSH's [81] Python API, and visualization is performed using PyVista [82]. Multiple finite element types are supported for 1D, 2D and 3D analyses, including geometries like lines, triangles, tetrahedra, quadrilaterals and hexahedra. A complete list of the current supported Finite Elements is shown in Appendix E. Addition of new elements is easily achieved as only the main basis functions have to be defined. OTI algebra is used to compute the derivatives of the basis functions with respect to the natural coordinates. As an example, consider the 2-node linear element (see Figure E.1) with

basis functions

$$N_0(\xi) = \frac{1}{2}(1-\xi) \quad (5.57)$$

$$N_1(\xi) = \frac{1}{2}(1+\xi) \quad (5.58)$$

and the implementation of this element with pyOTI becomes

```
def line2_iso( xi, eta, zeta):

  N0 = 0.5 * (1.0 - xi)
  N1 = 0.5 * (1.0 + xi)

  return [ N0, N1 ]
```

Figure 5.3: Sample implementation of the 2-node element basis functions in natural coordinates for pyOTI.

The library will evaluate these elemental basis functions at the natural coordinates required for Gaussian integration with an imaginary perturbation like `xi = xi_r + e(1)`, where `xi_r` is the array with all the real natural coordinates to be evaluated, hence obtaining all required derivatives with respect to ξ. The other coordinates are zero for this element, but are required for compatibility across elements.

5.3.3 Usage of the library

The general steps to use the library are the following:

i) Import the library to use OTI capabilities,

ii) Define a discretization of the domain (mesh), T_h;

iii) Create a Finite Element space V_h, that assigns a discretization T_h with interpolation functions of a selected degree,

iv) Define the test and state functions of the problem,

v) Define the problem by its weak form equation,

vi) Assemble and solve the problem; and

vii) Export/use the results.

In order to better understand these steps, consider a 2D Laplace problem defined over a rectangular domain as shown in Figure 5.4,

$$-\Delta u = 0 \qquad \text{in } \Omega \tag{5.59}$$

$$u = g \qquad \text{on } \Gamma_{\text{left}} \tag{5.60}$$

$$u = 1 \qquad \text{on } \Gamma_{\text{right}} \tag{5.61}$$

$$\nabla u \cdot \mathbf{n} = 0 \qquad \text{on } \Gamma_{\text{top}} \cup \Gamma_{\text{bottom}} \tag{5.62}$$

where u is the state function, Δu is the laplacian of u, $u_{,x^2} + u_{,y^2}$; Ω is the domain of the problem (see Figure 5.4), Γ_{left}, Γ_{bottom}, Γ_{right} and Γ_{top} are boundary segments of Ω and $\nabla u \cdot \mathbf{n}$ is the normal derivative of the state function. The value of u at the left boundary is $g = 0$.

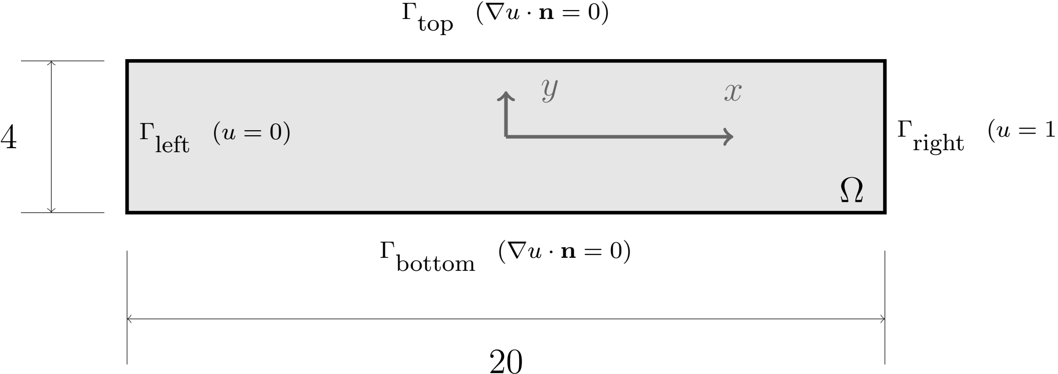


Figure 5.4: Example domain for Laplace problem.

The weak form of the problem [83] is defined as follows

$$\int_{\Omega} \nabla u \cdot \nabla v \, d\Omega - \cancel{\int_{\Gamma_N} (\nabla u \cdot \mathbf{n}) \, v \, dl} = 0 \qquad \rightarrow \qquad \int_{\Omega} \nabla u \cdot \nabla v \, d\Omega = 0 \tag{5.63}$$

where the boundary integral term cancels out because of the boundary conditions applied. OTIFEM is used to compute the sensitivity of u with respect to the left boundary condition g, i.e. $u_{,g}$.

The finite element solution is found by approximating the domain Ω by a discretization T_h made of elements Ω_e. Every function defined in T_h belongs to the functional space V_h that is defined by the linear combination of a specific set of basis function N_i. As a consequence, the state and test function become

$u_h, v_h \in V_h$, and the weak form is transformed into

$$\sum_e \int_{\Omega_e} \nabla u_h \cdot \nabla v_h \, d\Omega = 0$$

The program that evaluates the problem defined by Equation (5.63) is shown in Figure 5.5.

```
1  import pyoti.fem as fem           # Import pyoti FEM.
2  from pyoti.sparse import e        # import creator e.
3
4  Th = fem.square( 20, 4, he=1,    # Create a [20 x 4] square mesh
5                   quads = True )  # of 4 - node quads.
6
7
8  Vh = fem.fespace( Th, 1)          # Define a Finite Element space, establishing that
9                                    # its for 'first' order elements.
10
11 u = Vh()                          # Define a new function in the FE space.
12 v = Vh()                          # Define a new function in the FE space.
13
14 laplace = fem.feproblem( [ u ], [ v ],        #
15       int2d(   dx( u ) * dx( v )   ) +        #
16       int2d(   dy( u ) * dy( v )   ) +        # Generate the FE problem
17       on(  "left",   u, 0.0 + e(1) ) +        # <- OTI perturbation.
18       on( "right",   u, 1.0          )        #
19 )                                             #
20
21 laplace.solve(solver = 'SuperLU')             # Assemble and Solve problem
22                                               # using LU decomposition.
23
24 Th.export( "oti_laplace.vtk",       #
25            pd        = [  u  ],     # Export results
26            pd_names = [ 'u' ]    )  #
```

Figure 5.5: Example program that implements the 2D Laplace variational equation using the pyOTI Finite Element module.

Lines 1-2 step i, make pyOTI available to the current Python environment.

Lines 4-5 step ii, define the domain as a discretization T_h (`Th` in the code). It creates a rectangular mesh that defines the discretization shown in Figure 5.6.

Line 8 step iii, define the finite element space V_h, `Vh` in the code, associating the discretization T_h with an interpolation scheme, in this case 4 node quads `quad4`.

Lines 11-12 step iv, create the state and test functions that come from the finite element space V_h.

Lines 14-19 step v, define the problem using its weak form (5.63). Notice the similarity to the mathematical expression and notice that the Dirichlet (essential) boundary conditions are defined using the function `on(identifier, StateFunction, value)`. The input `identifier` is a string

or id associated to the boundary of interest. Input **`StateFunction`** is a state function variable such as the one defined in line 11, and **`value`** is the corresponding value at the domain, which can be a function (array of values) or a scalar as in the current example. In particular, since derivative $u_{,g}(x, y)$ is required, line 17 shows that the left boundary value is perturbed in the imaginary direction ϵ_1, using the sparse OTI implementation.

Line 21 step vi, assembles and solves the problem using SuperLU, as described in section 5.2. The solver assigns the numerical results to the state function **`u`** automatically.

Lines 24-26 step vii, export the solution in VTK [84] format. A new file is created to export results associated to T_h with name `"oti_laplace.vtk"`. The nodal data ("PointData" in VTK nomenclature) is an optional input **`pd`** that defines the list of nodal information to be exported. In can be vector or scalar valued array. The optional argument **`pd_names`** define a list of strings to name each array in the in **`pd`**.

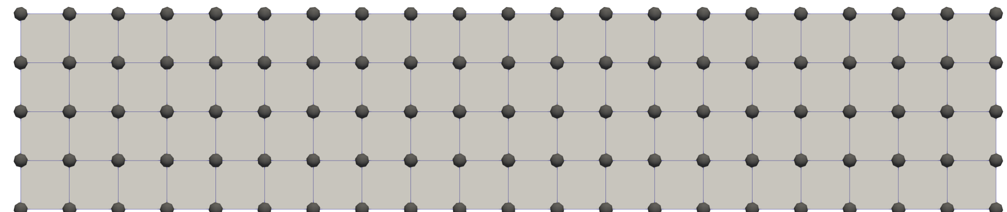


Figure 5.6: Mesh example for domain in the 2D Laplace equation.

The results of the analysis are shown in Figure 5.7.

5.3.4 Expression Equivalence

As seen in Section 5.3.3, the library syntax is related to the mathematical expressions of the variational formulation. To that end, Table 5.1 shows useful functions that resemble commonly used expression. The considerations to read Table 5.1 are:

- **`u`**: State function u,
- **`v`**: Test function v,
- **`g, f`**: Functions interpolated by a finite element space, i.e. array of values corresponding with all nodes in the elements of the mesh.

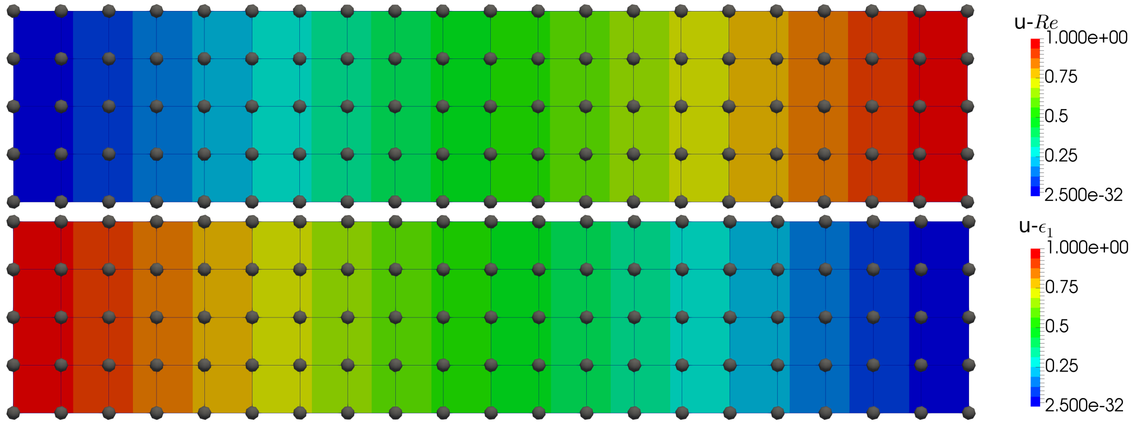


Figure 5.7: Result of the 2D Laplace problem contained in `"oti_laplace.vtk"`. The real solution (top figure) shows the standard solution of the problem. The imaginary solution of direction ϵ_1 (bottom figure) shows the sensitivity of the solution $u(x, y)$ with respect to the variation of the left boundary condition value.

- `"left"`: Identifier for boundary Γ_{left}, and
- `"right"`: Identifier for boundary Γ_{right}.

	Analytical expression	Equivalent pyoti.fem expression (for 2D analysis)
1	$u_{,x} \rightarrow$	`dx( u )`
2	$u_{,y} \rightarrow$	`dy( u )`
3	$\int_\Omega \nabla u \cdot \nabla v \; dV \rightarrow$	`int2d( dx(u)*dx(v) + dy(u)*dy(v) )`
4	$\int_\Omega fv \; dV \rightarrow$	`int2d( f*v )`
5	$\int_{\Gamma_{\text{left}}} gv \; dS \rightarrow$	`int1d('left', g*v )`
6	$u = g \text{ on } \Gamma_{\text{right}} \rightarrow$	`on('right', u, g)`

Table 5.1: Equivalence between analytical variational expressions with its corresponding pyOTI functions.

5.4 Results

This section describes some numerical and analytical implementations with OTIFEM. The numerical examples were run in a machine with an Intel® Core™ i7 4770 at 3.6 GHz, 32 GB ram at 1600 MHz and the operating system was Ubuntu 18.04 (with 2GB of swap memory). The compiler used for C and Cython [67] was the GNU C Compiler GCC 7.5.0 (the version of Cython used was 0.29.20). Python 3.7.4 [68] from the Anaconda distribution 2019.10 [69], and the following libraries were used: SciPy 1.5.2 [70], Sympy 1.6.2 [71] and Numpy 1.19.1 [72].

5.4.1 High-Order Gradients and Shape Derivatives

A case study to validate OTI algebra to compute high order shape derivatives and gradient has been implemented following an example presented in [85]. The differential equation is defined as follows,

$$-\Delta u + u = \sin(x) \quad \text{in } \Omega \tag{5.64}$$

$$u = 0 \quad \text{on } \Gamma_{\text{left}} \tag{5.65}$$

$$u = 0 \quad \text{on } \Gamma_{\text{right}} \tag{5.66}$$

where Ω is the 1D domain $[0, a]$, Γ_{left} is the left boundary ($x = 0$), Γ_{right} is the right boundary ($x = a$), operator Δ is the Laplacian ($\Delta = \nabla \cdot \nabla$) and u is the state function, thus $\Delta u = u_{,x^2}$ for the current problem. The analytical solution of problem is given by the following expression [85]:

$$u(x) = \frac{1}{2}\left(\sin(x) - \frac{\sin(a)}{\sinh(a)}\sinh(x)\right) \tag{5.67}$$

The weak form of the problem is

$$\int_\Omega \nabla u \cdot \nabla v \; dV + \int_\Omega u\, v \; dV = \int_\Omega \sin(x)\, v \; dx \tag{5.68}$$

where v is an arbitrary function that belongs to the same functional space as u.

5.4.1.1 Gradient of $u(x)$

The gradient refers to the derivatives of u with respect to the spatial coordinates that define the domain. In this case, the gradient is defined by the derivative of u with respect to x. In the standard Finite Element method, the spatial derivatives are defined by the derivatives of the basis functions that interpolate the solution.

The traditional finite element approximation of u, u_h, is given by $u_h = \sum_l u_l \psi_l(x)$, where u_l is the value of the function at node l and $\psi_l(x)$ is the basis function of node l. Therefore, the gradient is approximated as

$$u_{,x}(x) \approx u_{h,x}(x) = \sum_{l} u_l \psi_{l,x}(x) \tag{5.69}$$

using the derivatives of the basis function with respect to the spatial coordinate $\psi_{l,x}$. The second order gradient is approximated by

$$u_{,x^2}(x) \approx u_{h,x^2}(x) = \sum_{l} u_l \psi_{l,x^2}(x) \tag{5.70}$$

using the second order derivative of the basis function with respect to the spatial coordinate ψ_{l,x^2}.

Using a finite element space V_h of P1 polynomials defines $\psi_l(x)$ as linear interpolation function. A standard P1 solution of equation (5.64) is shown in Figure 5.9a for $a = 1$. Note however, that since P1 basis functions are linear, the first order derivative with respect to x is constant across every element and second order and higher derivatives are always zero.

Alternatively, OTI numbers can be used to compute the gradient of the finite element solution. The method consist in perturbing all the nodal coordinates inside the domain, excluding those of the boundary, in an imaginary direction formed by one basis, e.g. ϵ_1. Figure 5.8 sketches the perturbation scheme for a 5 element discretization. A sample code is provided in Appendix F.1.

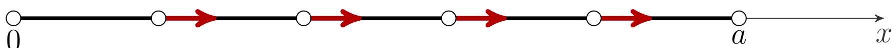


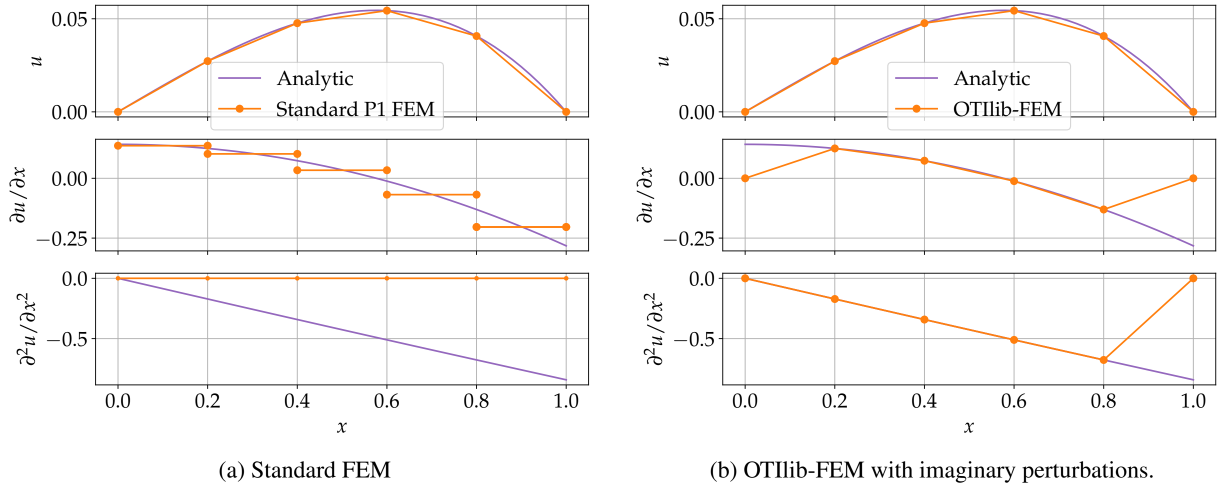
Figure 5.8: Imaginary perturbations applied to every node in the domain in direction ϵ_1 in order to compute the gradient.

The nodal coordinates of the nodes in the domain are therefore perturbed as $x_l^* = x_l + \epsilon_1$. Thus, the result will contain derivatives with respect to the position of those nodes, and hence approximating the derivative of $u(x)$ with respect to x. Results are shown in Figure 5.9b.

It can be observed that despite the problem is analyzed with linear elements, both first and second order derivatives are obtained by performing the corresponding hypercomplex perturbations to the nodal coordinates. The results obtained are only valid within the domain and exclude the values of the boundary.

5.4.1.2 Shape derivatives

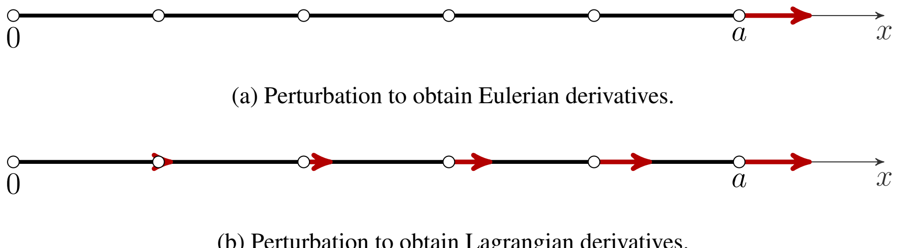
Two types of shape derivatives can be computed [85]: Eulerian and Lagrangian derivatives. Both can be obtained using OTI perturbations. In order to obtain *Eulerian* derivatives, the perturbation is shown



(a) Standard FEM (b) OTIlib-FEM with imaginary perturbations.

Figure 5.9: Gradient derivatives using P1 basis functions. 5.9a shows the approximation of u using the conventional basis functions and 5.9b shows the gradient using OTI perturbations along the nodes that does not belong to the boundary.

in Figure 5.10a. It is accomplished by perturbing the coordinate of the node related to the boundary of interest. In contrast, the computation of a Lagrangian derivative is accomplished by perturbing also the nodal coordinates in the domain as shown in Figure 5.10b. In the Lagrangian derivative case, all nodes are perturbed in ϵ_1, however by a factor proportional to the distance from the left node.



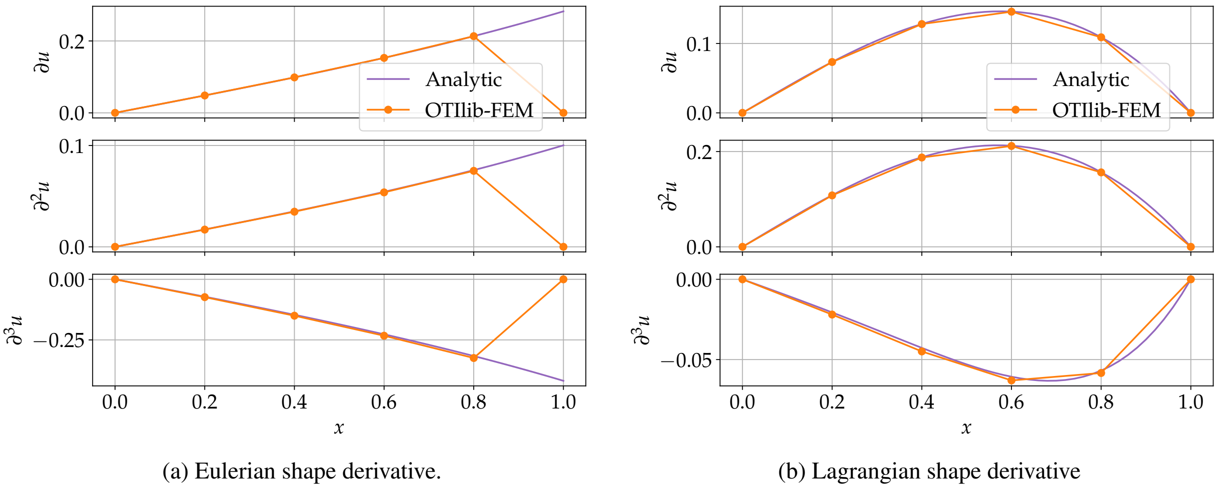
(a) Perturbation to obtain Eulerian derivatives.

(b) Perturbation to obtain Lagrangian derivatives.

Figure 5.10: Imaginary perturbations to obtain shape derivatives.

Figure 5.11 shows the result of perturbing the domain for both Lagrangian and Eulerian derivatives. Figure 5.11a shows the results of perturbing the boundary as in Figure 5.10a, i.e. to obtain the Eulerian shape derivative. Figure 5.11b shows the results of perturbing the boundary as in Figure 5.10b, i.e. to obtain the Lagrangian derivative.

It can be observed that in the Lagrangian shape derivative results, all nodes visually correspond to the analytic result. However, the Eulerian derivative is only obtained in all nodes except the node at the perturbed boundary.



(a) Eulerian shape derivative.

(b) Lagrangian shape derivative

Figure 5.11: Shape derivatives using P1 basis functions and OTIFEM.

Additionally, if the perturbation of the domain to obtain the Lagrangian derivative is applied to only to certain nodes and all other nodes remain unperturbed; the result is that both Eulerian and Lagrangian derivatives can be reconstructed. An example of this perturbation is shown in Figure 5.12 where every other node is perturbed. The corresponding result of the analysis is shown in Figure 5.13 for a domain with 33 nodes. The Python code used to simulate this is provided in Appendix F.1.

Figure 5.12: Selective perturbations of the domain.

It can be observed that every node perturbed correspond to the Lagrangian derivative but the solution at the nodes that were not perturbed correspond to the Eulerian derivative.

5.4.2 Heat Transfer: Hollow Cylinder with Internal Heat Generation and Convective Surfaces.

A two dimensional axisymmetric heat transfer analysis was conducted for a hollow cylinder as presented in section 3.7.1, with the same real input values. In this numerical implementation, however, OTI numbers and pyOTI library were used to perform an OTIFEM analysis to compute high order derivatives of the normalized temperature θ with respect to some input variables of the problem. Computation of derivatives of up to 30th order with respect to 3 variables were carried out in a single analysis, for a total of 5455 derivatives plus the real result. The derivatives were computed with respect to the normalized heat generation term $\hat{s}$

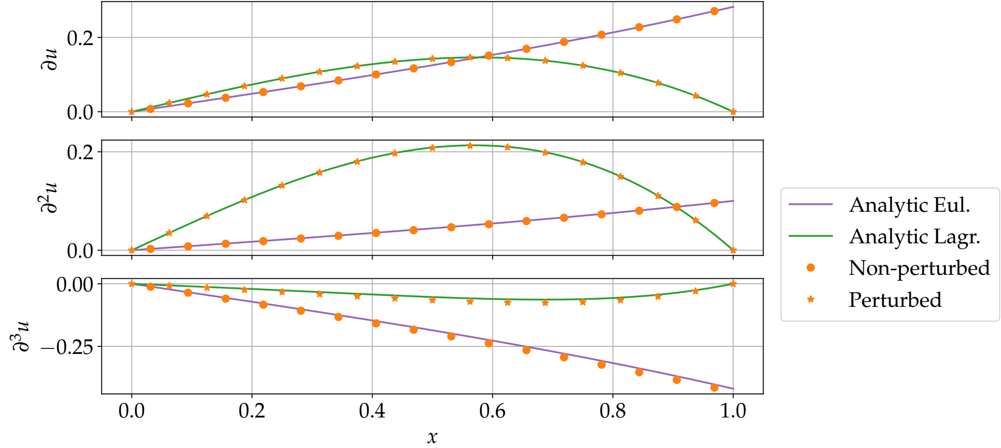


Figure 5.13: Result of the selective perturbation to obtain both Eulerian and Lagrangian shape derivatives in a domain with 33 nodes.

and the Biot numbers at the inner and outer surfaces, Bi_i and Bi_o respectively. The perturbations were as follows

$$\hat{s}^* = \hat{s} + \epsilon_1, \qquad \hat{s}^* \in \mathbb{OTI}_3^{30} \tag{5.71}$$

$$Bi_i^* = Bi_i + \epsilon_2, \qquad Bi_i^* \in \mathbb{OTI}_3^{30} \tag{5.72}$$

$$Bi_o^* = Bi_o + \epsilon_3, \qquad Bi_o^* \in \mathbb{OTI}_3^{30} \tag{5.73}$$

The finite element analysis was performed over a mesh with a total of 8005 nodes and 2000 8-node serendipity quadrilaterals, with 1000 elements distributed along the radial direction and 2 elements in the z direction. The error of the finite element solution is compared against the analytical solution of the problem, shown in equation (3.67) at the normalized radial coordinate $R = 1$, representing the internal surface at r_i. The relative error is computed as the error between the p'th order derivative obtained with OTIFEM analysis with respect to the p'th order analytical derivative, as follows

$$\text{Relative error} = \frac{|f_{\text{a}} - f_{\text{FEM}}|}{|f_{\text{a}}|} \tag{5.74}$$

Figure 5.14 shows the relative errors obtained after performing the perturbations shown in equations (5.71-5.73). From the analytical solution, equation (3.67), θ is linear with respect to the normalized heat generation term $\hat{s}$, thus it is expected that the second and higher order derivatives of θ with respect to $\hat{s}$

are zero. It is observed that the OTI-FEM analysis correctly addressed this, generating exact results. The derivatives with respect to the Biot numbers, however, start with the similar accuracy compared to the traditional FEM solution, depicted by the result at $p = 0$. However, the error increases with respect to the order of derivative computed in the analysis. The error, however, does not exceed 10^{-8} for any order of derivative, sufficiently accurate for most applications.

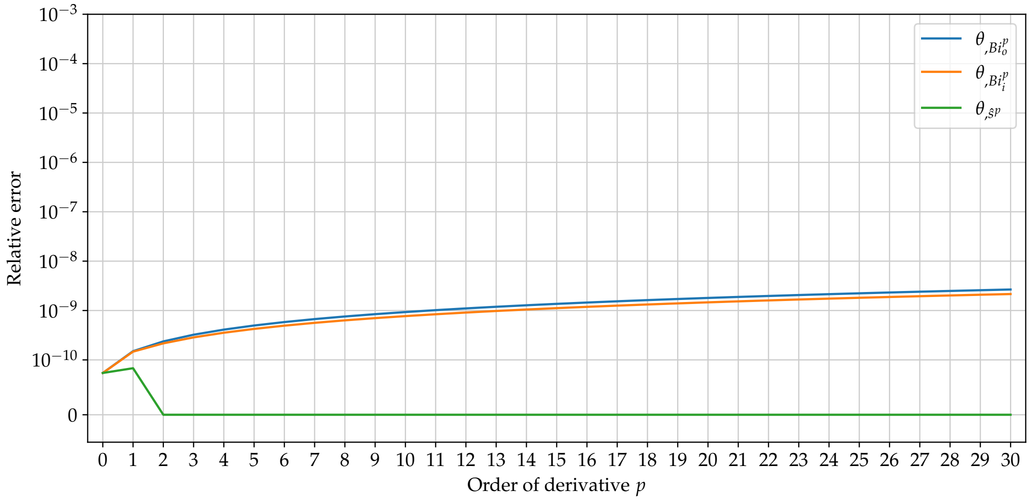


Figure 5.14: Relative error of the p'th order derivatives of the non-dimensional temperature θ with respect to the source $\hat{s}$.

5.4.3 Linear Elasticity: Thick Walled Cylinder Subject to Uniform Pressure.

A two dimensional linear elastic analysis was conducted for a thick-walled cylinder with uniform pressures at both the internal and external faces. The chosen domain (Ω) for the computational analysis was a quarter of the cross section of the cylinder, as shown in Figure 5.15. The domain Ω is delimited by the inner and outer radii, r_i and r_o, respectively. Symmetry boundary conditions are applied at both faces at the symmetry axis, Γ_x and Γ_y. The cylinder is subject to uniform inner and outer pressures P_i and P_o, at the inner and outer boundaries Γ_i and Γ_o, respectively.

The analytic solution of this problem is given in [86]. The displacements are defined as:

$$u_r(r) = \frac{r\,(1+\nu)}{E}\left((1-2\nu)B - \frac{A}{r^2}\right) \tag{5.75}$$

$$u_\theta(r) = 0 \tag{5.76}$$

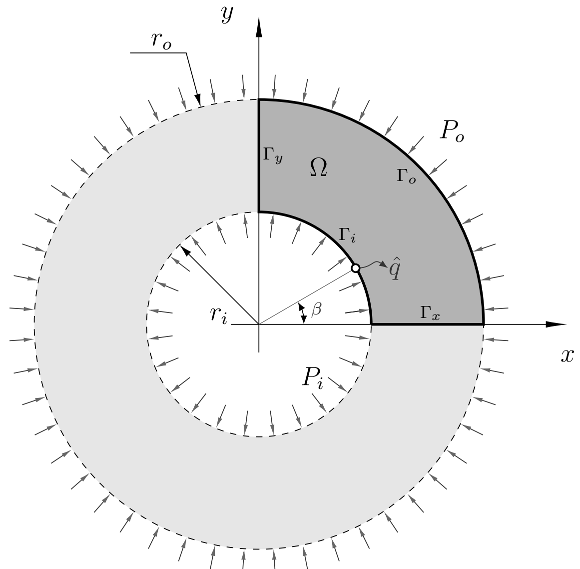


Figure 5.15: Sketch of the thick walled cylinder with internal and external uniform pressures and its computational analysis domain Ω.

where u_R and u_θ are the radial and tangential displacements respectively, E is the Young modulus and ν is the Poisson's ratio of the cylinder wall material. Finally, A and B are constants defined as follows

$$A = \frac{r_i^2 r_o^2 (P_o - P_i)}{r_o^2 - r_i^2} \tag{5.77}$$

$$B = \frac{r_i^2 P_i - r_o^2 P_o}{r_o^2 - r_i^2} \tag{5.78}$$

The analysis were performed using the input data shown in Table 5.2. The evaluation point $\hat{q}$ corresponds to $\beta = 30^o$ and $|\hat{q}| = r_i$. All analyses were performed with 6-node triangular elements.

Item	Value	Unit
E	2100000	Pa
ν	0.28	
r_i	1	m
r_o	2	m
P_i	10	Pa
P_i	1000	Pa

Table 5.2: Input variable values used in the simulation of the thick walled cylinder analysis.

5.4.3.1 Accuracy of high order derivatives.

Computation of up to 30'th order derivatives with respect to different input variables was carried out using OTI perturbations. The mesh used for this analysis had 19800 6-noded triangles with a total of 40000 nodes. Table 5.3 summarizes the characteristics of the Finite Element problem.

Item	Value
Elements in the mesh	19 800
Nodes in the mesh	40 000
Degrees of freedom of the problem	80 000
Non-zero coefficients in $\mathbf{K}^*$	1 827 974

Table 5.3: Properties of the system of equations for an OTIFEM analysis with 80k degree of freedom for a 2D linear elastic problem, using pyOTI.

The computation of derivatives with respect to Poisson ratio ν, outer and inner pressures, P_i and P_o respectively, was carried out in a single analysis with the following perturbations

$$\nu^* = \nu + \epsilon_1, \qquad \nu^* \in \mathbb{OTI}_3^{30} \tag{5.79}$$

$$P_i^* = P_i + \epsilon_2, \qquad P_i^* \in \mathbb{OTI}_3^{30} \tag{5.80}$$

$$P_o^* = P_o + \epsilon_3, \qquad P_o^* \in \mathbb{OTI}_3^{30} \tag{5.81}$$

On the other hand, the derivatives with respect to the Young modulus E and the outer radius r_o were computed in a single analysis. The perturbations applied to E and to the i'th nodal coordinate were

$$\mathbf{x}_i^* = \mathbf{x}_i + \frac{\mathbf{x}_i}{r}\frac{(r - r_1)}{(r_2 - r_1)}\epsilon_1, \; \mathbf{x}^* \in \mathbb{OTI}_2^{30\,(2)} \tag{5.82}$$

$$E^* = E + \epsilon_2, \qquad E^* \in \mathbb{OTI}_2^{30} \tag{5.83}$$

where $r = |\mathbf{x}_i|$, and the nodal perturbation was applied to every node in the mesh. The perturbation to the nodal coordinates correspond to a Lagrangian shape derivative.

The relative error of the p'th order derivative with respect to each variable evaluated at point $\hat{q}$ is compared with the derivative of the analytical solution, see equation (5.75). Results are shown in Figure 5.16. It can be clearly seen from equation (5.75) that the analytical solution is linear with respect to both the outer

and inner pressure, thus only first order derivatives exist. The finite element solution of the derivatives is exact to this result. Also, $\mathbf{u}$ is quadratic with respect to ν, thus only second order derivatives are non-zero. This is however not represented correctly by the finite element solution as the third and higher order derivatives with respect to ν are non zero. Figure 5.16 shows the absolute error of the derivatives with respect to ν with a dashed line. It can be observed that the error is accumulated by the increase in order of derivative. this performs similarly to the example shown in section 4.4.5.

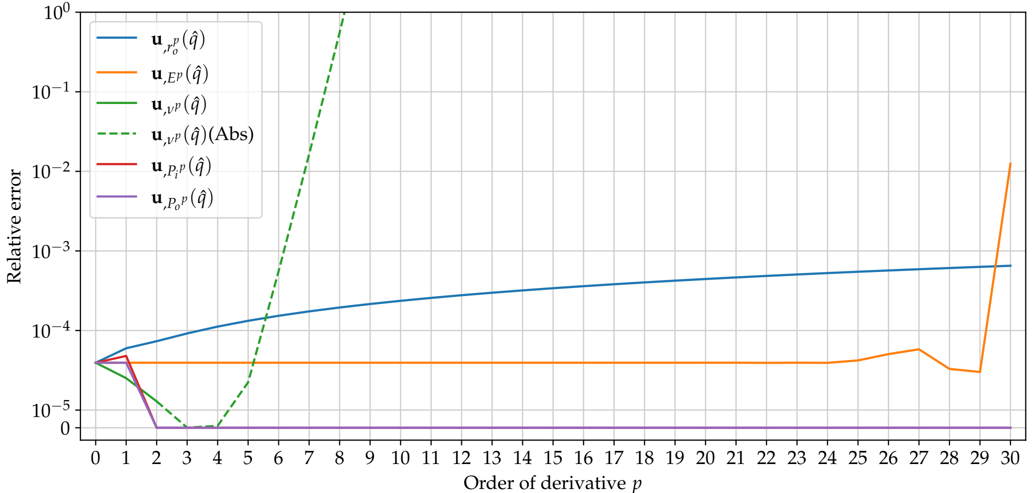
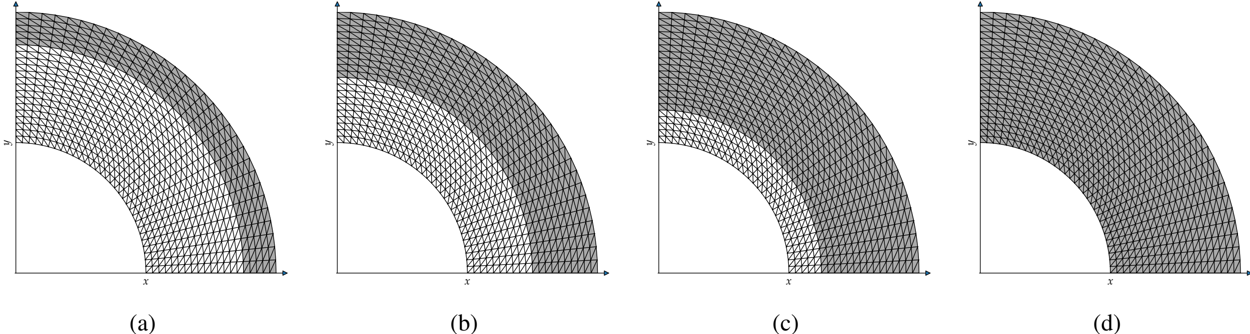


Figure 5.16: Relative error of the p'th order derivative of the displacement field with respect each of the following input variables: Poisson ratio ν, Young modulus E, outer P_o and inner P_i pressures and outer radius r_o. The absolute error of the derivative of the displacement field $\mathbf{u}$ with respect to ν is shown with a dashed line as the corresponding analytical derivative is zero.

5.4.3.2 Analysis of the nodal coordinate perturbation region.

Computation of sensitivities with respect to shape parameters requires the perturbation of a region of the mesh, as described in section 3.5. Choosing the perturbation region is a task left usually to the user. In this section the effect of the perturbation region over the accuracy of the shape derivatives will be investigated by computing derivatives with respect to the outer radius r_o for four perturbation regions, namely 25, 50, 75 and 100 per cent of the area of the domain, as indicated in Figure 5.17.

The perturbation to the nodal coordinates is carried out in the radial direction using a linearly decreasing function from outer boundary (with a value of 1) decreasing to some region in the mesh. The perturbation of the i'th node in the mesh is performed as follows



(a) (b) (c) (d)

Figure 5.17: Perturbation regions for the evaluation of the high order shape derivative behavior for a total of (a) 25%, (b) 50%, (c) 75% and (d) 100% of the elements in the mesh.

$$\mathbf{x}_i^* = \mathbf{x}_i + \frac{\mathbf{x}_i}{r} f(r)\, \epsilon_1,\ \mathbf{x}^* \in \mathbb{OTI}_1^{30\ (2)} \tag{5.84}$$

where $r = |\mathbf{x}_i|$ and $f(r)$ is a function defined as follows

$$f(r) = \begin{cases} 0 & \text{if } r < pc\, r_1 + (1-pc)\, r_2 \\ \frac{r - pc\, r_1 - (1-pc) r_2}{pc(r_2 - r_1)} & \text{otherwise} \end{cases} \tag{5.85}$$

where pc is the proportion of the area perturbed and the total area of the mesh, i.e. $pc = A_{\text{perturbed}}/A_{\text{total}}$. For this implementation, the values of pc were 0.25, 0.50, 0.75 and 1.00.

The properties of the mesh used in this analysis are shown in Table 5.3, and the analysis was solved in pyOTI using the sparse implementation. The relative error of the p'th order shape derivative of the nodal displacements $\mathbf{u}$ with respect to the outer radius r_o evaluated at $\hat{q}$ is shown in Figure 5.18. It can be observed that the best accuracy and stability of the derivatives is obtained with the perturbation of 100% of the elements. If a threshold is set to a relative error of 10^{-3}, then the analysis with 25% of elements perturbed exceeds the threshold first after the 9th order of derivatives, the analysis with 50% up to 16'th order derivatives and the analysis with 75% of the elements perturbed up to 27th order.

5.4.3.3 Accuracy of OTIROMs for shape and material perturbations.

A reduced order model was generated using the results from an OTIFEM analysis perturbing the Young modulus E and the outer radius r_o as in equations (5.82-5.83) on a mesh with properties described in Table 5.3. The OTIROM from this result is able to capture variations from shape and material inputs. The results

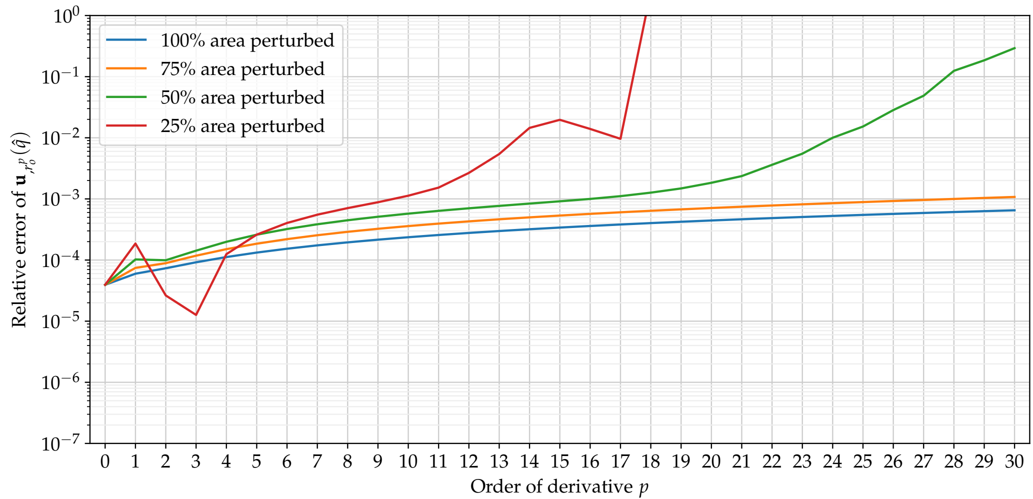


Figure 5.18: Relative error of the p'th order shape derivative of the displacement field u with respect to the outer radius r_o evaluated at the point $\hat{q}$ for different number of perturbed elements in the mesh.

from the OTIFEM analysis results in a solution with a total of $N = 496$ directions. The goal of this section is to evaluate the behavior of the variations to the outer radius and Young modulus, Δr_o and ΔE respectively, by evaluating the relative error of the OTIROM predicted value against the analytical solution for the corresponding evaluation points.

The approximated value of the OTIROM for the finite element solution is computed as follows

$$\mathbf{u}|_{r_o+\Delta r_o, E+\Delta E} \approx \text{OTIROM}(\mathbf{u}^*, \{\Delta r_o, \Delta E\}) \tag{5.86}$$

Figure 5.19 shows the contours delimiting the region where the relative error of the p'th order OTIROM is 10^{-4}. It can be observed that the evaluations performed generate rectangular regions, with an increased dimension for the material property values. The mixed derivatives computed in the OTIFEM analysis are useful to the reduced order model since the accuracy of these derivatives is crucial to the diagonal extension of the confidence region. It is the opposite to the example shown in section 4.4.5, where the inaccuracy of the mixed derivatives led to a decrease in the diagonal dimension of the confidence region.

For illustration purposes, Figure 5.20 shows the transformation of the nodal displacement magnitude with the variation of the outer radius and Young modulus as $r'_o = r_o + 40\%\ r_o$ and $E' = E + 70\%\ E$. It can be observed that although the transformation is significant, the OTIROM model accurately represents the solution for the given input values for every evaluation point in the domain.

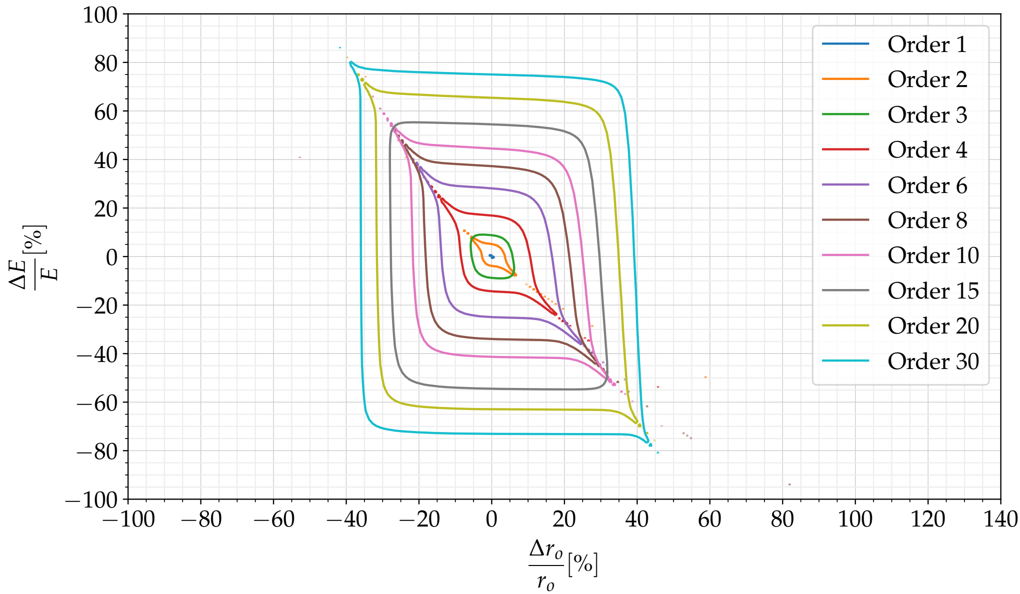


Figure 5.19: Contour lines delimiting the regions where the relative error of the p'th order OTIROM of the nodal displacements $\mathbf{u}(\hat{q})$ is below 10^{-4} with respect to the analytic solution evaluated using $r_o' = r_o + \Delta r_o$ and $E' = E + \Delta E$.

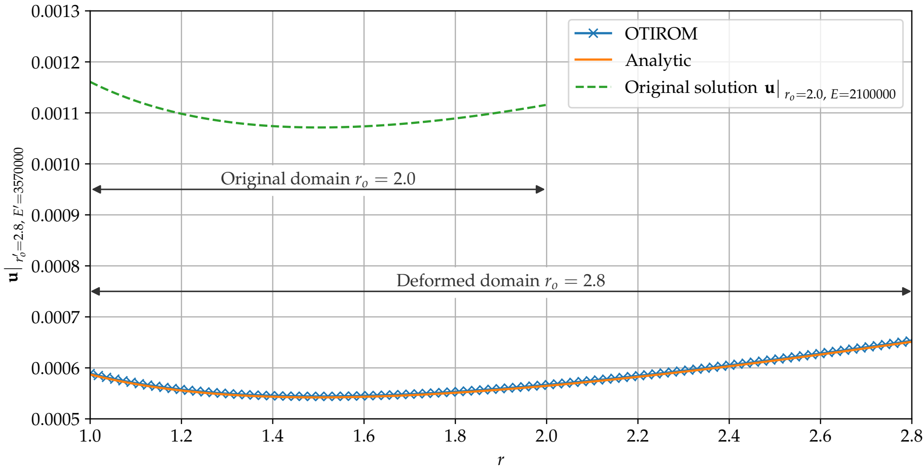


Figure 5.20: Result of the 30'th order OTIROM of the nodal displacement magnitude $\mathbf{u}$ for the new input values $r_o' = r_o + 40\%\ r_o$ and $E' = E + 70\%\ E$.

5.4.3.4 Computational performance.

The estimation of the implementation overhead with respect to the traditional real finite element method is addressed first to evaluate the additional time given by the operator overloads and function calls that a new algebras minimally require in comparison to the traditional methods. This is addressed by comparing a real FEM implementation using C double precision operations against equivalent-to-real OTI numbers using both the static-dense and sparse modules from pyOTI.

5.4.3.4.1 Library overhead. Three evaluations were performed on a mesh with approximately 250k 6-noded triangles, 500k nodes and 1M degrees of freedom for the real FEM (available in pyOTI) and real-equivalent static dense ($\mathbb{OTI}_0^0$) and sparse (no imaginary perturbations) implementations. The exact characteristics of the system of equations is summarized in Table 5.4.

Item	Value
Elements in the mesh	249 852
Nodes in the mesh	501 156
Degrees of freedom of the problem	1 002 312
Non-zero coefficients in $\mathbf{K}^*$	23 009 590

Table 5.4: Properties of the system of equations FEM analysis solving a 1M degree of freedom 2D linear elastic analysis, using the real module in pyOTI.

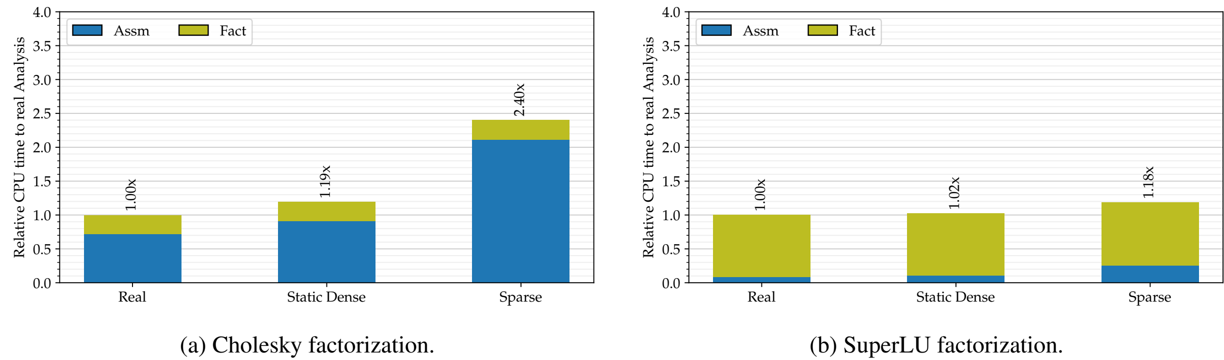
Two solution algorithms were used to factor and solve the system of equations from each problem: LU decomposition through SciPy's distribution SuperLU [76] and Cholesky factorization from Scikit-Sparse [78] package. CPU-times measured were the assembly time of the global system matrix and force vector ($\mathbf{K}^*$ and $\mathbf{f}^*$), and time to factor the matrix and solve the real system of equations for each case. The CPU-time measurements were averaged over 3 evaluations of the complete finite element solution run for every analysis. All results were normalized to the real FEM computation time for the respective solver algorithm. For reference purposes, the real FEM evaluation times are shown in Table 5.5.

Solution method	Assembly	Solution	Total time
Cholesky	23.42 s	9.07 s	32.49 s
SuperLU	23.42 s	250.60 s	274.02 s

Table 5.5: CPU time in seconds of the real FEM analysis solving a 1M degree of freedom 2D linear elastic analysis, using the real module in pyOTI.

The SuperLU solver took 27× the Cholesky solver and represents a 91% of the total solution time of the

problem. Cholesky solution, on the other hand, represents the 28% of the total solution time. This difference is significant when normalizing the total CPU time with respect to the real analysis, as the factorization algorithm is common to all evaluations. For this problem Cholesky is more efficient, but limited to positive-definite symmetric matrices. Therefore, both methods are reported for generalization purposes. The CPU time used by real-equivalent implementations with static-dense and sparse modules from pyOTI are shown in Figure 5.21.



(a) Cholesky factorization.

(b) SuperLU factorization.

Figure 5.21: Normalized CPU time of the static dense and sparse implementations of real-equivalent numbers in pyOTI for a 2D linear elastic finite element analysis of a mesh with 1M degrees of freedom using (a) Cholesky and (b) SuperLU factorization algorithms.

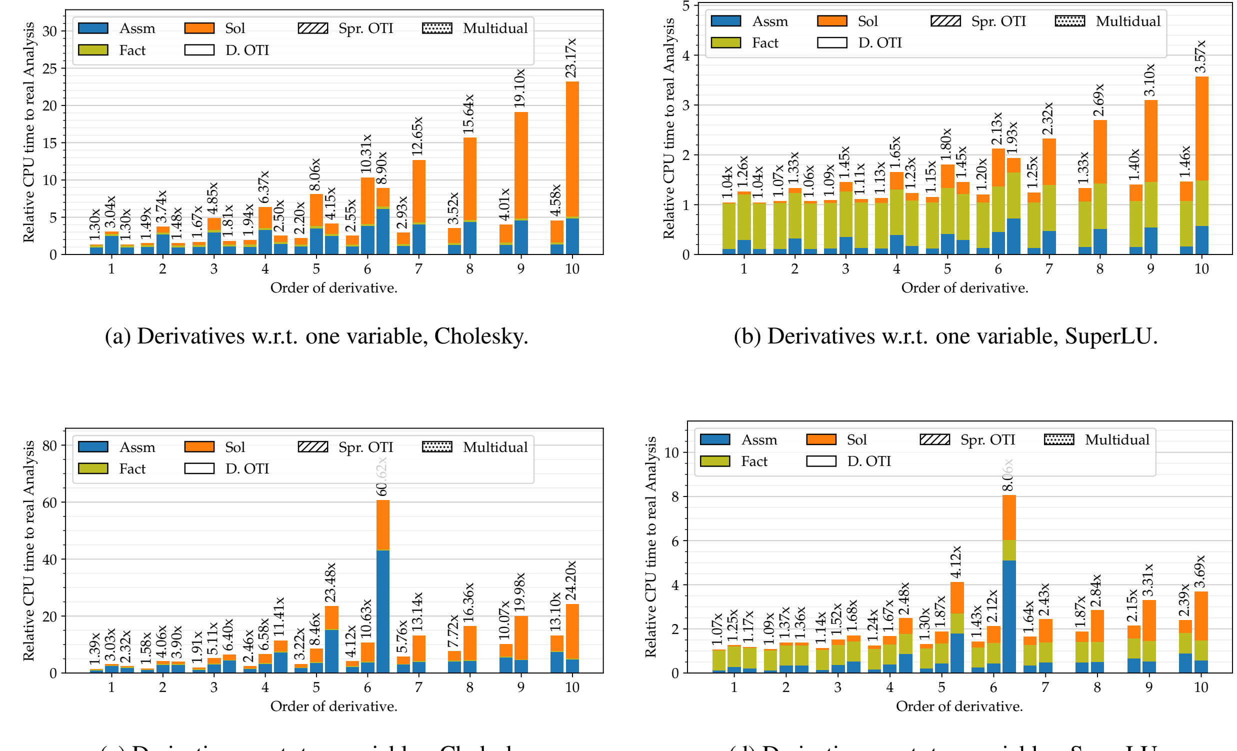
The computational overhead of the static dense implementation was 19% when compared with the Cholesky implementation, and 2% when compared with the SuperLU implementation. The sparse implementation was slower, showing 140% and 18% overhead compared to the real FEM for Cholesky and SuperLU solutions.

5.4.3.4.2 Static dense implementation. The behavior of the complete hypercomplexification of the finite element operations with dense algebra was evaluated to compare OTI and multidual implementations in a finite element use case. Complete hypercomplexification is understood as uplifting every input variable and intermediate operation to the selected algebra, even if that input variable or intermediate operation is not perturbed in a specific analysis. The analyses were performed in a system with the properties listed in Table 5.4.

Both OTI and multidual implementations used the developed solution-by-blocks method, and the repetitive evaluation scheme was used for computation of multivariable derivatives with multidual numbers (see Appendix A). Since the block solver always use the same factored $\mathbf{K}_r$, the repetitive multidual implementa-

tion used a single factorization of the matrix for all re-evaluations required. Evaluations with both the static dense and sparse implementations are shown, and the sparse implementation considered only perturbations to material parameters.

The CPU time relative to the real FEM analysis is presented in Figure 5.22 for single and bi-variate derivatives using Cholesky and SuperLU factorization for up to 10th order derivatives. The CPU times are subdivided by the assembly time, the time to factor $\mathbf{K}_r$ and solve $\mathbf{u}_r$ and the time to solve for the higher order imaginary directions for all cases. All implementations used the corresponding static dense implementation of the selected algebra. For instance, biduals were used for computation of second order derivatives by using module **`pyoti.static.mdnum2`**, triduals for third order derivatives with **`pyoti.static.mdnum3`** and so on. Results from Multiduals with 7 or more bases required more memory than what was physically available to the machine, as the total number of coefficients per element in every operation was $2^7 = 128$ or larger. The system matrix K*, stored in compressed stored row format, required an estimated 22.2 GB of memory and due to the other computations needed, the total 34 GB (32GB physical and 2GB swap) of memory was not enough and the simulations were unsuccessful. OTI with only one basis were used and the truncation order matched the required derivatives. All 10th order derivatives could be evaluated using OTI numbers for both single (module **`pyoti.static.onumm1n10`**) and bivariate static dense implementations (module **`pyoti.static.onumm2n10`**) as well as the sparse implementation (module **`pyoti.sparse`**). The maximum memory consumed was shown by the static dense implementation for the two variable analysis ($n = 10$, $m = 2$) with 66 imaginary coefficients, requiring a system matrix of 11.5 GB.



(a) Derivatives w.r.t. one variable, Cholesky.

(b) Derivatives w.r.t. one variable, SuperLU.

(c) Derivatives w.r.t. two variables, Cholesky.

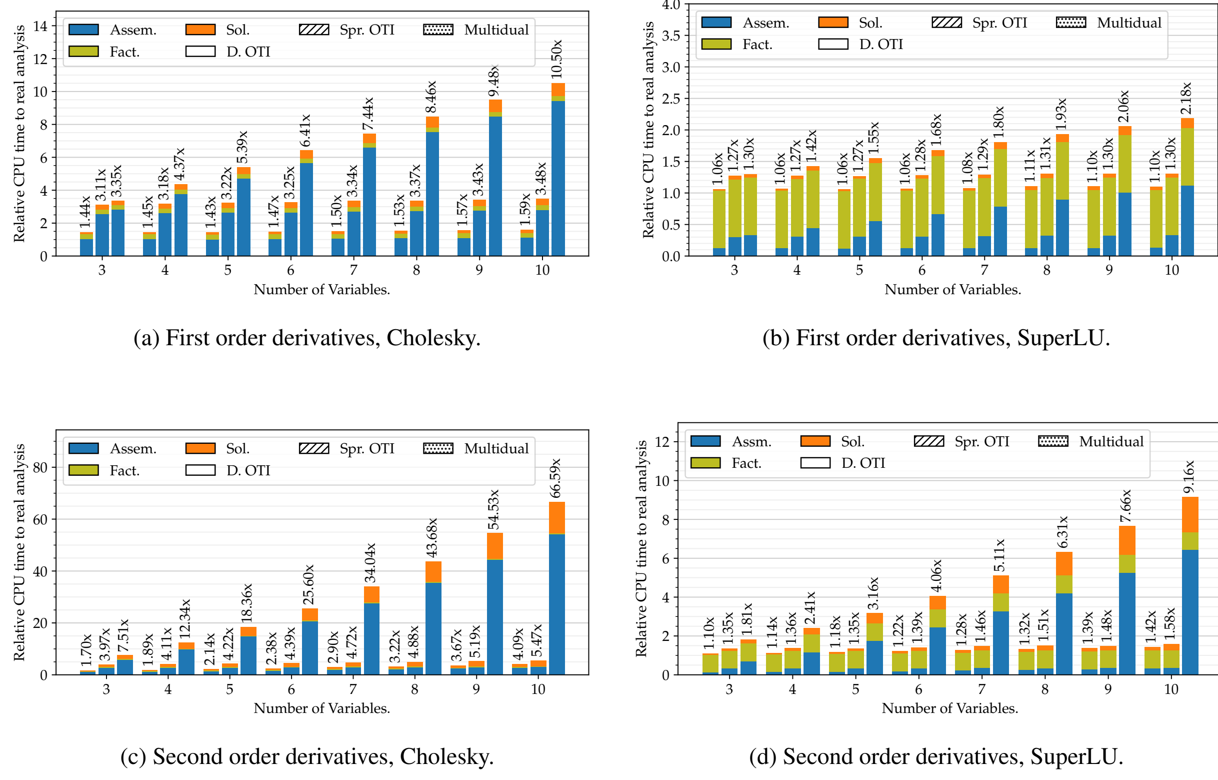
(d) Derivatives w.r.t. two variables, SuperLU.

Figure 5.22: Normalized CPU time of the static dense OTI, static dense multidual and sparse OTI implementations using pyOTI for a 2D linear elastic finite element analysis on a mesh with 1M degrees of freedom for computation of single variable high order derivatives using (a) Cholesky and (b) SuperLU factorization algorithms; and for computation of high order derivatives of two variables using (c) Cholesky and (d) SuperLU solvers. Multidual evaluations are implemented using the repetitive scheme.

Computation of first order derivatives of one variable reduces to dual numbers ($\mathbb{OTI}_1^1 = \mathbb{D}_1 = \mathbb{D}$), which is equivalent for OTIs and multiduals. Results show that it takes 4-30% extra time to compute a single first order derivative using pyOTI static dense implementation. Computation of 6th order derivatives of one variable requires 790% more time when using hexaduals, but 43% when using OTIs with truncation order 6 $\mathbb{OTI}_1^6$, in the case for a Cholesky implementation. With the SuperLU factorization algorithm, computing 6th order derivatives for two variables (instead of one), increased the CPU time by 19% when OTIs, in contrast to 317% with multidual, which required 7 hexadual evaluations. It can be observed that in general, multidual CPU time increase exponentially with respect to the order of derivative required. However, OTI increase in CPU time is considerably slower. It is also observed that the sparse implementation has an overhead with respect to the static dense library, which grows with respect to the order and is more significant when computing single variable derivatives. Also, it is observed that the overhead does not increase as significantly

when more variables are considered (variation is up to 3% when computing 6th order derivatives), in contrast to the static dense OTI and multidual implementations.

Hypercomplexification to enable computation of first and second order derivatives from 3 up to 10 variables was also compared implemented and compared. The normalized CPU times are shown in Figure 5.23 using both static dense OTI and multidual implementations. Results show that the CPU time grows more with multidual implementation than with OTIs when increasing the number of variables. The difference in CPU time for an OTI analysis for 3 to 10 variable first order derivatives is 15% for Cholesky and 4% for SuperLU. In contrast, the difference in CPU time for an OTI analysis for 3 to 10 variable second order derivatives is 239% for Cholesky and 32% for SuperLU implementations.



(a) First order derivatives, Cholesky.

(b) First order derivatives, SuperLU.

(c) Second order derivatives, Cholesky.

(d) Second order derivatives, SuperLU.

Figure 5.23: Normalized CPU time of static dense OTI, multidual and sparse OTI implementations using pyOTI for a 2D linear elastic finite element analysis on a mesh with 1M degrees of freedom for computation of multivariable derivatives of first order using (a) Cholesky and (b) SuperLU factorization algorithms; and for computation of multivariable second order derivatives using (c) Cholesky and (d) SuperLU solvers. Multidual evaluations are implemented using the repetitive scheme.

5.4.3.4.3 Sparse implementation Sparse OTI numbers allows computation derivatives with respect to any number of variables as the size of the algebra is adjusted automatically according to the basis and

truncation order needed. Thus, a single implementation is needed for any derivative required, in contrast to the dense implementation which needs a specific module for every combination of number of basis and truncation order. The performance of the OTI sparse implementation was evaluated on a mesh with 19800 6-noded triangles as indicated in Table 5.3. The measurements were performed using a Cholesky factorization.

In a first analysis, nodal coordinates were perturbed in two different manners. A first perturbation correspond to treat every nodal a value as an independent input. Thus, every component of every node was treated as an independent variable and perturbed in two independent imaginary directions, i.e. $x_i^* = x_i + \epsilon_{2i}$, $y_i^* = y_i + \epsilon_{2i+1}$. This may correspond, for example, to an analysis where every boundary node moves in an independent direction. A second perturbation was implemented, where every nodal coordinate in the mesh was perturbed in m imaginary directions, i.e. $x_i^* = x_i + \sum_{k=0}^{m} h_k \epsilon_k$. This perturbation corresponds, for example, as perturbing the mesh to compute Lagrangian shape derivatives with respect to variation of m boundary parameters, independently. The relative CPU time to a real FEM implementation is presented in Figure 5.24 for computation of up to 800 derivatives in a single analysis.

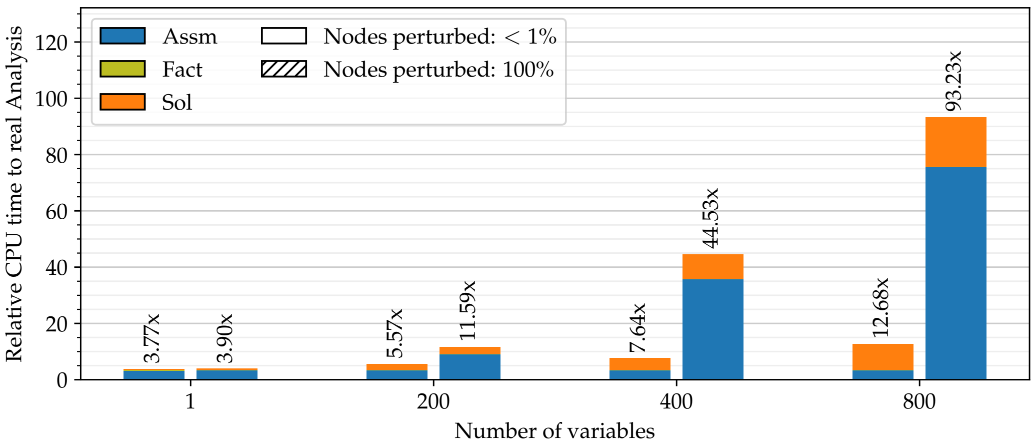


Figure 5.24: Normalized CPU time of sparse OTI numbers in pyOTI a 2D linear elastic finite element analysis in a problem with 80k degrees of freedom for computation of first order derivatives of up to 800 variables, when perturbing less than 1% of the total number of nodes and the total number of nodes in the mesh.

The difference in the CPU-time spent to compute a first order derivative of a nodal coordinate compared to a the total boundary perturbation is negligible, as the boundary perturbation was 3% slower than the perturbation of a single variable. Perturbation of the whole domain spent 7.35× more CPU time than the perturbation to 1% of the nodes.

Computation of higher order multivariable derivatives were used in a different analysis to the same mesh, to produce a 30th order OTIROM for shape derivatives with respect to the outer radius r_o (100% of the domain was perturbed) and the Young modulus E, see section 5.4.3.3. Figure 5.25 shows the normalized CPU times for the OTIFEM evaluations computing derivatives with respect to E only (30 derivatives), computing shape derivatives (30 derivatives) and perturbing both variables (495 derivatives).

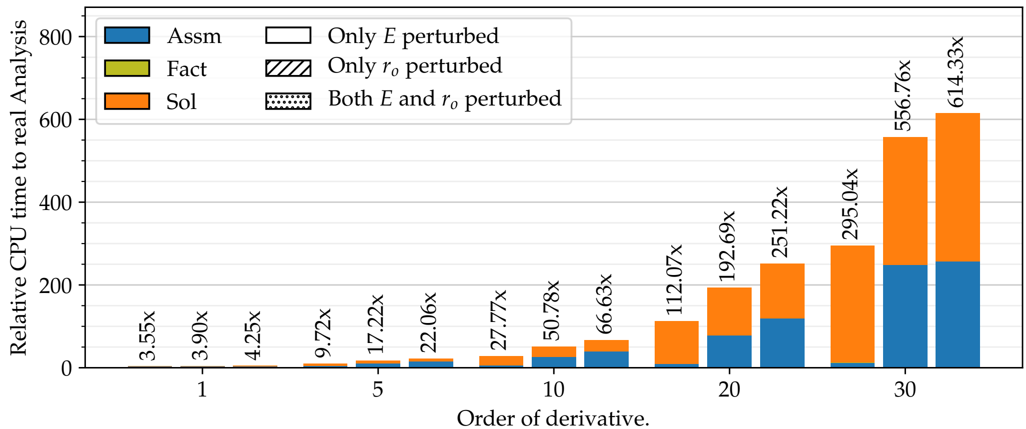


Figure 5.25: Normalized CPU time of sparse OTI numbers used for 2D linear elastic finite element analysis in a problem with 80k degrees of freedom for computation of up to 30th order derivatives with respect to the Young modulus E and the domain outer radius r_o, when derivatives were computed independently or in a single analysis.

The effect of using the sparse OTI implementation can be analyzed from Figure 5.25. Computation of 30th order derivatives with respect to the Young modulus E takes 53% of the time to compute 30th order derivatives with respect to the domain boundary. It however only takes 10% more time to solve the additional 465 derivatives when solving all 2 variable 30th order derivatives. The total CPU time of evaluating derivatives of the modulus of elasticity is heavily driven by the time to the OTI linear system of equations. However, for the other two simulations containing mesh perturbation, the CPU time of assembly and solution time contributed almost equally to the total simulation time.

The normalized CPU time for evaluating an OTIROM is shown in Figure 5.26 for different truncation orders. Results are normalized with respect to the total CPU time of a real FEM analysis. The evaluation time increases with respect to the truncation order of the OTIROM evaluated. For 80k degrees of freedom, evaluation of 30'th order OTIROM required 39% the time of a real analysis.

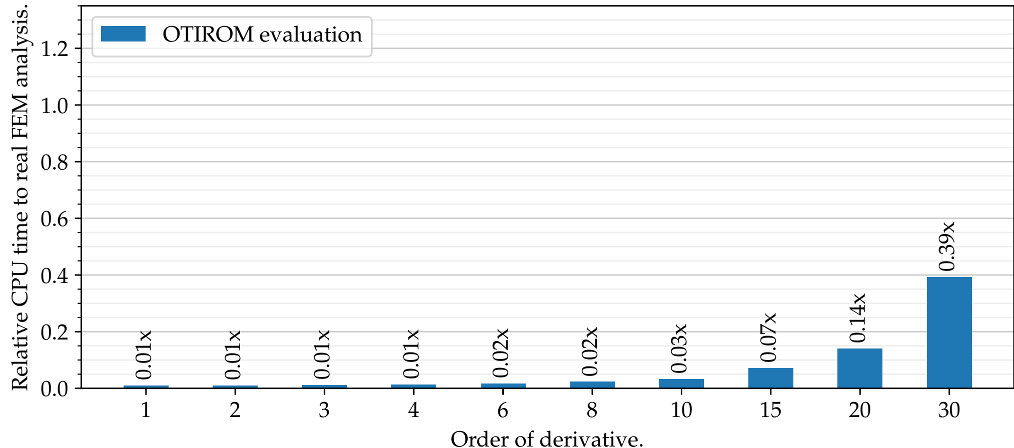


Figure 5.26: Normalized CPU time of an OTIROM generated from a 2D linear elastic OTIFEM in a problem with 80k degrees of freedom for up to 30th order reduced order models with respect to the Young modulus E and the domain outer radius r_o. The OTIROM evaluation time correspond to the evaluation of all 80k degrees of freedom. Results are normalized with respect to the total CPU time of a real FEM analysis with 80k degrees of freedom.

5.4.4 System of Stokes: Lid-driven cavity.

The lid-driven cavity is a well-known benchmark problem for fluid flow analysis [87]. In this numerical example, OTI numbers were used to generate a reduced order model of the velocity $\mathbf{u}$ and pressure p fields using OTIFEM method, see section 4.2.5. The problem consist of simulating the fluid flow in a square cavity with tangential velocity at the top boundary Γ_{top} and non-slip boundary conditions elsewhere, as shown in Figure 5.27. The evaluation point for error comparison is $\hat{q} = \{0.4L_x, 0.9L_y\}$.

Stokes equation is used to simulate the fluid flow neglecting effects of inertial forces, see appendix D.4. The simulation was run using a structured triangulation of the domain with a total of 20 000 6-noded triangles, see Table 5.6. The finite element spaces used for this analysis depend on the state function, for the case of the velocity field $\mathbf{u}$ second order 6-node triangle interpolation was used and for the pressure field p first order 3-node triangle was used (P2-P1 interpolation).

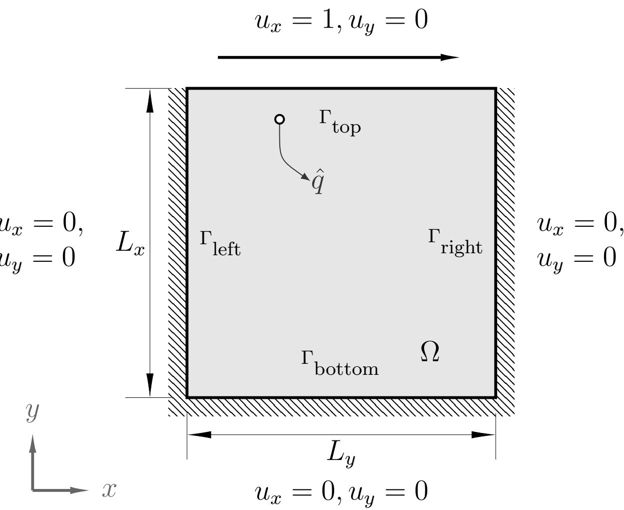


Figure 5.27: Domain of the cavity analysis.

Item	Value
Elements in the mesh	20 000
Nodes in the mesh	40 401
Degrees of freedom of the problem	91 003
Non-zero coefficients in $\mathbf{K}^*$	1 757 807

Table 5.6: Properties of the system of equations for an OTIFEM analysis with 91k degrees of freedom for a 2D lid driven cavity problem, using pyOTI.

OTIFEM was used to generate a 30'th order OTIROM for shape transformations of both L_x and L_y domain dimensions, independently, and took $583.25\times$ the solution time of a real equivalent FEM evaluation. The nodal coordinates were perturbed in two independent imaginary directions as shown in Figure 5.28. The perturbations were performed as to compute the Lagrangian shape derivatives, applied for all nodes in the mesh,

$$x_i^* = x_i + \frac{x_i}{L_x}\,\epsilon_1,\ x^* \in \mathbb{OTI}_2^{30} \tag{5.87}$$

$$y_i^* = y_i + \frac{y_i}{L_y}\,\epsilon_2,\ y^* \in \mathbb{OTI}_2^{30} \tag{5.88}$$

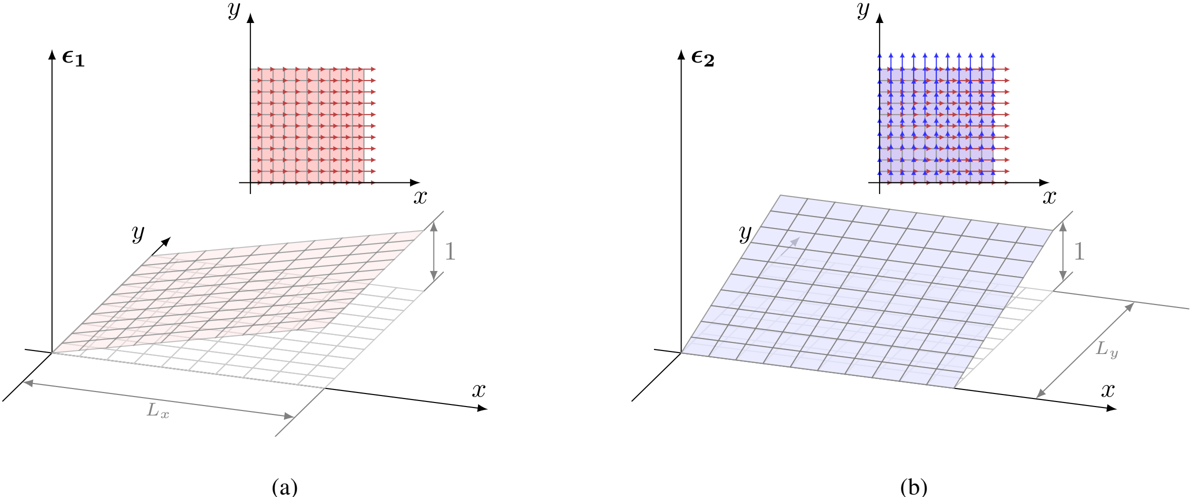


Figure 5.28: Perturbation applied to compute shape derivatives with respect to the domain dimensions L_x and L_y. Note that this image shows quadrilateral elements for simplification purposes only and the actual OTIFEM analysis was performed using triangular elements.

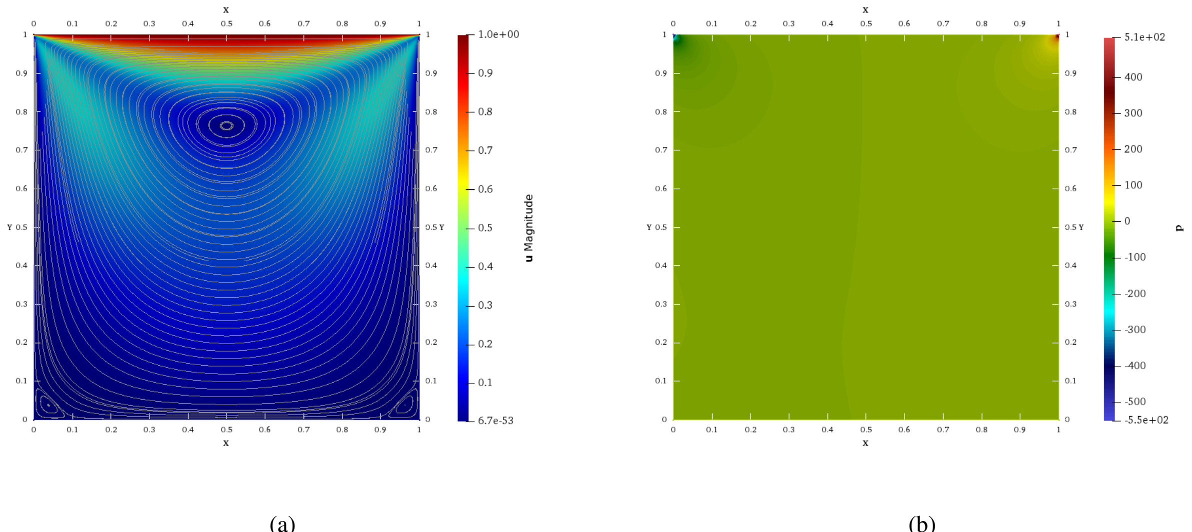
The velocity $\mathbf{u}^*$ and pressure p^* fields were solved using the OTIFEM method for the given perturbations using pyOTI sparse implementation, computing a total of $N = 496$ coefficients for every node. An example code to perform the finite element analysis calculations is provided in Appendix F.3. Finally, the OTIROM for both the velocity and pressure fields were evaluated for specific domain perturbations ΔL_x and ΔL_y,

$$\mathbf{u}|_{L_x+\Delta L_x, L_y+\Delta L_y} \approx \text{OTIROM}(\mathbf{u}^*, \{\Delta L_x, \Delta L_y\}) \tag{5.89}$$

$$p|_{L_x+\Delta L_x, L_y+\Delta L_y} \approx \text{OTIROM}(p^*, \{\Delta L_x, \Delta L_y\}) \tag{5.90}$$

The real results corresponding to the traditional FEM solution are shown in Figure 5.29. The pressure field and characteristic streamlines of solution is in accordance with the reported solutions [88], showing the occurrence of vortices at the center and at the lower right and left corners.

The application of the domain variations ΔL_x and ΔL_y using the reduced model was compared against the finite element solution of the problem on the warped mesh. The new nodal positions can be calculated using an OTIROM of the OTI perturbed nodal coordinates:



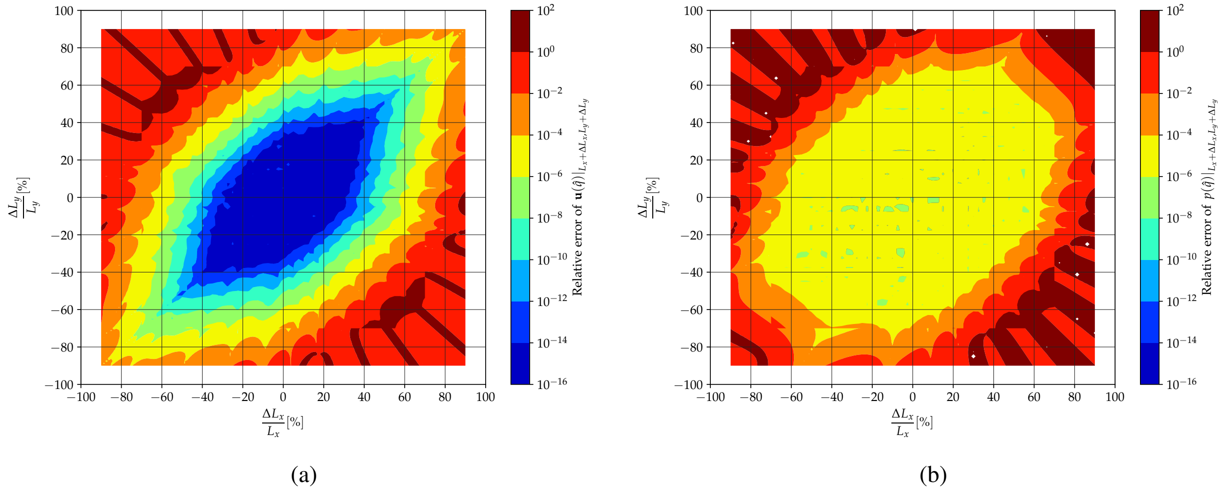
(a)

(b)

Figure 5.29: Magnitude of the velocity field and pressure distribution of the lid-driven cavity problem with dimensions $L_x = L_y = 1$.

$$\mathbf{x}|_{L_x+\Delta L_x, L_y+\Delta L_y} \approx \text{OTIROM}(\mathbf{x}^*, \{\Delta L_x, \Delta L_y\}) \tag{5.91}$$

The relative error distribution of the 30'th order OTIROM evaluated is shown in Figure 5.30 for both the velocity magnitude and pressure at point $\hat{q}$. The velocity reaches lower relative errors than the pressure field, possibly due to the different interpolation functions given by the adopted P2-P1 implementation. A significant area can reach relative error below 10^{-12} for the velocity field, while the pressure error distribution shows that errors below 10^{-4} are possible, with some scattered regions reaching errors below 10^{-6}.

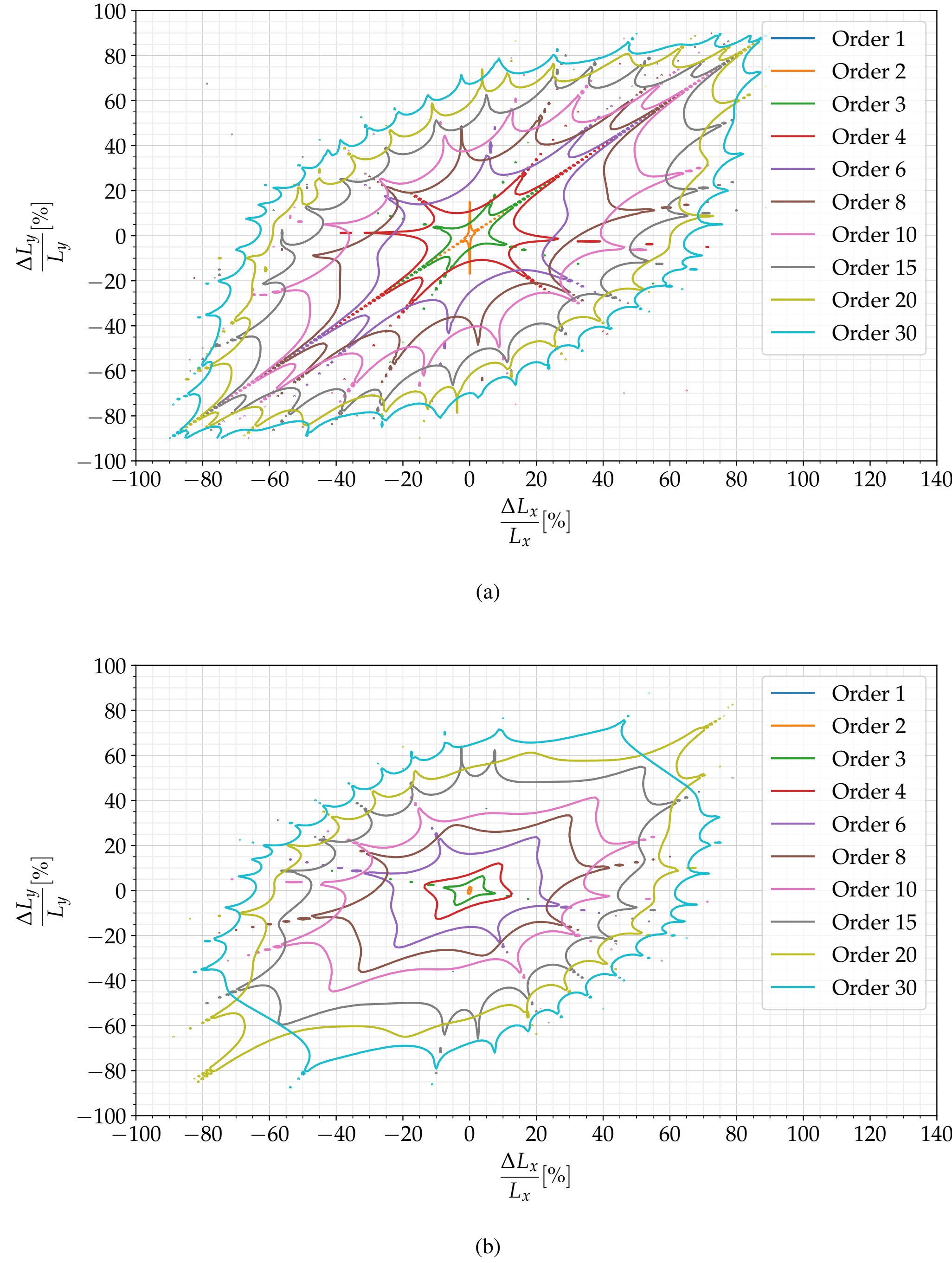


(a)

(b)

Figure 5.30: Relative error distribution of the 30'th order OTIROM generated from OTIFEM analysis to the lid-driven cavity problem for variation of the domain dimensions L_x and L_y. The relative error of (a) the velocity magnitude and (b) the pressure are evaluated at point $\hat{q}$.

The contour lines delimiting the regions where the relative error of the p'th order OTIROM solution lies below 10^{-4} are shown in Figure 5.31. An increase in the order of the OTIROM evaluation, increases the effective region where the solution has an error below 10^{-4}. This implies that the derivatives contained in the OTIFEM result are accurate for the different orders evaluated. For each order, some spikes are noted that locally extend the region of confidence for each OTIROM model. The general shape of the regions of confidence are oval contours, whose major axis is directed along the positive diagonal of the perturbation space, i.e. $\Delta L_x \approx \Delta L_y$; and the minor axis correspond to the negative diagonal where $\Delta L_x \approx -\Delta L_y$. The major axis of the oval region represents domain variations that conserve the aspect ratio of the square domain, which preserves the physical response of the flow (the same vortices are present). In contrast, the minor axes represent significant variations to the aspect ratio of the original domain, and this variation highly changes the physical response of the flow to the new geometry [88] (a transformation of the fluid flow is present).



(a)

(b)

Figure 5.31: Contour lines delimiting the regions with relative error of 10^{-4} for the p'th order OTIROM of the lid-driven cavity problem for (a) the velocity magnitude and (b) the pressure at the point $\hat{q}$ with respect to the FEM solution on the warped mesh.

Figure 5.32 illustrates the result of applying a perturbation of $\Delta L_x = 0L_x$ and $\Delta L_y = 0.7L_y$, for equivalent mesh dimensions of $L'_x = 1$ and $L'_y = 1.7$ to the 30th order OTIROM generated. This variation of the domain is very close to the limit of the 30th order error contour in Figure 5.31 for both pressure and

velocity. The FEM solutions on the deformed domain are shown in Figures 5.32a and 5.32c for the velocity field and pressure distribution (both streamline plots have the exact same source starting points). It is noted that there is a transformation of the vortices from initial the solution (see Figure 5.29), to three vortices in the bottom of the domain, a main vortex that spins along the whole width of the cavity and two inner vortices to the left and right of the center of the main bottom vortex. The pressure field preserves the characteristic two hot-spots at the upper right and left corners of the domain, although reducing the overall magnitude. The approximation from the OTIROM model captures these topology variations as shown in Figures 5.32b and 5.32d. The vortices are captured in the same overall position, showing perceivable differences with respect to specific stream line locations.

The absolute error distribution for the OTIROM result is shown in Figure 5.33 for both velocity magnitude and pressure field. The error is concentrated in two regions close to the $65\%L_y$ and for the horizontal coordinates $30\%L_x$ and $70\%L_x$. The pressure field has a more even distribution compared to the velocity error distribution, with small regions with lower error, in particular the vertical line at the center of the domain.

5.5 Discussion

The hypercomplex Order Truncated Finite Element Method (OTIFEM) was developed to enable computation of multivariable high order derivatives using OTI numbers. Accurate shape, gradient, material and load (boundary condition) high order derivatives were demonstrated in 2D problems of heat transfer, linear elasticity and fluid dynamics. The method can be extended to 3D and other elemental interpolations. First and second order elements were sufficient to accurately represent derivatives of higher order. OTI numbers were capable of computing higher order of derivatives compared to other hypercomplex algebras such as multiduals, as the algebra is smaller in size for an equivalent computation.

For instance, computation of 30th order derivatives is theoretically possible using multidual numbers; it is, however, not possible in practical terms because of memory limitations. Since the number of multidual basis required for this task is 30, then the number of coefficients for a single 30 basis multidual number is $N = 2^{30}$. In practice, for 8-Byte double precision coefficients, the total memory required for this multidual scalar is 8 GB, it is not practical for large problems like finite elements with matrices made by millions of scalars. In contrast, the OTI number used for computation of 30th order derivatives of two variables in examples 5.4.3 and 5.4.4 had a total size of $N = 496$, which uses approximately 8.2 kB in the sparse repre-

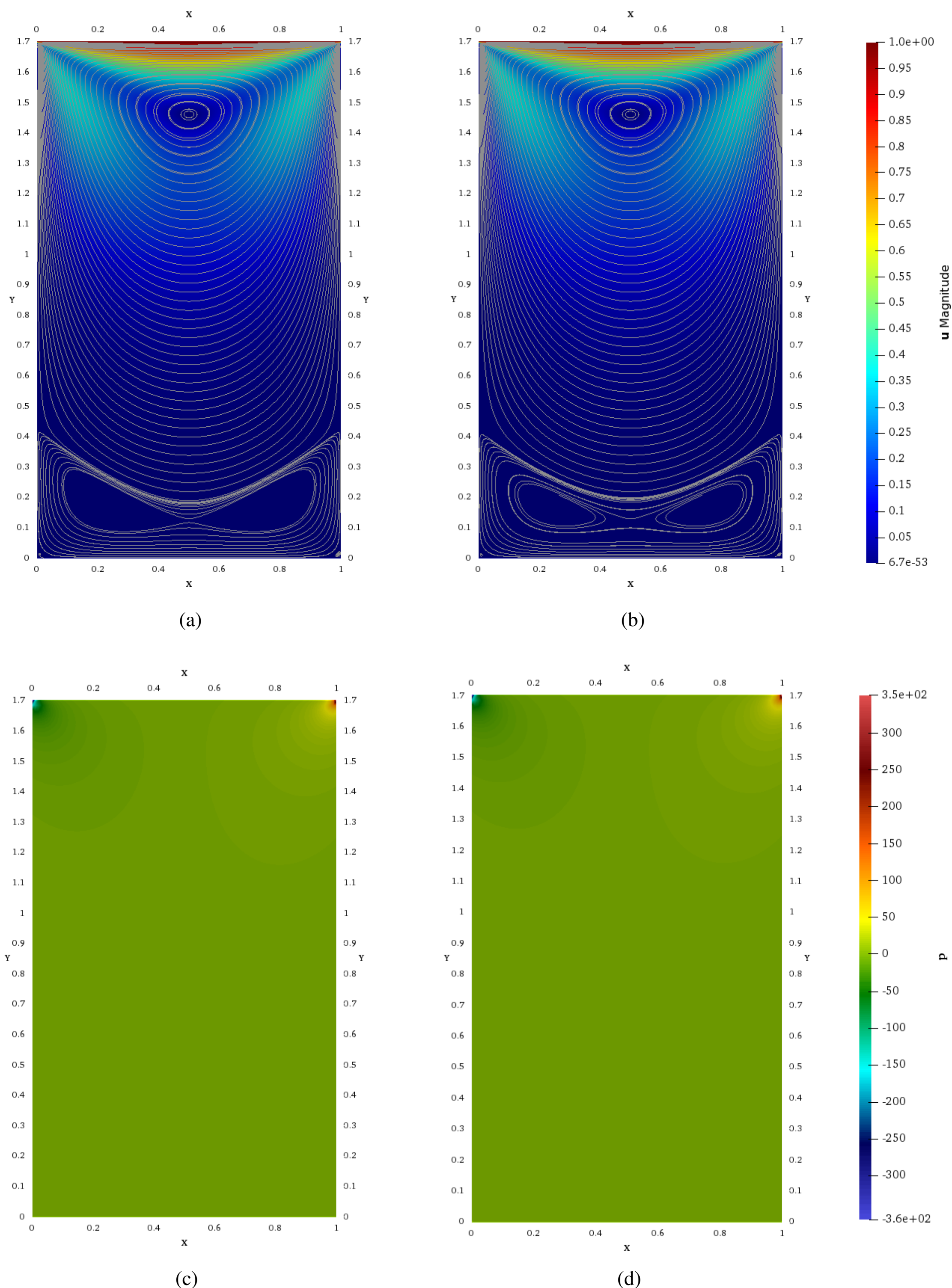


Figure 5.32: Comparison of the results of the simulation of the lid-driven cavity problem for a domain dimensions $L'_x = 1$ and $L'_y = 1.7$. The first row shows the magnitude of the velocity field for (a) the finite element solution on the domain L'_x and L'_y; and (b) the OTIROM generated solving the problem with domain $L_x = 1$ and $L_y = 1$ with perturbed nodal coordinates. The second row of figures show the pressure distribution for (c) the finite element solution on the domain L'_x and L'_y; and (d) the OTIROM approximation.

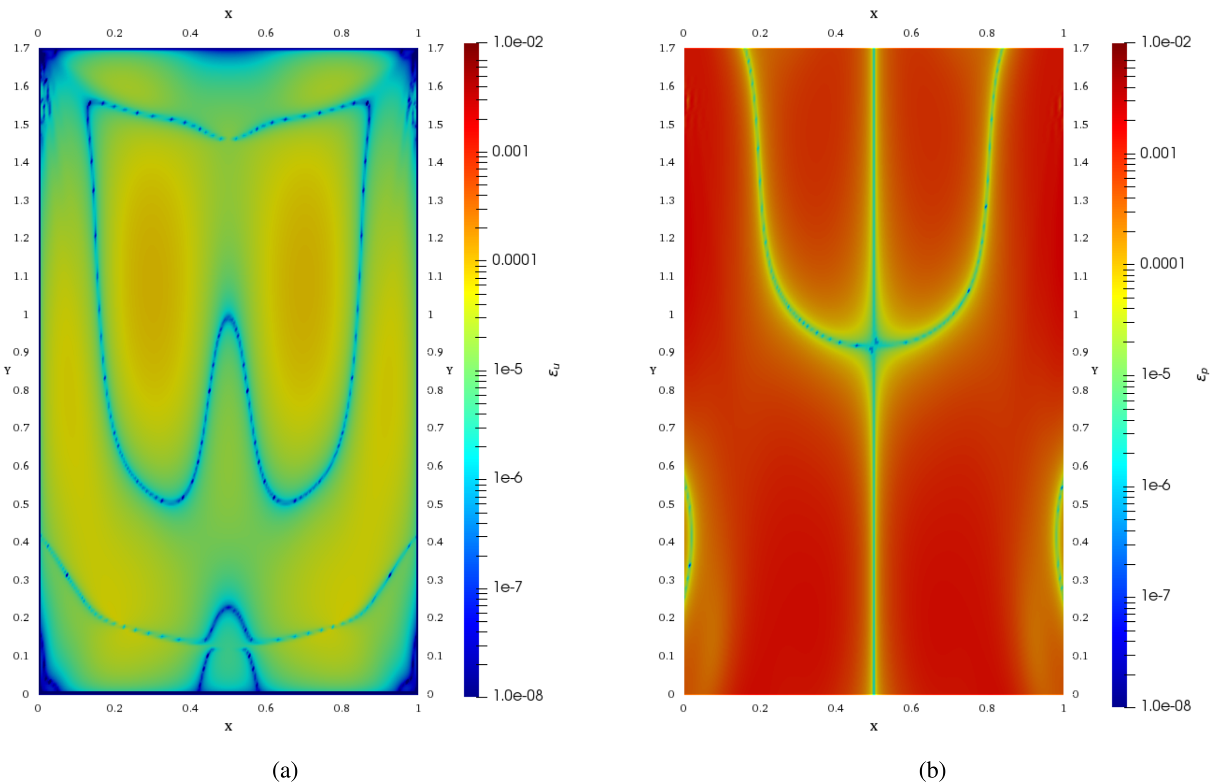


Figure 5.33: Error distribution of the OTI reduced order model for a domain perturbation of $L'_x = 1$ and $L'_y = 1.7$ for (a) the magnitude of the velocity field and (b) the pressure field of the lid-driven cavity simulation.

sentation. This shows that OTI numbers were able to overcome memory limitation of current hypercomplex algebras by allowing the 30th order derivatives to be computed even for cases with large matrix of coefficients. For instance, the Stokes analysis in section 5.4.4 had system matrix with a total of 1.7M non-zero OTI numbers, which requires an estimated 14 GB of memory for computing 495 derivatives of up to 30th order and with respect to two variables.

Memory usage of OTI implementation is, however, proportional to the number of imaginary directions for the required computation. In some cases, the memory required may exceed the capacity of the system, for example when increasing considerably the number of variables perturbed. This can be addressed by adopting the repetitive multidual philosophy the computation of derivatives into batches of variables. For example, when the second order derivatives are required with respect to 3 variables, e.g. x, y and z, then 3 analyses of OTIs with two bases and truncation order $n = 2$ can be performed. A first analysis perturbs x and y finding all mixed derivatives of these variables. Then x and z are perturbed to find all their derivatives. Finally, y and z are perturbed to find the missing mixed derivative. In contrast, multidual requires 6 evaluations of the function to compute all 6 second order derivatives. An alternative solution is to perform the computation of derivatives with an element-by-element approach. This is discussed in [89].

The perturbation region directly influences the accuracy of higher order shape derivatives. The perturbation region for computation of Lagrangian shape derivatives is critical for a stable behavior of high order derivatives. The error of the derivatives of the lower orders was similar with all perturbations applied, however differences were observed as the error accumulated with the computation of the higher order derivatives. Therefore, a correct selection of the perturbation function will allow the correct computation of the higher order derivatives, forming better reduced order models when used for that matter.

It was observed that for those real Finite element analyses dominated by the solution time of the linear system of equations, the incorporation of OTI algebra for computation of derivatives is less significant than for those analyses dominated by the assembly time. For instance, this could be observed by comparing the Cholesky and SuperLU normalized time comparison. In the case of 10th order bivariate derivatives (65 derivatives plus standard solution), a problem dominated by the solution time (SuperLU) required $2.39\times$ the solution time of the equivalent real analysis, meaning that each derivative required approximately 2.1% more time to be computed. For those problems dominated by the assembly of the matrix system, the evaluation time was $13.1\times$ the time of the real FEM simulation, thus implying close to 19% more time per derivative. Also, for a given number imaginary basis, it is more expensive to evaluate higher order derivatives and less

number of variables than to evaluate lower order of derivatives and larger number of variables. This was seen, for instance, for the cases of second order derivatives of ten variables and 10th order derivatives of two variables (both have the same number of imaginary direction), the former required $3.2\times$ the time the 10-variable evaluation required.

Using the repetitive approach from multiduals have the advantage that a single implementation is enough to compute derivatives with respect to any number of variables, but additional effort from the user is required to setup all required evaluations. As seen, multiple evaluations can hit negatively the computational performance as it increases the computational time. In contrast, static dense OTI numbers can evaluate more efficiently but are limited to the number of variables and order of derivatives supported by the algebra. This inconvenience is tackled by the sparse OTI implementation, as the implementation adapts to the number of variables and derivatives to be computed, simplifying the implementation and optimizing the computational performance, at a small cost. The sparse implementation significantly reduces the development time, as the computation of derivatives become merely an input variable perturbation instead of additional programming. Also, sparse OTI numbers may represent significant memory/CPU time savings when the perturbed variables do not propagate through all operations and do not 'fill' all coefficients of every OTI in the system. Significant computational performance improvements are expected when pyOTI algorithms are parallelized and optimized, mostly for the assembly and solution of the first and higher order imaginary coefficients in the solution strategy developed for linear systems of equations. As a consequence, sparse OTI numbers are recommended for prototyping scenarios, whereas static dense are better for deployment purposes.

Chapter 6: CONCLUSIONS, CONTRIBUTIONS AND FUTURE WORK

The complex Taylor series expansion (CTSE) method has been extended to Cayley-Dickson algebras (CDTSE) in order to compute multiple first order derivatives. In particular, quaternion algebra can compute 3 first order derivatives, octonion algebra can compute 7, and sedenion algebra can compute 15. This strategy can be extended to any number of first order derivatives in powers of 2.

Similarly to CTSE, CDTSE does not require subtraction of terms in order to compute a derivative. As a result, the truncation error can be driven below machine precision through the use of a very small step size. Contrary to CTSE, the error for CDTSE is of order h whereas for CTSE the error is of order h^2. However, this difference in error is of no practical importance once a sufficiently small h is used for numerical computations.

The accuracy, versatility and performance of CDTSE in the finite element method was shown in two numerical examples implemented in Abaqus through the use of user element subroutines. The use of QFEM and OFEM user elements allowed one to compute accurate three and seven first order derivatives respectively in heat transfer and linear elasticity problems. Two solution methods were discussed and compared: a direct Cauchy Riemann and a block solver approach. The block solver approach was significantly more efficient than the direct approach, although both resulted in the same output values.

Although significant improvements were obtained, steps towards a more efficient algebra were detected and implemented. As a result, the Order Truncated Imaginary numbers were developed to compute high order multivariable derivatives efficiently. The algebraic operations and method to evaluate elementary functions were described. OTI numbers are more efficient than other hypercomplex algebras as the size of the algebra adapts according to the number of derivatives required. Although the implementation of OTI number is serial, it performed faster in all situations studied compared to an automatic differentiation implementation.

The matrix form of OTI numbers, key element to integrate OTI in the Finite Element Method (FEM), was defined. A "block order" was found. It enables the automatic incorporation with real linear algebra procedures such as Cholesky and LU decomposition for solving linear systems of equations and finding matrix inverses, operations that are fundamental in FEM. This significantly improved the performance as a single decomposition of the stiffness matrix is required to solve the real and imaginary coefficients of the FEM result, which contain the derivatives.

The accuracy of the method was demonstrated with examples in heat transfer, linear elasticity and fluid dynamics analyses, showing that the methodology can be applied to multiple physics. The analyses were implemented using pyOTI, an efficient library developed during this doctoral project that supports of OTI numbers and FEM analyses in Python. Derivatives up to 30th order with respect to two variables (495 derivatives) were computed in a single FEM analysis to generate reduced order models of the finite element solutions with respect to shape and material parameters. Results showed that the approximation accuracy and the range of validity of the approximations effectively increased with respect to the truncation order of the OTI evaluation performed. Also, OTI numbers enabled computations of high order derivatives that were not possible before with other hypercomplex algebras.

6.1 Contribution

The main contributions of this dissertation are:

- The Cayley-Dickson Finite Element Method (CDFEM), an extension of the Complex Taylor Series Expansion (CSTE) to Cayley-Dickson algebras that allows computing multivariable first order derivatives in Finite Element analyses;
- The development of the Order Truncated Imaginary (OTI) numbers, a hypercomplex algebra capable of computing high-order derivatives of multivariable functions and allow to automatically generate reduced order models from the evaluated functions;
- The development of the Order Truncated Finite Element Method (OTIFEM), a method that integrates OTI numbers in Finite Element analyses using the OTI matrix form and enable accurate and efficient computation of high-order multivariable derivatives; and
- A numerical tool, pyOTI, that implements the developed OTI numbers and OTIFEM method with a fast static-dense implementation and a more convenient sparse implementation that adapts better to the problem, but with a computational cost.

The developed methods empowers Finite Element analyses with an easy to use, efficient and robust algebra to compute multivariable high-order derivatives in a single evaluation, removing the limitations that were present previously with other hypercomplex implementations.

6.2 Future work

This work showed that the way the imaginary perturbation to the domain is performed affects the accuracy of the derivatives from the OTIFEM analysis. A followup investigation is suggested to find appropriate shape perturbations to the analysis domain that minimizes computational cost while still finding a satisfactory derivative. Also, a deeper analysis to the behavior of nodal perturbations for gradient computation is suggested, as it may enable computation, for instance, of accurate stress distributions and higher derivatives using linear elements. The sparse implementation of the algebra shows great potential for large analyses, however implementational optimizations are still required. For instance, parallelization of the sparse linear algebra procedures, mainly those utilized in the solution to linear systems of equations are suggested for development. Application to other physics is suggested as well as the implementation of OTIs for crack propagation analysis. Finally, a research on the use of the block-solver approach to solve OTI matrix eigenvalues/eigenvectors is suggested as a method to automate the computation of its derivatives.

Appendix A: HYPERCOMPLEX DIFFERENTIATION METHODS

In a generalized perspective, hypercomplex numbers [26] considers a N-tuple of real coefficients (a_0, a_1, ..., a_{N-1}) related with a set of imaginary directions ($\boldsymbol{\alpha}_0$, $\boldsymbol{\alpha}_1$, ..., $\boldsymbol{\alpha}_{N-1}$) with some conditions. A total number of N imaginary directions exist in the algebra. In general, direction $\boldsymbol{\alpha}_0 = 1$, and represents the real direction. Examples of hypercomplex algebras are complex, dual, Cayley-Dickson (CD) algebras such as quaternions, octonions, sedenions; and others such as multicomplex, multidual, biquaternions, split biquaternions, split quaternions, etc.

To the knowledge of the author, only complex/multicomplex, dual/multidual and quaternion numbers have been reported for computation of derivatives. The purpose of this section is to introduce the concept of hypercomplex algebra numerical differentiation.

A.1 Complex and Dual Taylor Series Expansion Methods

Complex numbers $\mathbb{C}$ have one imaginary unit $\boldsymbol{\alpha}_1 = i$ and a corresponding imaginary condition $i^2 = -1$. The Complex Taylor Series Expansion (CTSE) [25,61] is a methodology to compute the sensitivity of a real function with respect to a variable of interest that is analogous to the finite difference (FD) method in the sense that a perturbation is applied to the parameter of interest. However, in CTSE the perturbation of the variable of interest x is applied along the imaginary axis, becoming

$$x^* = x_0 + hi \tag{A.1}$$

where x_0 represents the point at which the derivative of the function is required and h is the perturbation step. The typical procedure consists of replacing every operation in the real function of interest f by its complex-equivalent operation (a process known as complexification of f), e.g. real multiplication is replaced by complex multiplication, etc. Using the complex-variable Taylor series, a function f can be expressed as

$$f(x^*) = f(x_0 + hi) = f(x_0) + f_{,x}(x_0)h\ i - \frac{1}{2!}f_{,x^2}(x_0)h^2 - \frac{1}{3!}f_{,x^3}(x_0)h^3\ i + \text{H.O.T.} \tag{A.2}$$

where $f(x^*)$ is the result of complexifying $f(x)$ (the latter one evaluated in a real number and the first in the complex number). Also, $f_{,x}$ represents the first order derivative with respect to x, $f_{,x^2}$ the second

order derivative, etc. Note that the negative signs in Equation (A.2) come from factoring terms with i^2 and replacing it by the complex identity $i^2 = -1$. Taking the real and imaginary parts of both sides, Re[·] and Im[·] respectively; and solving for the function and its first order derivative, the following is obtained

$$f(x_0) = \text{Re}\left[f(x^*)\right] + \mathcal{O}(h^2) \tag{A.3}$$

$$f_{,x}(x_0) = \frac{1}{h}\text{Im}\left[f(x^*)\right] + \mathcal{O}(h^2) \tag{A.4}$$

Note that the perturbation step size h can be made arbitrarily small with no concern about subtraction cancellation error. Hence, higher order effects of $\mathcal{O}(h^2)$ can be made negligible through the use of a small h. A value for h can be 10^{-30} [53].

Dual numbers [9] have the imaginary direction $\boldsymbol{\alpha}_1 = \epsilon$ and imaginary condition $\epsilon^2 = 0$. In contrast to CTSE, equations (3.2) and (3.3) become insensitive to the step size h because terms multiple of ϵ^2 become zero. For simplicity $h = 1$ is used, and perturbation to the input variable

$$x^* = x_0 + \epsilon \tag{A.5}$$

is used to obtain exact-to-machine-error values for the function and its derivative, namely

$$f(x_0) = \text{Re}\left[f(x_0 + \epsilon)\right] \tag{A.6}$$

$$f_{,x}(x_0) = \text{Im}\left[f(x_0 + \epsilon)\right] \tag{A.7}$$

Complex and Dual numbers can only be used to compute first order derivatives with respect to one variable per function evaluation. Both methodologies are subtractive cancellation error free.

A.2 Multicomplex and Multidual Taylor Series Expansion

Multicomplex [7,30] and multidual [8] algebras can be used to compute multivariable high order derivatives in a single analysis. For example, a bicomplex ($\mathbb{C}_2$) number has two imaginary bases i_1 and i_2 that form 3

imaginary directions, i.e. $\boldsymbol{\alpha}_1 = i_1$, $\boldsymbol{\alpha}_2 = i_2$ and $\boldsymbol{\alpha}_3 = i_1 i_2$. The number can be written as

$$x^* = x_r + x_{i_1} i_1 + x_{i_2} i_2 + x_{i_1 i_2} i_1 i_2,\ x^* \in \mathbb{C}_2 \tag{A.8}$$

where $i_1^2 = i_2^2 = -1$ but $i_1 i_2 = i_2 i_1 \neq -1$.

Similarly, bidual numbers ($\mathbb{D}_2$) have two imaginary basis ϵ_1 and ϵ_2 and three imaginary directions, i.e. $\boldsymbol{\alpha}_1 = \epsilon_1$, $\boldsymbol{\alpha}_2 = \epsilon_2$ and $\boldsymbol{\alpha}_3 = \epsilon_1 \epsilon_2$. The number can be written as

$$x^* = x_r + x_{\epsilon_1} \epsilon_1 + x_{\epsilon_2} \epsilon_2 + x_{\epsilon_1 \epsilon_2} \epsilon_1 \epsilon_2 \tag{A.9}$$

where $\epsilon_1^2 = \epsilon_2^2 = 0$ but $\epsilon_1 \epsilon_2 = \epsilon_2 \epsilon_1 \neq 0$. The following notation will be used to extract the coefficient associated to an imaginary direction, e.g. the coefficient of $\epsilon_1 \epsilon_2$ is retrieved as $x_{\epsilon_1 \epsilon_2} = \mathrm{Im}_{\epsilon_1 \epsilon_2}[x^*]$.

A tricomplex number has three bases i_1, i_2 and i_3, and 7 imaginary directions, $\boldsymbol{\alpha}_1 = i_1$, $\boldsymbol{\alpha}_2 = i_2$, $\boldsymbol{\alpha}_3 = i_1 i_2$, $\boldsymbol{\alpha}_4 = i_3$, $\boldsymbol{\alpha}_5 = i_1 i_3$, $\boldsymbol{\alpha}_6 = i_2 i_3$ and $\boldsymbol{\alpha}_7 = i_1 i_2 i_3$ (plus the real direction). Similarly, a tridual number has three bases ϵ_1, ϵ_2 and ϵ_3, and 7 imaginary directions, $\boldsymbol{\alpha}_1 = \epsilon_1$, $\boldsymbol{\alpha}_2 = \epsilon_2$, $\boldsymbol{\alpha}_3 = \epsilon_1 \epsilon_2$, $\boldsymbol{\alpha}_4 = \epsilon_3$, $\boldsymbol{\alpha}_5 = \epsilon_1 \epsilon_3$, $\boldsymbol{\alpha}_6 = \epsilon_2 \epsilon_3$ and $\boldsymbol{\alpha}_7 = \epsilon_1 \epsilon_2 \epsilon_3$. Hence, the numbers $x^* \in \mathbb{C}_3$ and $y^* \in \mathbb{D}_3$ can be written as:

$$x^* = x_r + x_{i_1} i_1 + x_{i_2} i_2 + x_{i_1 i_2} i_1 i_2 + x_{i_3} i_3 + x_{i_1 i_3} i_1 i_3 + x_{i_2 i_3} i_2 i_3 + x_{i_1 i_2 i_3} i_1 i_2 i_3, \tag{A.10}$$

$$y^* = y_r + y_{\epsilon_1} \epsilon_1 + y_{\epsilon_2} \epsilon_2 + y_{\epsilon_1 \epsilon_2} \epsilon_1 \epsilon_2 + y_{\epsilon_3} \epsilon_3 + y_{\epsilon_1 \epsilon_3} \epsilon_1 \epsilon_3 + y_{\epsilon_2 \epsilon_3} \epsilon_2 \epsilon_3 + y_{\epsilon_1 \epsilon_2 \epsilon_3} \epsilon_1 \epsilon_2 \epsilon_3, \tag{A.11}$$

where the conditions $i_1^2 = i_2^2 = i_3^2 = -1$ are imposed for the tricomplex number and imaginary directions $i_1 i_2$, $i_2 i_3$, $i_1 i_3$ and $i_1 i_2 i_3$ exist and $i_1 i_2 \neq i_1 i_2 i_3 \neq -1$. For triduals, the imaginary conditions are $\epsilon_1^2 = \epsilon_2^2 = \epsilon_3^2 = 0$ and $\epsilon_1 \epsilon_2$, $\epsilon_2 \epsilon_3$, $\epsilon_1 \epsilon_3$ and $\epsilon_1 \epsilon_2 \epsilon_3$ exist and $\epsilon_1 \epsilon_2 \neq \epsilon_1 \epsilon_2 \epsilon_3 \neq 0$.

A tetracomplex number ($\mathbb{C}_4$) has four bases i_1, i_2, i_3 and i_4, and 15 imaginary directions,

$$\begin{aligned} x^* = x_r + x_{i_1} i_1 + x_{i_2} i_2 + x_{i_1 i_2} i_1 i_2 + x_{i_3} i_3 + x_{i_1 i_3} i_1 i_3 \\ + x_{i_2 i_3} i_2 i_3 + x_{i_1 i_2 i_3} i_1 i_2 i_3 + x_{i_4} i_4 + x_{i_1 i_4} i_1 i_4 + x_{i_2 i_4} i_2 i_4 + x_{i_1 i_2 i_4} i_1 i_2 i_4 + \\ x_{i_3 i_4} i_3 i_4 + x_{i_1 i_3 i_4} i_1 i_3 i_4 + x_{i_2 i_3 i_4} i_2 i_3 i_4 + x_{i_1 i_2 i_3 i_4} i_1 i_2 i_3 i_4 \end{aligned} \tag{A.12}$$

In general, the total number of imaginary directions N of multicomplex and multidual numbers is determined by the number of imaginary bases m as,

$$N = 2^m \tag{A.13}$$

A.2.1 Computation of Single Variable High Order Derivatives

Consider a single variable function $f : \mathbb{R} \to \mathbb{R}$, and the following bicomplex perturbation of variable x,

$$x^* = x_0 + \underline{hi_1} + \underline{hi_2} + 0i_1i_2 \tag{A.14}$$

the Taylor series expansion of f becomes

$$f(x^*) = f(x_0 + hi_1 + hi_2) = f(x_0) + f_{,x}(x_0)hi_1 + f_{,x}(x_0)hi_2 + f_{,x^2}(x_0)h^2i_1i_2 + \text{H.O.T.} \tag{A.15}$$

To obtain derivatives using multicomplex,

$$f(x_0) = \text{Re} \; [f(x_0 + hi_1 + hi_2)] + \mathcal{O}(h^2) \tag{A.16}$$

$$f_{,x}(x_0) = \frac{1}{h}\text{Im}_{i_1} \; [f(x_0 + hi_1 + hi_2)] + \mathcal{O}(h^2) \tag{A.17}$$

$$f_{,x}(x_0) = \frac{1}{h}\text{Im}_{i_2} \; [f(x_0 + hi_1 + hi_2)] + \mathcal{O}(h^2) \tag{A.18}$$

$$f_{,x^2}(x_0) = \frac{1}{h^2}\text{Im}_{i_1i_2} [f(x_0 + hi_1 + hi_2)] + \mathcal{O}(h^2) \tag{A.19}$$

and using multidual numbers,

$$f(x_0) = \text{Re} \quad [f(x_0 + \epsilon_1 + \epsilon_2)] \tag{A.20}$$

$$f_{,x}(x_0) = \text{Im}_{\epsilon_1} \quad [f(x_0 + \epsilon_1 + \epsilon_2)] \tag{A.21}$$

$$f_{,x}(x_0) = \text{Im}_{\epsilon_2} \quad [f(x_0 + \epsilon_1 + \epsilon_2)] \tag{A.22}$$

$$f_{,x^2}(x_0) = \text{Im}_{\epsilon_1\epsilon_2} [f(x_0 + \epsilon_1 + \epsilon_2)] \tag{A.23}$$

which are insensitive to the value of h, thus $h = 1$ is used for simplicity. However, using a sufficiently small step size h can achieve the same accuracy as multiduals in computer applications.

In general, the technique consists of perturbing the sensitivity variable by a step size h in n imaginary basis. The algebra must have at least n basess, thus its number of coefficients is given by

$$N = 2^n \tag{A.24}$$

The result will contain all derivatives up to order n. Mixed directions with p bases will contain the p'th order derivative of f with respect to x, as in Equation A.19. A tricomplex number is required to compute the third order derivatives of f. The perturbation is done as

$$x^* = x_0 + \underline{hi_1} + \underline{hi_2} + 0i_1i_2 + \underline{hi_3} + 0i_1i_3 + 0i_2i_3 + 0i_1i_2i_3 \tag{A.25}$$

where the first order derivative will be contained and repeated in terms i_1, i_2 and i_3; second order derivative will be in the coefficients of i_1i_2, i_1i_3 and i_2i_3; and the third order derivative will be in the coefficient of the mixed direction $i_1i_2i_3$.

A.2.2 Computation of Multivariable High Order Derivatives

Considering now a two variable function $f : \mathbb{R}^2 \rightarrow \mathbb{R}$, and the following bicomplex perturbation of variables x and y,

$$x^* = x_0 + \underline{hi_1} + 0i_2 + 0i_1i_2 \quad (A.26)$$

$$y^* = y_0 + 0i_1 + \underline{hi_2} + 0i_1i_2 \quad (A.27)$$

the Taylor series expansion of $f(x^*, y^*)$ can be expressed as

$$f(x^*, y^*) = f(x_0, y_0) + hf_{,x}(x_0, y_0)i_1 + hf_{,y}(x_0, y_0)i_2 + h^2 f_{,xy}(x_0, y_0)i_1i_2 + \text{H.O.T} \quad (A.28)$$

To obtain the derivatives, using bicomplex numbers,

$$f(x_0, y_0) = \text{Re} \quad [f(x_0 + hi_1, y_0 + hi_2)] + \mathcal{O}(h^2) \quad (A.29)$$

$$f_{,x}(x_0, y_0) = \frac{1}{h}\text{Im}_{i_1} \quad [f(x_0 + hi_1, y_0 + hi_2)] + \mathcal{O}(h^2) \quad (A.30)$$

$$f_{,y}(x_0, y_0) = \frac{1}{h}\text{Im}_{i_2} \quad [f(x_0 + hi_1, y_0 + hi_2)] + \mathcal{O}(h^2) \quad (A.31)$$

$$f_{,xy}(x_0, y_0) = \frac{1}{h^2}\text{Im}_{i_1i_2} [f(x_0 + hi_1, y_0 + hi_2)] + \mathcal{O}(h^2) \quad (A.32)$$

and can be easily extended to bidual numbers. Notice that both first order derivatives are obtained but only o the mixed second order derivative is obtained in the mixed term i_1i_2 A.32, from the 3 second order derivatives possible from function f. To compute three first order derivatives will require one tricomplex/tridual analysis. If perturbations are performed as

$$x^* = x_0 + \underline{hi_1} + 0i_2 + 0i_1i_2 + 0i_3 + 0i_1i_3 + 0i_2i_3 + 0i_1i_2i_3 \quad (A.33)$$

$$y^* = y_0 + 0i_1 + \underline{hi_2} + 0i_1i_2 + 0i_3 + 0i_1i_3 + 0i_2i_3 + 0i_1i_2i_3 \quad (A.34)$$

$$z^* = z_0 + 0i_1 + 0i_2 + 0i_1i_2 + \underline{hi_3} + 0i_1i_3 + 0i_2i_3 + 0i_1i_2i_3 \quad (A.35)$$

then, computing derivatives of a 3 variable function $f : \mathbb{R}^3 \to \mathbb{R}$ becomes

$$
\begin{aligned}
f(x_0, y_0, z_0) &= \text{Re} \quad [f(x_0 + hi_1, y_0 + hi_2, z_0 + hi_3)] + \mathcal{O}(h^2) && \text{(A.36)} \\
f_{,x}(x_0, y_0, z_0) &= \frac{1}{h}\text{Im}_{i_1} \quad [f(x_0 + hi_1, y_0 + hi_2, z_0 + hi_3)] + \mathcal{O}(h^2) && \text{(A.37)} \\
f_{,y}(x_0, y_0, z_0) &= \frac{1}{h}\text{Im}_{i_2} \quad [f(x_0 + hi_1, y_0 + hi_2, z_0 + hi_3)] + \mathcal{O}(h^2) && \text{(A.38)} \\
f_{,xy}(x_0, y_0, z_0) &= \frac{1}{h^2}\text{Im}_{i_1 i_2} [f(x_0 + hi_1, y_0 + hi_2, z_0 + hi_3)] + \mathcal{O}(h^2) && \text{(A.39)} \\
f_{,z}(x_0, y_0, z_0) &= \frac{1}{h}\text{Im}_{i_3} \quad [f(x_0 + hi_1, y_0 + hi_2, z_0 + hi_3)] + \mathcal{O}(h^2) && \text{(A.40)} \\
f_{,xz}(x_0, y_0, z_0) &= \frac{1}{h^2}\text{Im}_{i_1 i_3} [f(x_0 + hi_1, y_0 + hi_2, z_0 + hi_3)] + \mathcal{O}(h^2) && \text{(A.41)} \\
f_{,yz}(x_0, y_0, z_0) &= \frac{1}{h^2}\text{Im}_{i_2 i_3} [f(x_0 + hi_1, y_0 + hi_2, z_0 + hi_3)] + \mathcal{O}(h^2) && \text{(A.42)} \\
f_{,xyz}(x_0, y_0, z_0) &= \frac{1}{h^3}\text{Im}_{i_1 i_2 i_3} [f(x_0 + hi_1, y_0 + hi_2, z_0 + hi_3)] + \mathcal{O}(h^2) && \text{(A.43)}
\end{aligned}
$$

noticing that the result has the incomplete set of second and third order derivatives. For the case of tridual, then $h = 1$ and the result is exact.

In particular, if the purpose is to compute multiple first order derivatives the technique consists of perturbing each sensitivity variable by a step size h in a basis direction. The result will contain the first order derivatives in the coefficients associated to the corresponding pure imaginary directions, see equations (A.30) and (A.31). However, the coefficients associated to the mixed directions will contain the higher order derivatives, e.g. second order derivative is contained in the term $i_1 i_2$, see equations (A.28) and (A.32).

If the complete set of n order derivatives is required, then two approaches can be followed: A single evaluation approach where all derivatives are obtained in a single analysis and a multiple evaluation approach where each evaluation uses a different perturbation scheme. Both are discussed in what follows.

A.2.2.1 Single Evaluation

For most applications, the complete set of n order derivatives are required, and thus the results obtained in the previous bicomplex and tricomplex analyses are incomplete. As a consequence, a tetracomplex analysis is required to obtain all second order terms in a single analysis. Perturbations are done, in a simplified manner, as follows

$$x^* = x_0 + \underline{hi_1} + \underline{hi_2} \tag{A.44}$$

$$y^* = y_0 + \underline{hi_3} + \underline{hi_4} \tag{A.45}$$

A total of 16 terms will be obtained after evaluating $f(x^*, y^*)$, containing all first and second order derivatives, but incomplete third and fourth order derivatives, that is

$$\begin{aligned} f(x^*, y^*) = f(x_0, y_0) + hf_{,x}i_1 + hf_{,x}i_2 + h^2 f_{,x^2}i_1 i_2 + hf_{,y}i_3 + h^2 f_{,xy}i_1 i_3 \\ + h^2 f_{,xy}i_2 i_3 + h^3 f_{,x^2y}i_1 i_2 i_3 + hf_{,y}i_4 + h^2 f_{,xy}i_1 i_4 + h^2 f_{,xy}i_2 i_4 + h^3 f_{,x^2y}i_1 i_2 i_4 + \\ h^2 f_{,y^2}i_3 i_4 + h^3 f_{,xy^2}i_1 i_3 i_4 + h^3 f_{,xy^2}i_2 i_3 i_4 + h^4 f_{,x^2y^2}i_1 i_2 i_3 i_4 \end{aligned} \tag{A.46}$$

In general, if derivatives up to order n are to be computed, the technique consists of perturbing each sensitivity variable by a step size h in m imaginary basis. As a consequence, for m variable functions, the total number of basis required to perform the full analysis is mn. Therefore, the dimension of the associated hypercomplex algebra is

$$N = 2^{nm} \tag{A.47}$$

A.2.2.2 Multiple Evaluations

As seen in Section A.2.1, a bicomplex analysis can be used to obtain the second order derivatives of a single variable function by perturbing the variable of interest in two pure directions. Moreover, a bicomplex analysis can be used to compute the mixed derivative of a two variable function by perturbing each variable in one pure imaginary direction. As a consequence, in order to obtain all second order of a two variable function $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ using multiple analyses one proceeds as follows:

i) perturb x to obtain the first and second order derivatives of the function with respect to x,

$$x^* = x_0 + \underline{hi_1} + \underline{hi_2} + 0i_1i_2 \quad (A.48)$$

$$y^* = y_0 + 0i_1 + 0i_2 + 0i_1i_2 \quad (A.49)$$

the Taylor series expansion of $f(x^*, y^*)$ can be expressed as

$$f(x^*, y^*) = f(x_0, y_0) + hf_{,x}(x_0, y_0)i_1 + hf_{,x}(x_0, y_0)i_2 + h^2 f_{,x^2}(x_0, y_0)i_1i_2 + \text{H.O.T} \quad (A.50)$$

ii) perturb x and y in order to obtain the mixed second order derivative of the function,

$$x^* = x_0 + \underline{hi_1} + 0i_2 + 0i_1i_2 \quad (A.51)$$

$$y^* = y_0 + 0i_1 + \underline{hi_2} + 0i_1i_2 \quad (A.52)$$

the Taylor series expansion of $f(x^*, y^*)$ can be expressed as

$$f(x^*, y^*) = f(x_0, y_0) + hf_{,x}(x_0, y_0)i_1 + hf_{,y}(x_0, y_0)i_2 + h^2 f_{,xy}(x_0, y_0)i_1i_2 + \text{H.O.T} \quad (A.53)$$

iii) finally, perturb y in order to obtain the first and second order derivatives of the function with respect to y:

$$x^* = x_0 + 0i_1 + 0i_2 + 0i_1i_2 \quad (A.54)$$

$$y^* = y_0 + \underline{hi_1} + \underline{hi_2} + 0i_1i_2 \quad (A.55)$$

the Taylor series expansion of $f(x^*, y^*)$ can be expressed as

$$f(x^*, y^*) = f(x_0, y_0) + hf_{,y}(x_0, y_0)i_1 + hf_{,y}(x_0, y_0)i_2 + h^2 f_{,y^2}(x_0, y_0)i_1i_2 + \text{H.O.T} \quad (A.56)$$

Using this repetitive method, the global size of the algebra was reduced from tetracomplex (16 coefficients) to bicomplex (4 coefficients). However, it requires to evaluate the function $f(x^*, y^*)$ three times. The requirement of the algebra is therefore bounded by the maximum order of derivative n to be computed. As a consequence, the dimension of the multicomplex algebra required to compute the derivative is

$$N = 2^n \tag{A.57}$$

The total number of function evaluations is given by the total number of n order derivatives that exist for an m variable function. This is given by

$$N_{\text{evals}} = \binom{n+m-1}{m-1} = \frac{(n+m-1)!}{n!(m-1)!} \tag{A.58}$$

Table A.1 shows the values of Equation (A.58) for multiple m variables and n order of derivatives. It can be observed that the number of function evaluations grows quickly with respect to the number of variables and order. As shown in the table, a one variable function only needs a single multidual/multicomplex evaluation for any order. It, however, needs

	Order (n)									
Number of Variables (m)	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	1	1	1	1	1	1	1	1	1	1
2	2	3	4	5	6	7	8	9	10	11
3	3	6	10	15	21	28	36	45	55	66
4	4	10	20	35	56	84	120	165	220	286
5	5	15	35	70	126	210	330	495	715	1001
6	6	21	56	126	252	462	792	1287	2002	3003
7	7	28	84	210	462	924	1716	3003	5005	8008
8	8	36	120	330	792	1716	3432	6435	11440	19448
9	9	45	165	495	1287	3003	6435	12870	24310	43758
10	10	55	220	715	2002	5005	11440	24310	48620	92378

Table A.1: Growth of the number of evaluations the repetitive approach does with respect to the number of variables (m) and order of derivatives (n).

A.3 Matrix and Vector Forms of Hypercomplex Numbers

Hypercomplex algebras have equivalent matrix and vector representations that let the algebras be operated using standard linear algebra operations. The vector form of a complex number $x^* = x_r + x_i i$ is

$$\mathbf{t}(x^*) = \begin{Bmatrix} x_r \\ x_i \end{Bmatrix} \tag{A.59}$$

and for the case of dual numbers $x^* = x_r + x_\epsilon \epsilon$,

$$\mathbf{t}(x^*) = \begin{Bmatrix} x_r \\ x_\epsilon \end{Bmatrix} \tag{A.60}$$

In general, the vector form is defined as the tuple of all coefficients in the imaginary algebra.

$$\mathbf{t}(x^*) = \begin{Bmatrix} \text{Re} & [x^*] \\ \text{Im}_{\boldsymbol{\alpha}_1} & [x^*] \\ \text{Im}_{\boldsymbol{\alpha}_2} & [x^*] \\ & \vdots \\ \text{Im}_{\boldsymbol{\alpha}_{N-1}} & [x^*] \end{Bmatrix} \tag{A.61}$$

The matrix form of hypercomplex algebras is referred as the Cauchy-Riemann (CR) matrix [66]. The CR matrix depends on each algebra. For example, the matrix form of complex number $x^* = x_r + x_i i$ is

$$\mathbf{T}(x^*) = \begin{bmatrix} x_r & -x_i \\ x_i & x_r \end{bmatrix} \tag{A.62}$$

and the matrix form of a dual number $x^* = x_r + x_\epsilon \epsilon$ is,

$$\mathbf{T}(x^*) = \begin{bmatrix} x_r & 0 \\ x_\epsilon & x_r \end{bmatrix} \tag{A.63}$$

Tables A.2 and A.3 summarize the CR matrices for multicomplex and multidual numbers respectively.

As shown in Tables A.2 and A.3, multicomplex matrix form is a dense matrix with total number of non-zero coefficients N_{T} given by

$$N_{\text{T}} = N \times N = 2^{2N_{\text{Bases}}} \tag{A.64}$$

The multidual matrix is a lower triangular matrix of the same dimension as the multicomplex matrix form. However, the number of non-zero elements in the matrix form is given by

	Cauchy Riemann matrix form
Bicomplex	$\begin{bmatrix} x_r & -x_{i_1} & -x_{i_2} & x_{i_1i_2} \\ x_{i_1} & x_r & -x_{i_1i_2} & -x_{i_2} \\ x_{i_2} & -x_{i_1i_2} & x_r & -x_{i_1} \\ x_{i_1i_2} & x_{i_2} & x_{i_1} & x_r \end{bmatrix}$
Tricomplex	$\begin{bmatrix} x_r & -x_{i_1} & -x_{i_2} & x_{i_1i_2} & -x_{i_3} & x_{i_1i_3} & x_{i_2i_3} & -x_{i_1i_2i_3} \\ x_{i_1} & x_r & -x_{i_1i_2} & -x_{i_2} & -x_{i_1i_3} & -x_{i_3} & x_{i_1i_2i_3} & x_{i_2i_3} \\ x_{i_2} & -x_{i_1i_2} & x_r & -x_{i_1} & -x_{i_2i_3} & x_{i_1i_2i_3} & -x_{i_3} & x_{i_1i_3} \\ x_{i_1i_2} & x_{i_2} & x_{i_1} & x_r & -x_{i_1i_2i_3} & -x_{i_2i_3} & -x_{i_1i_3} & -x_{i_3} \\ x_{i_3} & -x_{i_1i_3} & -x_{i_2i_3} & x_{i_1i_2i_3} & x_r & -x_{i_1} & -x_{i_2} & x_{i_1i_2} \\ x_{i_1i_3} & x_{i_3} & -x_{i_1i_2i_3} & -x_{i_2i_3} & x_{i_1} & x_r & -x_{i_1i_2} & -x_{i_2} \\ x_{i_2i_3} & -x_{i_1i_2i_3} & x_{i_3} & -x_{i_1i_3} & x_{i_2} & -x_{i_1i_2} & x_r & -x_{i_1} \\ x_{i_1i_2i_3} & x_{i_2i_3} & x_{i_1i_3} & x_{i_3} & x_{i_1i_2} & x_{i_2} & x_{i_1} & x_r \end{bmatrix}$

Table A.2: Equivalent Cauchy Riemann matrix form of bicomplex and tricomplex numbers.

	Cauchy Riemann matrix form
Bidual	$\begin{bmatrix} x_r & 0 & 0 & 0 \\ x_{\epsilon_1} & x_r & 0 & 0 \\ x_{\epsilon_2} & 0 & x_r & 0 \\ x_{\epsilon_1\epsilon_2} & x_{\epsilon_2} & x_{\epsilon_1} & x_r \end{bmatrix}$
Tridual	$\begin{bmatrix} x_r & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ x_{\epsilon_1} & x_r & 0 & 0 & 0 & 0 & 0 & 0 \\ x_{\epsilon_2} & 0 & x_r & 0 & 0 & 0 & 0 & 0 \\ x_{\epsilon_1\epsilon_2} & x_{\epsilon_2} & x_{\epsilon_1} & x_r & 0 & 0 & 0 & 0 \\ x_{\epsilon_3} & 0 & 0 & 0 & x_r & 0 & 0 & 0 \\ x_{\epsilon_1\epsilon_3} & x_{\epsilon_3} & 0 & 0 & x_{\epsilon_1} & x_r & 0 & 0 \\ x_{\epsilon_2\epsilon_3} & 0 & x_{\epsilon_3} & 0 & x_{\epsilon_2} & 0 & x_r & 0 \\ x_{\epsilon_1\epsilon_2\epsilon_3} & x_{\epsilon_2\epsilon_3} & x_{\epsilon_1\epsilon_3} & x_{\epsilon_3} & x_{\epsilon_1\epsilon_2} & x_{\epsilon_2} & x_{\epsilon_1} & x_r \end{bmatrix}$

Table A.3: Equivalent Cauchy Riemann matrix form of bidual and tridual numbers.

$$N_{\mathrm{T}} = 3^{N_{\text{Bases}}} \quad \text{(A.65)}$$

A.4 Integration with the Finite Element Method

The method to integrate hypercomplex algebras in the finite element method is named ZFEM [44, 50, 64]. Briefly, the method consist in uplifting every input variable to hypercomplex, so that imaginary perturbations can be applied. Also, elemental computations are also uplifted to hypercomplex by adding the required degrees of freedom to hold the imaginary coefficients from the hypercomplex algebra used. An example of this is presented in Figure A.1 for bicomplex algebra.

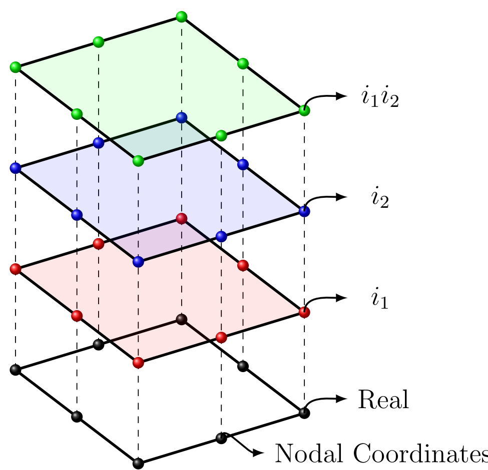


Figure A.1: Representation of the nodal layers according to the imaginary directions in the hypercomplex number.

A hypercomplex system of equations is formed after the finite element computations,

$$\mathbf{K}^* \mathbf{u}^* = \mathbf{f}^* \quad \text{(A.66)}$$

where $\mathbf{u}^*$ is the solution vector of the finite element problem, now with additional degrees of freedom,

$$\mathbf{u}^* \quad = \quad \mathbf{u}_r + \mathbf{u}_{i_1} i_1 + \mathbf{u}_{i_2} i_2 + \mathbf{u}_{i_1 i_2} i_1 i_2 \quad \text{(A.67)}$$

where the vectors $\mathbf{u}_{i_1}$, $\mathbf{u}_{i_2}$ and $\mathbf{u}_{i_1 i_2}$ contain the nodal derivatives of the state function with respect to the perturbed parameters. For example, if a problem variable ϕ_1 is pertubed as $\phi_1^* = \phi_1 + h i_1 + h i_2$, then

derivatives are obtained as follows

$$
\begin{aligned}
\mathbf{u} &\approx \mathrm{Re}\left[\mathbf{u}^*\right] && \text{(A.68)}\\
\mathbf{u}_{,\phi_1} &\approx \frac{1}{h}\mathrm{Im}_{i_1}\left[\mathbf{u}^*\right] && \text{(A.69)}\\
\mathbf{u}_{,\phi_1} &\approx \frac{1}{h}\mathrm{Im}_{i_2}\left[\mathbf{u}^*\right] && \text{(A.70)}\\
\mathbf{u}_{,\phi_1^2} &\approx \frac{1}{h^2}\mathrm{Im}_{i_1 i_2}\left[\mathbf{u}^*\right] && \text{(A.71)}
\end{aligned}
$$

Details on specific transformation of the Finite Element Method to compute the global stiffness matrix $\mathbf{K}^*$ and force vector $\mathbf{f}^*$ are included in section 3.5.1. Solution of the system of equations A.66 relies heavily on the matrix form of the algebra, as it allows interfacing with standard *real*-solvers.

A.4.1 Solution of Hypercomplex System of Equations

The hypercomplex system of equations (A.66) can not be solved directly using standard linear algebra libraries, because every element of the matrix $\mathbf{K}^*$ and vectors $\mathbf{u}^*$ and $\mathbf{f}^*$ are hypercomplex type. It is possible, however, to transform the hypercomplex system into a linear system of equations such that every element of the system is formed only by real coefficients. The transformation is driven by the matrix and vector forms of the corresponding hypercomplex number (see Section A.3). The equivalent system of equations for the bicomplex case is

$$
\overbrace{\begin{bmatrix}
\mathbf{K}_r & -\mathbf{K}_{i_1} & -\mathbf{K}_{i_2} & \mathbf{K}_{i_1 i_2}\\
\mathbf{K}_{i_1} & \mathbf{K}_r & -\mathbf{K}_{i_1 i_2} & -\mathbf{K}_{i_2}\\
\mathbf{K}_{i_2} & -\mathbf{K}_{i_1 i_2} & \mathbf{K}_r & -\mathbf{K}_{i_1}\\
\mathbf{K}_{i_1 i_2} & \mathbf{K}_{i_2} & \mathbf{K}_{i_1} & \mathbf{K}_r
\end{bmatrix}}^{\mathbf{T}(\mathbf{K}^*)}
\overbrace{\begin{Bmatrix}\mathbf{u}_r\\ \mathbf{u}_{i_1}\\ \mathbf{u}_{i_2}\\ \mathbf{u}_{i_1 i_2}\end{Bmatrix}}^{\mathbf{t}(\mathbf{u}^*)}
=
\overbrace{\begin{Bmatrix}\mathbf{f}_r\\ \mathbf{f}_{i_1}\\ \mathbf{f}_{i_2}\\ \mathbf{f}_{i_1 i_2}\end{Bmatrix}}^{\mathbf{t}(\mathbf{f}^*)}
\qquad \text{(A.72)}
$$

The resulting system can solved using standard real linear algebra solvers, however note that since the matrix forms of multicomplex and multidual numbers are not symmetric, therefore requires non symmetric solver even if the original system of equations is symmetric. The size of the system has been also increased as the total number fo degrees of freedom has been multiplied a factor equal to the number of imaginary directions of the algebra.

Another approach, reported using multiduals in [48], is to use ZFEM as the means to compute the derivatives for the semi-analytic method.

A.5 Summary

The current hypercomplex differentiation methods, through multicomplex and multidual algebras, provide a simple, highly accurately and versatile methodology to compute high order derivatives of multivariable functions. These methods require minimum transformation of source code as computer implementations are made usually through operation overloading. This methods have been used successfully in a large variety of problems, including Finite Element simulations. However, the methods do not scale well because the size of the algebras increase exponentially ($N = 2^m$), and therefore prevent a wider adoption. On one hand, the method to compute all derivatives in one analysis (Section A.2.2.1) is not computationally efficient for large problems requiring derivatives with respect to a large number of variables, because memory requirements increase exponentially. On the other hand, the repetitive approach makes the approach viable for few variables (≤ 3) and low orders of derivatives (≤ 3), but larger number of variables and higher orders of derivatives require large number of evaluations. The later fact makes the method unattractive because multiple evaluations significantly increase significantly the CPU time.

Appendix B: QUATERNIONS

Quaternion numbers, $\mathbb{H}$, are a subset of hypercomplex numbers. They extend the complex numbers to three imaginary units. A number q^* is a quaternion when

$$q^* = q_r + q_i i + q_j j + q_k k \tag{B.1}$$

where q_r, q_i, q_j and q_k are real coefficients and i, j, k are imaginary units governed by the following conditions

$$i^2 = j^2 = k^2 = ijk = -1 \tag{B.2}$$

$$ij = -ji = k \tag{B.3}$$

$$jk = -kj = i \tag{B.4}$$

$$ki = -ik = j \tag{B.5}$$

B.1 Algebra

Quaternion addition is defined by the addition of real coefficients of the same imaginary units, whereas product operation is defined by imaginary conditions in Equation (B.2). Assume two quaternion numbers q^* and p^* are defined as follows:

$$\begin{aligned} q^* &= q_r + q_i i + q_j j + q_k k \\ p^* &= p_r + p_i i + p_j j + p_k k \end{aligned} \tag{B.6}$$

Addition of p^* and q^* is calculated as:

$$q^* + p^* = (q_r + p_r) + (q_i + p_i)i + (q_j + p_j)j + (q_k + p_k)k \tag{B.7}$$

and the product of q^* and p^* results in

$$
\begin{aligned}
q^*p^* = (q_rp_r - q_ip_i - q_jp_j - q_kp_k) + \\
(q_rp_i + q_ip_r + q_jp_k - q_kp_j)i + \\
(q_rp_j - q_ip_k + q_jp_r + q_kp_i)j + \\
(q_rp_k + q_ip_j - q_jp_i + q_kp_r)k
\end{aligned}
\tag{B.8}
$$

As seen in Equation (B.8), p^*q^* differs from q^*p^*. Therefore, this algebra is non-commutative.

Multiplication of a quaternion by a scalar, i.e. a quaternion with zero imaginary coefficients, is defined as

$$q^*\alpha = \alpha q^* = \alpha q_r + \alpha q_i i + \alpha q_j j + \alpha q_k k \tag{B.9}$$

The conjugate $\bar{q^*}$ of a Quaternion number q^* is

$$\bar{q^*} = q_r - q_i i - q_j j - q_k k \tag{B.10}$$

such that multiplication of $\bar{q^*}q^*$ eliminates all imaginary directions

$$\bar{q^*}q^* = q^*\bar{q^*} = q_r^2 + q_i^2 + q_j^2 + q_k^2 \tag{B.11}$$

Consider a quaternion number $p^* \in \mathbb{H}$ with its conjugate as defined in Equation (B.10). The norm of p^* is defined as

$$||p^*|| = \sqrt{p^*\bar{p^*}} \tag{B.12}$$

As indicated in [26], Quaternion is a division algebra, because an inverse operation can be defined:

$$\frac{1}{p^*} = p^{*-1} = \frac{\bar{p^*}}{||p^*||^2} \tag{B.13}$$

Therefore, division is performed as

$$\frac{q^*}{p^*} = q^*\frac{1}{p^*} = \frac{q^*\bar{p^*}}{||p^*||^2} \tag{B.14}$$

B.2 Matrix and vector representations

Quaternion algebra is isomorphic to matrix algebra. This implies that quaternion numbers can be transformed to an equivalent matrix such that operations like addition and multiplication are equivalent to its corresponding quaternion algebra operations. In general, given the transformation $\mathbf{T}(\cdot)$ from quaternion to matrix space and two quaternion numbers q^* and p^* as Equation (B.6), then

$$\begin{aligned} \mathbf{T}(q^* + p^*) &= \mathbf{T}(q^*) + \mathbf{T}(p^*) \quad &(B.15) \\ \mathbf{T}(q^* p^*) &= \mathbf{T}(q^*)\mathbf{T}(p^*) \quad &(B.16) \end{aligned}$$

Transformation $\mathbf{T}(\cdot)$ is not unique; however, there is no preferential form if quaternions are used for computation of derivatives. A possible matrix form of a quaternion q^* as in Equation (B.1) is the following:

$$\mathbf{T}(q^*) = \begin{bmatrix} q_r & -q_i & -q_j & -q_k \\ q_i & q_r & -q_k & q_j \\ q_j & q_k & q_r & -q_i \\ q_k & -q_j & q_i & q_r \end{bmatrix} \quad (B.17)$$

Notice the transformation generates an asymmetric matrix composed by real coefficients. Therefore, this transformation allows one to use real-only algebra with no imaginary concepts involved. This can be useful in a computational implementation.

A vector form also exists for quaternions. For quaternion q^*, its vector form is

$$\mathbf{t}(q^*) = \begin{bmatrix} q_r & q_i & q_j & q_k \end{bmatrix}^T \quad (B.18)$$

The transformation from quaternion to vector space, $\mathbf{t}(\cdot)$, relates to quaternion operations as follows

$$\begin{aligned} \mathbf{t}(q^* + p^*) &= \mathbf{t}(q^*) + \mathbf{t}(p^*) \quad &(B.19) \\ \mathbf{t}(q^* p^*) &= \mathbf{T}(q^*)\,\mathbf{t}(p^*) \quad &(B.20) \end{aligned}$$

Notice that only transformation of the multiplication operation requires the matrix form of the multiplier (left term), meanwhile the multiplicand (right term) is in its vector form. Moreover, addition may be done using the corresponding vector forms. The advantage of using using matrix and vector representation is that arithmetic operations can be computed using linear algebra libraries.

In summary, quaternion algebra can be represented by vector and matrix forms. As shown in Equations (B.19) and (B.20), quaternion addition can be computed using the vector form, and multiplication can be computed using a matrix-vector product.

B.3 Quaternion Taylor Series Expansion

The aim of this section is to derive the quaternion Taylor Series Expansion for computing derivatives, shown in Equation (3.5), and to show that the convergence of the approximated derivatives is linear with respect to the step size. Consider a quaternion function, $\mathcal{F} : \mathbb{H}^3 \to \mathbb{H}$, that depends on three quaternion inputs x^*, y^* and z^*. From a real function perspective, $\mathcal{F}$ is formed by four real functions,

$$\mathcal{F}(x^*, y^*, z^*) = f_{\mathrm{r}}(\mathbf{x}) + f_i(\mathbf{x})i + f_j(\mathbf{x})j + f_k(\mathbf{x})k \tag{B.21}$$

where, $x^* = x_{\mathrm{r}} + x_i i + x_j j + x_k k$, $y^* = y_{\mathrm{r}} + y_i i + y_j j + y_k k$, $z^* = z_{\mathrm{r}} + z_i i + z_j j + z_k k$, $\mathbf{x}$ is a vector formed by all real coefficients of the quaternion inputs of $\mathcal{F}$, namely

$$\mathbf{x} = \begin{bmatrix} x_{\mathrm{r}} & x_i & x_j & x_k & y_{\mathrm{r}} & y_i & y_j & y_k & z_{\mathrm{r}} & z_i & z_j & z_k \end{bmatrix}^T \tag{B.22}$$

where the number of real input parameters corresponds to four times the number of input quaternion variables of the function $\mathcal{F}$. Also, $f_{\mathrm{r}}(\mathbf{x})$, $f_i(\mathbf{x})$, $f_j(\mathbf{x})$ and $f_k(\mathbf{x})$ are real functions that generate the real, i'th, j'th and k'th components of $\mathcal{F}$, respectively. It is considered that the function $\mathcal{F}$ is isomorphic to a real function f, i.e., when evaluated at $x^* = x + 0i + 0j + 0k$, $y^* = y + 0i + 0j + 0k$ and $z^* = z + 0i + 0j + 0k$, the function behaves as

$$\mathcal{F}(x, y, z) = f(x, y, z) + 0i + 0j + 0k \tag{B.23}$$

The derivative of a quaternion function with respect to a real parameter t is given by

$$\mathcal{F}_{,t}(t) = f_{\mathrm{r},t}(t) + f_{i,t}(t)i + f_{j,t}(t)j + f_{k,t}(t)k \tag{B.24}$$

where $\mathcal{F}_{,t}(t)$ is $d\mathcal{F}/dt$, $f_{\mathrm{r},t}(t)$ is df_{r}/dt, etc.

The Taylor series expansion of the quaternion function can be achieved by generating the Taylor series expansions of its four real functions with respect to all real variables in $\mathbf{x}$,

$$f_{\mathrm{r}}(\mathbf{x}_0 + \mathbf{h}) = f_{\mathrm{r}}(\mathbf{x}_0) + \nabla f_{\mathrm{r}}(\mathbf{x}_0)^T\mathbf{h} + \frac{1}{2}\mathbf{h}^T\mathbf{H}_{\mathrm{r}}(\mathbf{x}_0)\mathbf{h} + \mathbf{H.O.T.} \tag{B.25}$$

$$f_i(\mathbf{x}_0 + \mathbf{h}) = f_i(\mathbf{x}_0) + \nabla f_i(\mathbf{x}_0)^T\mathbf{h} + \frac{1}{2}\mathbf{h}^T\mathbf{H}_i(\mathbf{x}_0)\mathbf{h} + \mathbf{H.O.T.} \tag{B.26}$$

$$f_j(\mathbf{x}_0 + \mathbf{h}) = f_j(\mathbf{x}_0) + \nabla f_j(\mathbf{x}_0)^T\mathbf{h} + \frac{1}{2}\mathbf{h}^T\mathbf{H}_j(\mathbf{x}_0)\mathbf{h} + \mathbf{H.O.T.} \tag{B.27}$$

$$f_k(\mathbf{x}_0 + \mathbf{h}) = f_k(\mathbf{x}_0) + \nabla f_k(\mathbf{x}_0)^T\mathbf{h} + \frac{1}{2}\mathbf{h}^T\mathbf{H}_k(\mathbf{x}_0)\mathbf{h} + \mathbf{H.O.T.} \tag{B.28}$$

where $\mathbf{h}$ contains the perturbations of all variables in $\mathbf{x}$,

$$\mathbf{h} = \begin{bmatrix} h_{x_{\mathrm{r}}} & h_{x_i} & h_{x_j} & h_{x_k} & h_{y_{\mathrm{r}}} & h_{y_i} & h_{y_j} & h_{y_k} & h_{z_{\mathrm{r}}} & h_{z_i} & h_{z_j} & h_{z_k} \end{bmatrix}^T \tag{B.29}$$

$\mathbf{x}_0$ is a vector containing the center or reference points of all Taylor series expansions

$$\mathbf{x}_0 = \begin{bmatrix} x_{0\mathrm{r}} & x_{0i} & x_{0j} & x_{0k} & y_{0\mathrm{r}} & y_{0i} & y_{0j} & y_{0k} & z_{0\mathrm{r}} & z_{0i} & z_{0j} & z_{0k} \end{bmatrix}^T \tag{B.30}$$

the matrices $\mathbf{H}_{\mathrm{r}}(\mathbf{x})$, $\mathbf{H}_i(\mathbf{x})$, $\mathbf{H}_j(\mathbf{x})$ and $\mathbf{H}_k(\mathbf{x})$ are the Hessian matrices of the real functions f_{r}, f_i, f_j and f_k respectively, with respect to all parameters in $\mathbf{x}$, and the term $\mathbf{H.O.T.}$ contains the higher order terms from the Taylor series.

Using quaternion notation, Equations (B.25)-(B.28) can be simplified in the following manner

$$\mathcal{F}(x_0^* + \Delta x^*, y_0^* + \Delta y^*, z_0^* + \Delta z^*) = \mathcal{F}(x_0^*, y_0^*, z_0^*) + \nabla\mathcal{F}(x_0^*, y_0^*, z_0^*)^T\mathbf{h} + \frac{1}{2}\mathbf{h}^T\boldsymbol{\mathcal{H}}(x_0^*, y_0^*, z_0^*)\mathbf{h} + \mathbf{H.O.T.} \tag{B.31}$$

where $\boldsymbol{\mathcal{H}}$ contains the Hessian matrices of the the functions in $\mathcal{F}$

$$\boldsymbol{\mathcal{H}}(x_0^*, y_0^*, z_0^*) = \mathbf{H}_{\mathrm{r}}(\mathbf{x}_0) + \mathbf{H}_i(\mathbf{x}_0)i + \mathbf{H}_j(\mathbf{x}_0)j + \mathbf{H}_k(\mathbf{x}_0)k \tag{B.32}$$

x_0^*, y_0^* , z_0^* and Δx^*, Δy^*, Δz^*, respectively correspond to the quaternion center points and perturbations:

$$
\begin{aligned}
x_0^* &= x_{0\mathrm{r}} + x_{0i}i + x_{0j}j + x_{0k}k, \qquad \Delta x^* = h_{x_\mathrm{r}} + h_{x_i}i + h_{x_j}j + h_{x_k}k,\\
y_0^* &= y_{0\mathrm{r}} + y_{0i}i + y_{0j}j + y_{0k}k, \qquad \Delta y^* = h_{y_\mathrm{r}} + h_{y_i}i + h_{y_j}j + h_{y_k}k,\\
z_0^* &= z_{0\mathrm{r}} + z_{0i}i + z_{0j}j + z_{0k}k, \qquad \Delta z^* = h_{z_\mathrm{r}} + h_{z_i}i + h_{z_j}j + h_{z_k}k
\end{aligned} \tag{B.33}
$$

In the context of Quaternion Taylor series for computation of derivatives of real functions as described in section 3.2, the center points and perturbations are

$$
\begin{aligned}
x_0^* &= x_0 + 0i + 0j + 0k, \qquad \Delta x^* = 0 + hi + 0j + 0k,\\
y_0^* &= y_0 + 0i + 0j + 0k, \qquad \Delta y^* = 0 + 0i + hj + 0k,\\
z_0^* &= z_0 + 0i + 0j + 0k, \qquad \Delta z^* = 0 + 0i + 0j + hk
\end{aligned} \tag{B.34}
$$

Taking the values of Equation (B.34) and inserting them in Equation (B.31), leads to

$$
\begin{aligned}
&\mathcal{F}(x_0 + hi, y_0 + hj, z_0 + hk) = \mathcal{F}(x_0, y_0, z_0) + h\left(\mathcal{F}_{,x_i} + \mathcal{F}_{,y_j} + \mathcal{F}_{,z_k}\right)\\
&+ \frac{1}{2}h^2\left(\mathcal{F}_{,x_i^2} + \mathcal{F}_{,y_j^2} + \mathcal{F}_{,z_k^2} + \mathcal{F}_{,x_iy_j} + \mathcal{F}_{,x_iz_k} + \mathcal{F}_{,y_jx_i} + \mathcal{F}_{,z_jx_i} + \mathcal{F}_{,y_jz_k} + \mathcal{F}_{,z_ky_j}\right) + \mathcal{O}(h^3)
\end{aligned} \tag{B.35}
$$

where $\mathcal{F}_{,\alpha}$ represents the derivative of $\mathcal{F}$ with respect to α, $\partial\mathcal{F}/\partial\alpha$. Similarly, $\mathcal{F}_{,\alpha^2}$ and $\mathcal{F}_{,\alpha\beta}$ represent the second order derivatives $\partial^2\mathcal{F}/\partial\alpha^2$ and $\partial^2\mathcal{F}/\partial\alpha\partial\beta$ respectively. All derivatives are evaluated at $(x^*, y^*, z^*) = (x_0, y_0, z_0)$. Notice that the imaginary coefficients of the evaluation points are zero, and the derivatives of $\mathcal{F}$ are with respect to real terms, thus calculated using Equation (B.24).

Since $\mathcal{F}$ is isomorphic to f (see Equation (B.23)), it can be shown that $\mathcal{F}$ and its first order derivatives $\mathcal{F}_{,x_i}$, $\mathcal{F}_{,y_j}$ and $\mathcal{F}_{,z_k}$ evaluated at (x_0, y_0, z_0) correspond to:

$$\mathcal{F}(x_0, y_0, z_0) = f(x_0, y_0, z_0) + 0i + 0j + 0k \tag{B.36}$$

$$\mathcal{F}_{,x_i}(x_0, y_0, z_0) = 0 + f_{,x}(x_0, y_0, z_0)i + 0j + 0k \tag{B.37}$$

$$\mathcal{F}_{,y_j}(x_0, y_0, z_0) = 0 + 0i + f_{,y}(x_0, y_0, z_0)j + 0k \tag{B.38}$$

$$\mathcal{F}_{,z_k}(x_0, y_0, z_0) = 0 + 0i + 0j + f_{,z}(x_0, y_0, z_0)k \tag{B.39}$$

and, similarly, the second order derivatives $\mathcal{F}_{,x_i^2}$, $\mathcal{F}_{,y_j^2}$ and $\mathcal{F}_{,z_k^2}$ are related to the second order derivatives of f as

$$\mathcal{F}_{,x_i^2}(x_0, y_0, z_0) = -f_{,x^2}(x_0, y_0, z_0) + 0i + 0j + 0k \quad \text{(B.40)}$$

$$\mathcal{F}_{,y_j^2}(x_0, y_0, z_0) = -f_{,y^2}(x_0, y_0, z_0) + 0i + 0j + 0k \quad \text{(B.41)}$$

$$\mathcal{F}_{,z_k^2}(x_0, y_0, z_0) = -f_{,z^2}(x_0, y_0, z_0) + 0i + 0j + 0k \quad \text{(B.42)}$$

Notice that the derivatives of the quaternion function $\mathcal{F}$, Equations (B.36)-(B.42), are directly related to the derivatives of its isomorphic real function f. This, however, is not the case for the second order mixed derivatives of $\mathcal{F}$ because in general, they are result of operations between terms with two different imaginary directions, thus being influenced by the distribution of terms in $\mathcal{F}$ due to the non-commutativity quaternion property. This makes the magnitude of the second order derivative dependant on the distribution and arrangement of terms in the quaternion function and therefore it may differ from the second order mixed derivative of f.

The mixed second order derivative result lays only on the direction of the product between the imaginary directions of the derivands, e.g. $\mathcal{F}_{,\alpha_k \beta_j}$ lies on i and so forth. Consequently, the general case of the mixed second order quaternion derivatives evaluated at (x_0, y_0, z_0) will be non-zero and directed along the imaginary directions i, j and k as shown below.

$$\mathcal{F}_{,z_k y_j} + \mathcal{F}_{,y_j z_k} = 0 + 2\gamma_i i + 0j + 0k \quad \text{(B.43)}$$

$$\mathcal{F}_{,x_i z_k} + \mathcal{F}_{,z_k x_i} = 0 + 0i + 2\gamma_j j + 0k \quad \text{(B.44)}$$

$$\mathcal{F}_{,x_i y_j} + \mathcal{F}_{,y_j x_i} = 0 + 0i + 0j + 2\gamma_k k \quad \text{(B.45)}$$

where γ_i, γ_j and γ_k are real functions that depend on the input coefficients $\mathbf{x}$.

Consequently, the quaternion Taylor series expansion for the applications discussed in section 3.2 is

$$\mathcal{F}(x_0 + hi, y_0 + hj, z_0 + hk) = f(x_0, y_0, z_0) - \frac{1}{2}h^2\left(f_{,x^2} + f_{,y^2} + f_{,z^2}\right) + \left(hf_{,x} + h^2\gamma_i\right)i + \left(hf_{,y} + h^2\gamma_j\right)j + \left(hf_{,z} + h^2\gamma_k\right)k + O(h^3) \quad \text{(B.46)}$$

and taking the real and imaginary coefficients of Equation (B.46) and then solving for the function value and its first order derivatives, the following is obtained

$$f(x_0, y_0, z_0) = \text{Re}\ [\mathcal{F}(x^*, y^*, z^*)] + h^2 \left(f_{,x^2} + f_{,y^2} + f_{,z^2}\right)/2 + \mathcal{O}(h^3) \quad \text{(B.47)}$$

$$f_{,x}(x_0, y_0, z_0) = \text{Im}_i\,[\mathcal{F}(x^*, y^*, z^*)]/h - h\gamma_i(\mathbf{x}) + \mathcal{O}(h^2) \quad \text{(B.48)}$$

$$f_{,y}(x_0, y_0, z_0) = \text{Im}_j\,[\mathcal{F}(x^*, y^*, z^*)]/h - h\gamma_j(\mathbf{x}) + \mathcal{O}(h^2) \quad \text{(B.49)}$$

$$f_{,z}(x_0, y_0, z_0) = \text{Im}_k\,[\mathcal{F}(x^*, y^*, z^*)]/h - h\gamma_k(\mathbf{x}) + \mathcal{O}(h^2) \quad \text{(B.50)}$$

which shows that QTSE approximates the function quadratically, but approximates the first order derivatives linearly with respect to h.

Reorganizing terms of Equation (B.46) by first order (h) and then second order terms (h^2)

$$\mathcal{F}(x_0 + hi, y_0 + hj, z_0 + hk) = f(x_0, y_0, z_0) + h\,(f_{,x} + f_{,y} + f_{,z}) + h^2 \left(-\frac{1}{2}\left(f_{,x^2} + f_{,y^2} + f_{,z^2}\right) + \gamma_i i + \gamma_j j + \gamma_k k\right) + O(h^3) \quad \text{(B.51)}$$

leads to Equation (3.5) by collecting terms with $\mathcal{O}(h^2)$ and higher.

This justification extrapolates to higher dimensional Cayley Dickson algebras such as octonions, sedenions, etc. The only cases where quadratic convergence is obtained in the derivative approximation are those cases whose second order mixed derivatives of the function $\mathcal{F}$ have magnitude zero when evaluated at (x_0, y_0, z_0).

B.4 Fortran support library

It is straightforward to develop a library to support hypercomplex operations needed for finite element methods hereby presented. Operator overloading is used to simplify the development and modification of current programs in order to use quaternion algebra. The library is composed by a single Fortran module that defines the type structure, functions and operator overloads. It contains zero (scalar), one (vector) and two (matrix) dimensional array support.

Scalar operations are the minimal requirement to apply hypercomplex algebras in the computation of

derivatives. In addition, vector and matrix support is provided in order to simplify coding of finite element subroutines. The functions currently implemented in the library are listed in Table B.1. Operations **TOMATRIX**, **UNFOLD** and **FOLD** are extrinsic functions. All other implemented operations are overloads to intrinsic Fortran functions.

Operation	Fortran operator/function
Addition	+
Multiplication	*
Division	/
Power	**
Sine	SIN
Cosine	COS
Tangent	TAN
Exponential	EXP
Logarithm	LOG
Vector dot product	DOT_PRODUCT
Matrix multiplication	MATMUL
Matrix transpose	TRANSPOSE
Conversion to equiv. matrix form	**TOMATRIX**
Expand quaternion matrix to equiv. matrix form	**UNFOLD**
Convert from expanded form to standard form	**FOLD**

Table B.1: List of supported operations and functions available in the Fortran library.

B.5 Scalar Type

Table B.2 lists the minimal functions needed to support scalar operations of a hypercomplex algebra. The library implements operations that follow the algebra as well as functions of quaternions using the truncated Taylor series approach (see section 3.4).

Operation	Fortran operator
Addition	+
Multiplication	*
Division	/
Power	**

Table B.2: List of minimum operations and functions available in the Quaternion Fortran package

Operator overloads are linked to the quaternion type so that manipulation of the quaternion variables are straightforward and also to reduce the modifications required to adapt current codes to quaternion variables. Overloaded operators were "+", "−" , "*", "/", "**" as well as functions such as sine, cosine, etc.

A minimal version of the quaternion module is given below in order to provide the fundamental functions

and operator overloads required to evaluate the example listed in section 3.3.

The Fortran source code to perform the numerical example listed in section 3.3 using quaternion numbers is given below. The code computes the derivatives of $f(a, b, c, d, x, y, z) = \sin(a^3 b^2 c^3 d^4 x^3 y^2 z^4)$ with respect to a, b, c, d, x, y and z using three quaternion evaluations of the function f.

B.6 Vector / Matrix support.

The library supports vector/matrix of quaternion type.

```
TYPE(QTRN)        :: QA(N,   N) ! Matrix A*
TYPE(QTRN)        :: QC(N, 2*N) ! Matrix C*
TYPE(QTRN)        :: QF(N)      ! Vector f*
```

Figure B.1: Example of how to define vectors and matrices using a Fortran quaternion module.

Element-wise operator overloads are supported with: "+", "−" and "⋆". Also, operations of scalar-vector/scalar-matrix of quaternions types are supported using "+", "−" , "⋆" and "/".

Matrix operations such as matrix multiplications and transpose are supported through the overloaded `TRANSPOSE` and `MATMUL` functions. Vector dot product is performed using the `DOT_PRODUCT` function. An example of its use is given below.

Quaternion matrix/vector specific functions such as expand operations to equivalent vector and matrix forms are provided using the functions **`FOLD`** and **`UNFOLD`**.

B.7 Source Code

A minimal version of the quaternion module is given in Figure B.3 in order to provide the fundamental functions and operator overloads required to evaluate the example listed in section 3.3. The actual code used to evaluate the function to compute all 7 derivatives is shown in Figure B.4.

```
!...

! Type definitions.
TYPE(QTRN)       :: QB(N, N)   ! Matrix of quaternion numbers
TYPE(QTRN)       :: QD(N, N)   ! Matrix of quaternion numbers
! Result quaternion variables
TYPE(QTRN)       :: QK(N, N)   ! Matrix of quaternion numbers
TYPE(QTRN)       :: QF(N)      ! Vector of quaternion numbers

! Real value equivalents
DOUBLE PRECISION :: F(4*N)
DOUBLE PRECISION :: K(4*N,4*N)

! ...

! Perform  K*=B*^T D* B*
QK = MATMUL(TRANSPOSE(QB),MATMUL(QD,QB))

! Extract the results into real valued
K = UNFOLD(QK)
F = UNFOLD(QF)

! - or equivalently -

F(    1:  N) = QF%RE ! Real coefficients
F(  N+1:2*N) = QF%I  !  "i" coefficients
F(2*N+1:3*N) = QF%J  !  "j" coefficients
F(3*N+1:4*N) = QF%K  !  "k" coefficients

  ! ...
```

Figure B.2: Example of how to define vectors and matrices using a Fortran quaternion module.

```
MODULE QUATERNIONS  ! A MODULE FOR BASIC QUARTERNION OPERATIONS
  IMPLICIT NONE
  INTEGER, PARAMETER :: REAL_KIND = 8
  ! MEMORY STRUCTURE OF THE QUATERNION TYPE
  TYPE QTRN
     REAL(REAL_KIND) :: RE  ! REAL COMPONENT
     REAL(REAL_KIND) :: I   !-
     REAL(REAL_KIND) :: J   ! | - IMAGINARY COMPONENTS
     REAL(REAL_KIND) :: K   !-
  END TYPE QTRN
! OPERATOR OVERLADS
INTERFACE OPERATOR(*)          ! QUARTERNION PRODUCT OPERATOR
   MODULE PROCEDURE QPROD
END INTERFACE
!
INTERFACE OPERATOR(**)         ! QUARTERNION POWER OPERATOR
   MODULE PROCEDURE QPOW
END INTERFACE
!
INTERFACE SIN                  ! QUARTERNION SIN (TAYLOR SERIES)
  MODULE PROCEDURE TSIN
END INTERFACE
!
CONTAINS   ! DEFINE THE FUNCTION OVERLOADS
  FUNCTION QPROD(Q1,Q2) RESULT(QN) ! RETURNS THE QUATERNION PRODUCT
    IMPLICIT NONE
    TYPE(QTRN), INTENT(IN) :: Q1, Q2
    TYPE(QTRN) :: QN
    QN%RE = Q1%RE*Q2%RE- Q1%I*Q2%I - Q1%J*Q2%J - Q1%K*Q2%K
    QN%I  = Q1%RE*Q2%I + Q1%I*Q2%RE+ Q1%J*Q2%K - Q1%K*Q2%J
    QN%J  = Q1%RE*Q2%J - Q1%I*Q2%K + Q1%J*Q2%RE+ Q1%K*Q2%I
    QN%K  = Q1%RE*Q2%K + Q1%I*Q2%J - Q1%J*Q2%I + Q1%K*Q2%RE
  END FUNCTION QPROD
  !
  FUNCTION TSIN(Q) RESULT(RES) ! RETURNS TRUNCATED TAYLOR SERIES SINE
    IMPLICIT NONE
    TYPE(QTRN),   INTENT(IN) :: Q
    TYPE(QTRN)               :: RES
    RES%RE = SIN(Q%RE)
    RES%I  = COS(Q%RE)*Q%I
    RES%J  = COS(Q%RE)*Q%J
    RES%K  = COS(Q%RE)*Q%K
  END FUNCTION
  !
  FUNCTION QPOW(Q,N) RESULT(QN) ! RETURNS POWER OF A QUATERNION
    IMPLICIT NONE
    INTEGER, INTENT(IN) :: N
    INTEGER :: I
    TYPE(QTRN), INTENT(IN) :: Q
    TYPE(QTRN) :: QN
    QN%RE = 1.0D0
    QN%I  = 0.0D0
    QN%J  = 0.0D0
    QN%K  = 0.0D0
    DO I=1,N
      QN = QN*Q
    END DO
  END FUNCTION QPOW
END MODULE QUATERNIONS
```

Figure B.3: Implementation of a minimal quaternion number module to be used to compute derivatives of analytic function such as the one presented in section 3.3.

```fortran
PROGRAM SEVEN_VAR_EXAMPLE
    ! ================ DECLARATIONS ==============================
    USE QUATERNIONS
    IMPLICIT NONE
    ! DEFINE STEP SIZE:
    REAL(REAL_KIND) :: H = 1D-15
    ! DEFINE THE REAL COORDINATES
    REAL(REAL_KIND) :: A0=0.1D0, B0=0.2D0, C0=0.3D0, D0=0.4D0,    &
                       X0=0.5D0, Y0=0.6D0, Z0=0.7D0
    ! DEFINE QUATERNION VERSIONS OF VARIABLES
    TYPE(QTRN)      :: A, B, C, D, X, Y, Z
    ! RESULT HOLDERS
    TYPE(QTRN)      :: QF
    REAL(REAL_KIND) ::    F, DFDA, DFDB, DFDC, &
                       DFDD, DFDX, DFDY, DFDZ
    ! ============================================================
    ! PERTURB THE COORDINATES IN QUATERNION IMAGINARY DIRECTIONS,
    A = QTRN( A0,  H , 0.0, 0.0 )
    B = QTRN( B0, 0.0,  H , 0.0 )
    C = QTRN( C0, 0.0, 0.0,  H  )
    D = QTRN( D0, 0.0, 0.0, 0.0 )
    X = QTRN( X0, 0.0, 0.0, 0.0 )
    Y = QTRN( Y0, 0.0, 0.0, 0.0 )
    Z = QTRN( Z0, 0.0, 0.0, 0.0 )
    ! EVALUATE THE SIN FUNCTION USING PERTURBED VARIABLES.
    QF = SIN(A**3 * B**2 * C**3 * D**4 * X**3 * Y**2 * Z**4)
    ! EXTRACT RESULTS
    F    = QF%RE  !
    DFDA = QF%I/H !-
    DFDB = QF%J/H ! |- DERIVATIVES
    DFDC = QF%K/H !-
  ! ===================== SECOND EVALUATION ===================
    A = QTRN( A0, 0.0, 0.0, 0.0 )
    B = QTRN( B0, 0.0, 0.0, 0.0 )
    C = QTRN( C0, 0.0, 0.0, 0.0 )
    D = QTRN( D0,  H , 0.0, 0.0 )!-
    X = QTRN( X0, 0.0,  H , 0.0 )! |- Second perturbations.
    Y = QTRN( Y0, 0.0, 0.0,  H  )!-
    Z = QTRN( Z0, 0.0, 0.0, 0.0 )
    ! EVALUATE THE SIN FUNCTION USING PERTURBED VARIABLES.
    QF = SIN(A**3 * B**2 * C**3 * D**4 * X**3 * Y**2 * Z**4)
    ! EXTRACT RESULTS
    DFDD = QF%I/H !
    DFDX = QF%J/H ! EXTRACT THE RESULTS
    DFDY = QF%K/H !
    ! ===================== THIRD EVALUATION ====================
    A = QTRN( A0, 0.0, 0.0, 0.0 )
    B = QTRN( B0, 0.0, 0.0, 0.0 )
    C = QTRN( C0, 0.0, 0.0, 0.0 )
    D = QTRN( D0, 0.0, 0.0, 0.0 )
    X = QTRN( X0, 0.0, 0.0, 0.0 )
    Y = QTRN( Y0, 0.0, 0.0, 0.0 )
    Z = QTRN( Z0,  H , 0.0, 0.0 )! - Third perturbation.
    ! EVALUATE THE SIN FUNCTION USING PERTURBED VARIABLES.
    QF = SIN(A**3 * B**2 * C**3 * D**4 * X**3 * Y**2 * Z**4)
  ! EXTRACT RESULTS
    DFDZ = QF%I/H !
  ! ================== PRINT RESULTS IN CONSOLE ================
    PRINT*, '== RESULT OF QUATERNION ANALYSIS =='
    PRINT*, 'F:   ',    F, 'DFDA:', DFDA, 'DFDB:', DFDB, 'DFDC:', DFDC
    PRINT*, 'DFDD:', DFDD, 'DFDX:', DFDX, 'DFDY:', DFDY, 'DFDZ:', DFDZ
END PROGRAM SEVEN_VAR_EXAMPLE
```

Figure B.4: Implementation of the quaternion program to compute all derivatives of the numerical example shown in section 3.3.

Appendix C: OCTONIONS

Also known as Cayley numbers [26], octonions are also a subset of hypercomplex numbers. They are considered a doubling of the algebra of quaternions, because the total number of directions (including real direction) is doubled. They extend complex numbers to a total of 7 imaginary units. A number o^* is an octonion when

$$o^* = o_r + o_i i + o_j j + o_k k + o_{e'} e' + o_{i'} i' + o_{j'} j' + o_{k'} k' \tag{C.1}$$

where o_r, o_i, o_j, o_k, $o_{e'}$, $o_{i'}$, $o_{j'}$ and $o_{k'}$ are real coefficients and i, j, k, e', i', j', k' are imaginary units with imaginary conditions summarized by the multiplication table shown in Table C.1. According to Table C.1, the result of multiplying, e.g. k times j' is given in the corresponding k-row and j'-column, i.e. $kj' = i'$.

Left factor \ Right factor	1	i	j	k	e'	i'	j'	k'
1	1	i	j	k	e'	i'	j'	k'
i	i	-1	k	$-j$	i'	$-e'$	$-k'$	j'
j	j	$-k$	-1	i	j'	k'	$-e'$	i'
k	k	j	$-i$	-1	k'	$-j'$	i'	$-e'$
e'	e'	$-i'$	$-j'$	$-k'$	-1	i	j	k
i'	i'	e'	$-k'$	j'	$-i$	-1	$-k$	j
j'	j'	k'	e'	$-i'$	$-j$	k	-1	$-i$
k'	k'	$-j'$	i'	e'	$-k$	$-j$	i	-1

Table C.1: Multiplication table of Octonion imaginary units.

C.1 Algebra

Octonion addition is similar to quaternion addition. Assume two octonion numbers o^* and r^*,

$$o^* = o_r + o_i i + o_j j + o_k k + o_{e'} e' + o_{i'} i' + o_{j'} j' + o_{k'} k' \tag{C.2}$$

$$r^* = r_r + r_i i + r_j j + r_k k + r_{e'} e' + r_{i'} i' + r_{j'} j' + r_{k'} k' \tag{C.3}$$

addition of o^* and r^* results in

$$o^* + r^* = (o_r + r_r) + (o_i + r_i)i + (o_j + r_j)j + (o_k + r_k)k +$$
$$(o_{e'} + r_{e'})e' + (o_{i'} + r_{i'})i' + (o_{j'} + r_{j'})j' + (o_{k'} + r_{k'})k' \quad \text{(C.4)}$$

Multiplication of octonions is performed as follows:

$$\begin{aligned} o^*r^* = &(o_r r_r - o_i r_i - o_j r_j - o_k r_k - o_{e'} r_{e'} - o_{i'} r_{i'} - o_{j'} r_{j'} - o_{k'} r_{k'}) + \\ &(o_r r_i + o_i r_r - o_j r_k + o_k r_j - o_{e'} r_{i'} + o_{i'} r_{e'} + o_{j'} r_{k'} - o_{k'} r_{j'})i + \\ &(o_r r_j + o_i r_k + o_j r_r - o_k r_i - o_{e'} r_{j'} - o_{i'} r_{k'} + o_{j'} r_{e'} + o_{k'} r_{i'})j + \\ &(o_r r_k - o_i r_j + o_j r_i + o_k r_r - o_{e'} r_{k'} + o_{i'} r_{j'} - o_{j'} r_{i'} + o_{k'} r_{e'})k + \\ &(o_r r_{e'} + o_i r_{i'} + o_j r_{j'} + o_k r_{k'} + o_{e'} r_r - o_{i'} r_i - o_{j'} r_j - o_{k'} r_k)e' + \\ &(o_r r_{i'} - o_i r_{e'} + o_j r_{k'} - o_k r_{j'} + o_{e'} r_i + o_{i'} r_r + o_{j'} r_k - o_{k'} r_j)i' + \\ &(o_r r_{j'} - o_i r_{k'} - o_j r_{e'} + o_k r_{i'} + o_{e'} r_j - o_{i'} r_k + o_{j'} r_r + o_{k'} r_i)j' + \\ &(o_r r_{k'} + o_i r_{j'} - o_j r_{i'} - o_k r_{e'} + o_{e'} r_k + o_{i'} r_j - o_{j'} r_i + o_{k'} r_r)k' \end{aligned} \quad \text{(C.5)}$$

Multiplication by a scalar real number α results in

$$o^*\alpha = \alpha o^* = \alpha o_r + \alpha o_i i + \alpha o_j j + \alpha o_k k + \alpha o_{e'} e' + \alpha o_{i'} i' + \alpha o_{j'} j' + \alpha o_{k'} k' \quad \text{(C.6)}$$

Besides being non-commutative, octonion multiplication is non-associative. Consider the triplet ije', where the result varies according to the order of associative operations:

$$(ij)e' = ke' = k' \quad \text{(C.7)}$$

meanwhile

$$i(je') = ij' = -k' \quad \text{(C.8)}$$

$$\therefore (ij)e' \neq i(je') \quad \text{(C.9)}$$

The conjugate $\bar{o^*}$ of an octonion number o^* is

$$\bar{o^*} = o_0 - o_i i - o_j j - o_k k - o_{e'} e' - o_{i'} i' - o_{j'} j' - o_{k'} k' \tag{C.10}$$

such that multiplication $\bar{o^*} o^*$ eliminates all imaginary directions

$$\bar{o^*} o^* = o^* \bar{o^*} = o_r^2 + o_i^2 + o_j^2 + o_k^2 + o_{e'}^2 + o_{i'}^2 + o_{j'}^2 + o_{k'}^2 \tag{C.11}$$

Further algebraic operations are described in section 3.4.

C.2 Matrix and Vector Representations

Matrix representation of an octonion number (see [90]) is given by

$$\mathbf{T}(o^*) = \begin{bmatrix} o_r & -o_i & -o_j & -o_k & -o_{e'} & -o_{i'} & -o_{j'} & -o_{k'} \\ o_i & o_r & -o_k & o_j & -o_{i'} & o_{e'} & o_{k'} & -o_{j'} \\ o_j & o_k & o_r & -o_i & -o_{j'} & -o_{k'} & o_{e'} & o_{i'} \\ o_k & -o_j & o_i & o_r & -o_{k'} & o_{j'} & -o_{i'} & o_{e'} \\ o_{e'} & o_{i'} & o_{j'} & o_{k'} & o_r & -o_i & -o_j & -o_k \\ o_{i'} & -o_{e'} & o_{k'} & -o_{j'} & o_i & o_r & o_k & -o_j \\ o_{j'} & -o_{k'} & -o_{e'} & o_{i'} & o_j & -o_k & o_r & o_i \\ o_{k'} & o_{j'} & -o_{i'} & -o_{e'} & o_k & o_j & -o_i & o_r \end{bmatrix} \tag{C.12}$$

and its vector representation is

$$\mathbf{t}(o^*) = \begin{bmatrix} o_r & o_i & o_j & o_k & o_{e'} & o_{i'} & o_{j'} & o_{k'} \end{bmatrix}^T \tag{C.13}$$

Addition and multiplication of octonions can be performed using vector and matrix forms analogously to quaternions, see Equations (B.19) and (B.20).

Appendix D: WEAK FORMS OF SOME PARTIAL DIFFERENTIAL EQUATIONS

The purpose of this appendix is to define the weak forms of various partial differential equations used in this document. For this, a complementary nomenclature list is defined as a guide to the reader.

Nomenclature

$\bar{(\cdot)}$ Prescribed value.

Ω Domain

$\Gamma_{\mathbf{D}}$ Boundary of Ω with Dirichlet conditions.

$\Gamma_{\mathbf{N}}$ Boundary of Ω with Neumann conditions.

$\Gamma_{\mathbf{R}}$ Boundary of Ω with Robin conditions.

$\nabla(\cdot)$ Gradient operator, i.e. $[(\cdot)_{,x}, (\cdot)_{,y}, (\cdot)_{,z}]^T$

$\nabla \cdot (\cdot)$ Divergence operator, i.e. $(\cdot)_{,x} + (\cdot)_{,y} + (\cdot)_{,z}$

$\Delta(\cdot)$ Laplacian operator, i.e. $\nabla \cdot \nabla(\cdot) = \big((\cdot)_{,x^2} + (\cdot)_{,y^2} + (\cdot)_{,z^2}\big)$

$L^2(\Omega)$ Function space $\left\{ w \,\middle|\, \int_\Omega |w|^2 \, dV < \infty \right\}$

$H^1(\Omega)$ Sobolev function space $\left\{ w \in L^2(\Omega) \,\middle|\, \int_\Omega |\nabla w|^2 \, dV < \infty \right\}$

$H^1_0(\Omega)$ Function space $\left\{ w \in H^1(\Omega) \,\middle|\, w = 0 \text{ on } \Gamma_{\mathrm{D}} \right\}$

D.1 Laplace equation

The Laplace boundary value problem [83] is defined as follows,

$$-\Delta u = 0 \qquad \text{in } \Omega \tag{D.1}$$

$$u = \bar{u} \qquad \text{on } \Gamma_{\mathrm{D}} \tag{D.2}$$

$$\nabla u \cdot \mathbf{n} = \bar{t} \qquad \text{on } \Gamma_{\mathrm{N}} \tag{D.3}$$

where u is the state function, Δu is the laplacian of u; Ω is the domain of the problem and $\mathbf{n}$ is the unit normal vector. The boundary conditions are defined with the prescribed value of the state function on Γ_{D} and the prescribed normal derivative $\bar{t}$ on Γ_{N}.

Table D.1 summarizes the input parameters that define the Laplace boundary value problem.

Input parameter	**Variable** ϕ
Geometry	(x, y, z)
Prescribed state function value	$\bar{u}$
Prescribed normal derivative	$\bar{t}$

Table D.1: Input variables that define the Laplace boundary value problem.

The weak form of the equation is

$$\int_{\Omega} \nabla u \cdot \nabla v \, dV - \int_{\Gamma_{\mathrm{N}}} (\nabla u \cdot \mathbf{n}) \, v \, dS = 0 \tag{D.4}$$

where $v \in H_0^1(\Omega)$ is an arbitrary test function, and the conditions for solving the problem have been weakened to $u \in H^1(\Omega)$.

D.2 Heat Conduction for Isotropic Homogeneous Materials

The conservation of thermal energy for steady-state conditions [83] can be expressed for isotropic homogeneous materials as

$$\begin{aligned} -\alpha \Delta T &= s && \text{in } \Omega \\ T &= \bar{T} && \text{on } \Gamma_{\mathrm{D}} \\ -\alpha \nabla T \cdot \mathbf{n} &= \bar{b} && \text{on } \Gamma_{\mathrm{N}} \\ -\alpha \nabla T \cdot \mathbf{n} &= h_c (T - T_\infty) && \text{on } \Gamma_{\mathrm{R}} \end{aligned} \tag{D.5}$$

where Ω is the region over which the problem is defined, T is the temperature field in Ω, α is the thermal conductivity, ∇ is the gradient operator and s is the inner heat generation rate per unit volume. The boundary $\partial\Omega$ is decomposed into three regions: Γ_{D} represents the region with essential (Dirichlet) conditions where the temperature $\bar{T}$ is prescribed; Γ_{N} represents the region with natural (Neumann) conditions where the normal heat flux $\bar{b}$ is prescribed; and Γ_{R} represents the region with mixed (Robin) conditions where convection is developed, hence h_c is the convection coefficient, T is the unknown temperature at the solid/fluid interface Γ_{R}, and T_∞ is the fluid temperature. $\mathbf{n}$ is the unit normal vector to the corresponding boundary.

Table D.2 summarizes the input parameters that define the stationary heat transfer problem for isotropic materials.

Input parameter	**Variable ϕ**
Geometry	(x, y, z)
Thermal conductivity	α
Heat generation rate	s
Prescribed temperature	$\bar{T}$
Prescribed heat flux	$\bar{b}$
Convection coefficient	h_c
Fluid temperature	T_∞

Table D.2: Input variables that define the heat transfer problem for isotropic materials.

The weak form of this problem is given by the following expression

$$\int_{\Omega} \nabla T \cdot \nabla v \, dV - \int_{\Gamma_{\mathrm{N}}} \bar{b} \, v \, dS - \int_{\Gamma_{\mathrm{R}}} \left(h_c(T - T_\infty)\right) v \, dS - \int_{\Omega} s \, v \, dV = 0 \tag{D.6}$$

where $v \in H_0^1(\Omega)$ is an arbitrary test function, and the conditions for solving the problem have been weakened to $T \in H^1(\Omega)$.

D.3 System of Elasticity for Solid Isotropic Materials

The static equilibrium of linear elastic solids [83] is given by the differential equation

$$\nabla \cdot \boldsymbol{\sigma}(\mathbf{u}) = \mathbf{b} \quad \text{in } \Omega \tag{D.7}$$

$$\mathbf{u} = \bar{\mathbf{u}} \quad \text{on } \Gamma_{\mathrm{D}} \tag{D.8}$$

$$\boldsymbol{\sigma}(\mathbf{u})\mathbf{n} = \bar{\mathbf{t}} \quad \text{on } \Gamma_{\mathrm{N}} \tag{D.9}$$

where $\mathbf{b}$ is a body force, Ω is the domain, $\mathbf{n}$ is the unit normal to the boundary and $\boldsymbol{\sigma}(\mathbf{u})$ is the stress tensor that directly depends on the deformation vector field $\mathbf{u} = [u_x, u_y, u_z]$. Boundary conditions are defined as Dirichlet (essential) with the prescribed displacements $\bar{\mathbf{u}}$ on Γ_D; and Neumann (natural) with the prescribed forces $\bar{\mathbf{t}}$ at the boundary Γ_N. The stress is defined in terms of the strain tensor $\boldsymbol{\varepsilon}(\mathbf{u})$ using the constitutive equation

$$\boldsymbol{\sigma}(\mathbf{u}) = \lambda(\nabla \cdot \mathbf{u})\mathbf{I} + 2\mu\boldsymbol{\varepsilon}(\mathbf{u}) \tag{D.10}$$

where $\mathbf{I}$ is the identity matrix, λ and μ are the Lamé constants that depend on the Young's modulus E and Poisson's ratio ν as

$$\lambda = \frac{\nu E}{(1+\nu)(1-2\nu)}, \quad \mu = \frac{E}{2(1+\nu)} \tag{D.11}$$

The divergence of the deformation vector field, $\nabla \cdot \mathbf{u}$, is found in other textbooks as Tr($\boldsymbol{\varepsilon}(u)$), where Tr($\cdot$) is the trace, i.e. the sum of the diagonal elements of the tensor. Finally, the linear kinematic law quantifies the relationship of the strain tensor and deformation

$$\boldsymbol{\varepsilon}(\mathbf{u}) = \frac{1}{2}\left(\nabla\mathbf{u} + \nabla\mathbf{u}^T\right) \tag{D.12}$$

Table D.3 summarizes the input parameters that define the static equilibrium linear elastic problem for isotropic materials.

Input parameter	**Variable** ϕ
Geometry	(x, y, z)
Elastic modulus	E
Poisson ratio	ν
Body force	$\mathbf{b}$
Surface displacements	$\bar{\mathbf{u}}$
Surface traction	$\bar{\mathbf{t}}$

Table D.3: Input variables that define the linear elastic problem for isotropic materials.

The weak form of the problem is given by

$$\int_\Omega \boldsymbol{\sigma}(\mathbf{u}) : \boldsymbol{\varepsilon}(\mathbf{v})\, dV - \int_{\Gamma_\mathrm{N}} \bar{\mathbf{t}} \cdot \mathbf{v}\, dS - \int_\Omega \mathbf{b} \cdot \mathbf{v}\, dV = 0 \tag{D.13}$$

where $\mathbf{v} = [v_x, v_y, v_z]$ is a vector function where every element $v_x, v_y, v_z \in H_0^1(\Omega)$ are arbitrary test functions, and the conditions for solving the problem have been weakened to $u_x, u_y, u_z \in H^1(\Omega)$. The weak form can be expanded using the constitutive equation (D.10) as follows

$$\int_\Omega \lambda(\nabla \cdot \mathbf{u})(\nabla \cdot \mathbf{v}) + 2\mu\boldsymbol{\varepsilon}(\mathbf{u}) : \boldsymbol{\varepsilon}(\mathbf{v})\, dV - \int_{\Gamma_\mathrm{N}} \bar{\mathbf{t}} \cdot \mathbf{v}\, dS - \int_\Omega \mathbf{b} \cdot \mathbf{v}\, dV = 0 \tag{D.14}$$

D.4 Stokes System for Stationary Incompressible Fluids

The Stokes system for stationary incompressible fluids neglects the effects of inertial forces with respect to the contribution of viscous forces [88]. The boundary value problem given by the following equations:

$$-\mu\Delta\mathbf{u} + \nabla p = 0 \quad \text{in } \Omega \tag{D.15}$$

$$\nabla \cdot \mathbf{u} = 0 \quad \text{in } \Omega \tag{D.16}$$

$$\mathbf{u} = \bar{\mathbf{u}} \quad \text{on } \Gamma_{\mathrm{D}_\mathbf{u}} \tag{D.17}$$

$$p = \bar{p} \quad \text{on } \Gamma_{\mathrm{D}\,p} \tag{D.18}$$

where $\mathbf{u} = [u_x, u_y, u_z]$ is the fluid velocity field, μ is the fluid viscosity and p is its pressure. Boundary conditions are defined at the Dirichlet region with the prescribed velocity $\bar{\mathbf{u}}$, $\Gamma_{\mathbf{D_u}}$; and at the region with the prescribed pressure $\bar{p}$, $\Gamma_{\mathbf{D}_p}$. The Stokes equation is valid for those flows with small Reynolds numbers, i.e. Re $\to 0$ (Re $= \rho\,|\mathbf{u}|\,L/\mu$, where ρ is the density of the fluid and L is a characteristic length).

Table D.4 summarizes the input parameters that define the stationary Stokes system for fluid problems.

Input parameter	**Variable ϕ**
Geometry	(x, y, z)
Fluid viscosity	μ
Prescribed velocity	$\bar{\mathbf{u}}$
Prescribed pressure	$\bar{p}$

Table D.4: Input variables that define the stokes problem.

The weak form of the problem is as follows

$$\int_\Omega \mu\nabla\mathbf{u} : \nabla\mathbf{v}\; dV - \int_\Omega \nabla p \cdot \mathbf{v}\; dV + \int_\Omega \nabla \cdot \mathbf{u}\, q\; dV - \int_\Omega \beta pq\; dV = 0 \tag{D.19}$$

where $\mathbf{v} = [v_x, v_y, v_z]$ is a vector function where every element $v_x, v_y, v_z \in H_0^1(\Omega)$ are arbitrary test functions, and the conditions for solving the problem have been weakened to $u_x, u_y, u_z \in H^1(\Omega)$. The factor β, usually $\beta = 10^{-10}$, is a numerical stabilization term for the global stiffness matrix used in Finite Element formulations [87].

Appendix E: SUPPORTED ELEMENTS IN PYOTI FEM LIBRARY

In the current state of the library, 1D and 2D elements are supported. The geometric types of supported finite elements are: line, triangle and quadrangle, tetrahedra and hexahedra. Table E.1, shows the element types supported in the current version of the library.

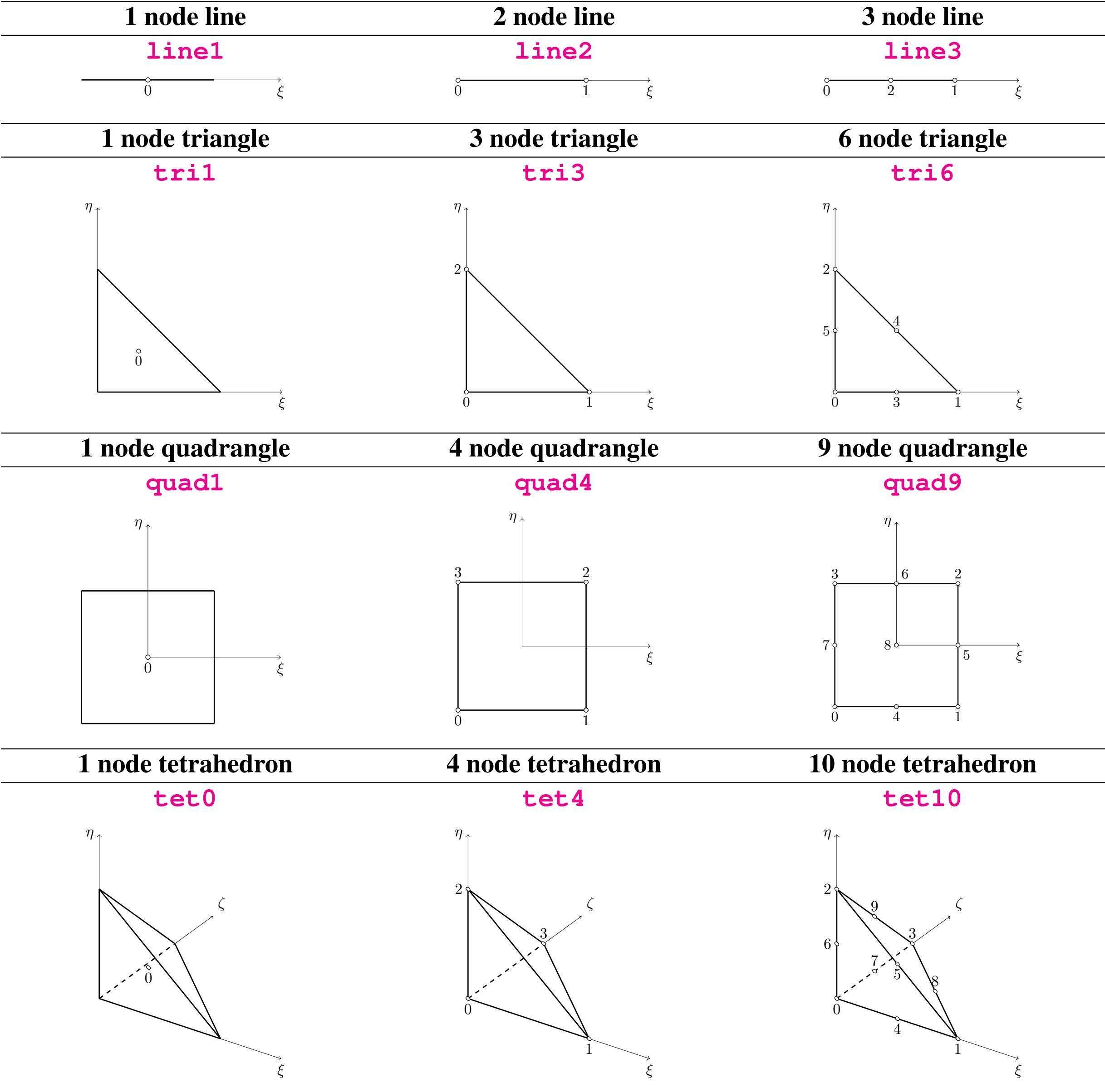


Table E.1: Standard finite elements supported in pyOTI finite element module.

E.1 Serendipity elements.

The serendipity elements are shown in Tables E.2

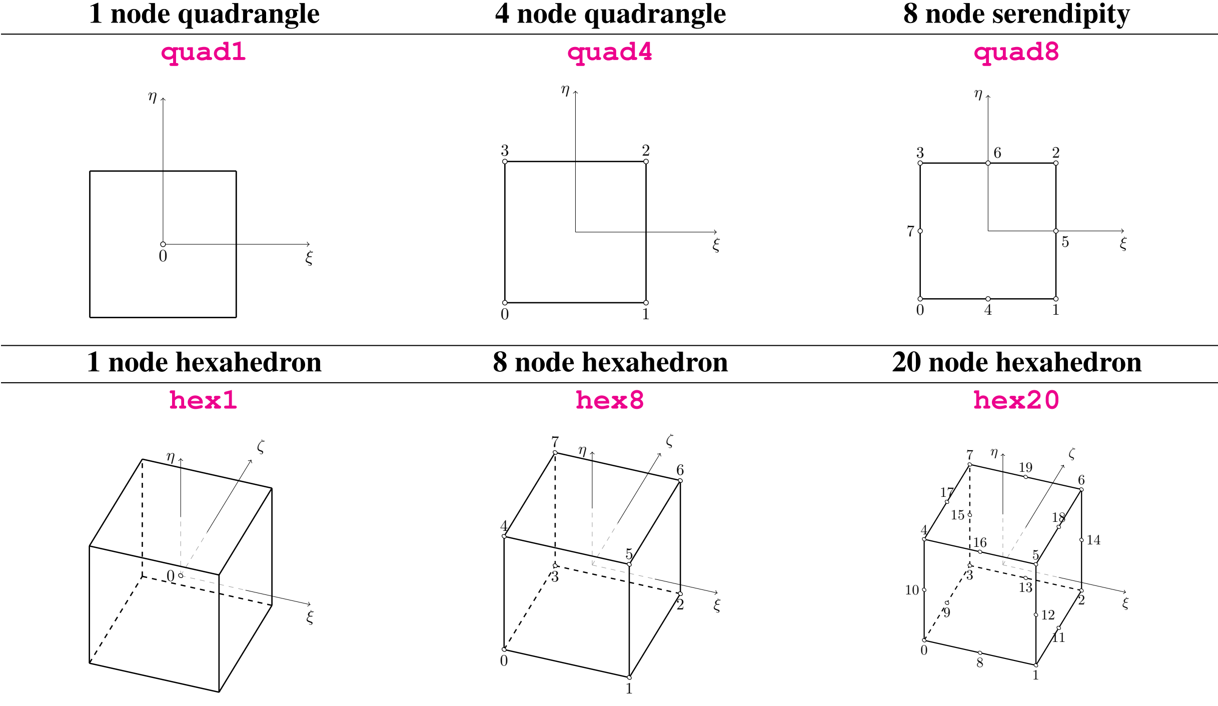


Table E.2: Serendipity elements supported in pyOTI finite element module.

Appendix F: EXAMPLE PYOTI FEM SOURCE CODES

F.1 1D Example

```
1  from pyoti.fem import mesh, fespace, feproblem, on, int1d
2  from pyoti.sparse import sin, e
3
4  order = 2  # Truncation order 2.
5
6  Th = mesh("line.msh",otiorder = order)
7
8  Vh = fespace( Th, 1)
9
10 x = Th.x # Create an alias for x-coordinates of the mesh.
11
12 # Perturbation for Eulerian derivative
13 x[1,0] = x[1,0] + e(1, order = order)
14
15 # Perturbation for Lagrangian derivative.
16 for i in range(x.shape[0]):
17     xi = x[i,0]
18     x[i,0] = xi + xi.real * e(2, order = order)
19 # end for
20
21 # Pertbation for gradient.
22 for i in range(2,x.shape[0]):
23     xi = x[i,0]
24     x[i,0] = xi + e(3, order = order)
25 # end for
26
27 # Create State, test functions and independent function.
28 u = Vh()
29 v = Vh()
30
31 f = Vh( sin(x) )
32
33 problem = feproblem([u],[v],
34     int1d( dx(u)*dx(v) ) +
35     int1d( u*v)          +
36     int1d(-f*v)          +
37     on('left',  u, 0)    +
38     on('right', u, 0)        )
39
40 problem.solve()
41
42 Th.export("out.vtk",pd=[u],pdNames=['u'])
```

Figure F.1: Program that implements a 1D example.

F.2 2D cylinder

```
# Import library
from pyoti.fem    import mesh, fespace, feproblem
from pyoti.sparse import e

# Import mesh.
Th = mesh( "cylinder.msh")

# Define Finite Element space for order 2 elements.
Vh = fespace( Th, 2)

# State functions.
ux = Vh()
uy = Vh()

# Test functions.
vx = Vh()
vy = Vh()

E  =  21e5 + e(1, order = 2)  # Define Elasticity parameters
nu =  0.28                     # with corresponding perturbations.

mu     =  E/(2*(1+nu))           # Define Lamé constants.
lambda =  E*nu/((1+nu)*(1-2*nu)) #

equation = (
    int2d( lambda*dx(ux)*dx(vx) + lambda*dx(ux)*dy(vy) ) +
    int2d( lambda*dy(uy)*dx(vx) + lambda*dy(uy)*dy(vy) ) +
    int2d(   2*mu*dx(ux)*dx(vx) +   2*mu*dy(uy)*dy(vy) ) +
    int2d(     mu*dy(ux)*dy(vx) +     mu*dy(ux)*dx(vy) ) +
    int2d(     mu*dx(uy)*dy(vx) +     mu*dx(uy)*dx(vy) ) +
    on( 'left', ux, 0) +on( 'left', uy, 0)                )

problem = feproblem([ ux, uy],[ vx, vy],   equation )

# Solve problem
problem.solve()

Th.export("beam.vtk", pd=[[ux,uy]], pdNames=['u'])
```

Figure F.2: Program that implements a 2D Elasticity analysis of a cylinder and computes second order derivatives with respect to the modulus of elasticity.

F.3 2D Cavity

```
1  # Import pyoti.
2  from pyoti.fem import int2d, on, fespace, feproblem, square
3  from pyoti.sparse import e
4
5  # Generate a triangulation of a 1x1 square mesh with 6 noded triangles
6  Th = square(1,1,he = 0.01, element_order=2)
7
8  # Mesh is centered at 0,0. Move it so that left bottom is (0,0)
9  Th.x  += 0.5; Th.y  += 0.5
10
11 # Perturbation of domain coordinates.
12 Th.x = Th.x + Th.x * e(1, order = 30)
13 Th.y = Th.y + Th.y * e(2, order = 30)
14
15 # Define the finite element spaces for velocity Vh and pressure Qh.
16 Vh = fespace( Th, 2)
17 Qh = fespace( Th, 1)
18
19 # State functions.
20 ux = Vh(); uy = Vh(); p  = Qh()
21
22 # Test functions.
23 vx = Vh(); vy = Vh(); q  = Qh()
24
25 # Define viscosity
26 mu = 1
27
28 # Define Variational equation
29 equation = (
30   int2d( mu*dx(ux)*dx(vx) + mu*dy(ux)*dy(vx)  ) +
31     int2d( mu*dx(uy)*dx(vy) + mu*dy(uy)*dy(vy)  ) +
32     int2d( dx(p)*vx         + dy(p)*vy          ) +
33     int2d( dx(ux)*q         + dy(uy)*q          ) +
34     int2d( (-1e-10)*p*q                      ) +
35     on('left',  ux, 0) + on('left',  uy, 0) +
36     on('bottom',ux, 0) + on('bottom',uy, 0) +
37     on('right', ux, 0) + on('right', uy, 0) +
38     on('top',   ux, 1) + on('top',   uy, 0)    )
39
40 problem = feproblem([ ux, uy, p ],[ vx, vy, q ] , equation )
41
42 problem.solve(solver = 'umfpack')
43
44 ux_prime = ux.rom_eval([ 1, 2],[ 0.0, 0.7] )
45 uy_prime = uy.rom_eval([ 1, 2],[ 0.0, 0.7] )
46 p_prime  =  p.rom_eval([ 1, 2],[ 0.0, 0.7] )
```

Figure F.3: Program that implements a 2D Stokes analysis of the cavity problem and computes the 30th order OTIROM of the velocity and pressure fields at $L_x' = 0$, $L_y' = 1.7$.

BIBLIOGRAPHY

[1] M. Kleiber, T. D. Hien, H. Antúnez, P. Kowalczyk, Parameter sensitivity in nonlinear mechanics: Theory and finite element computations, Wiley, 1997.

[2] K. K. Choi, N.-H. Kim, Structural sensitivity analysis and optimization 1: linear systems, Springer Science & Business Media, 2006.

[3] A. Griewank, A. Walther, Evaluating derivatives: principles and techniques of algorithmic differentiation, Vol. 105, Siam, 2008.

[4] V. Pascual, L. Hascoët, Mixed-language automatic differentiation, Optimization Methods and Software (2018) 1–15.

[5] U. Naumann, The art of differentiating computer programs: an introduction to algorithmic differentiation, Vol. 24, Siam, 2012.

[6] D. Izzo, F. Biscani, A. Mereta, Differentiable genetic programming, in: European Conference on Genetic Programming, Springer, 2017, pp. 35–51.

[7] G. Lantoine, R. P. Russell, T. Dargent, Using multicomplex variables for automatic computation of high-order derivatives, ACM Transactions on Mathematical Software (TOMS) 38 (3) (2012) 16.

[8] J. A. Fike, J. J. Alonso, The development of hyper-dual numbers for exact second-derivative calculations, AIAA paper 886 (2011) 124.

[9] W. Yu, M. Blair, Dnad, a simple tool for automatic differentiation of fortran codes using dual numbers, Computer Physics Communications 184 (5) (2013) 1446–1452.

[10] W. Jin, B. H. Dennis, B. P. Wang, Improved sensitivity analysis using a complex variable semi-analytical method, Structural and Multidisciplinary Optimization 41 (3) (2010) 433–439.

[11] F. Fernandez, D. A. Tortorelli, Semi-analytical sensitivity analysis for nonlinear transient problems, Structural and Multidisciplinary Optimization 58 (6) (2018) 2387–2410.

[12] L. Hascoet, V. Pascual, The tapenade automatic differentiation tool: principles, model, and specification, ACM Transactions on Mathematical Software (TOMS) 39 (3) (2013) 20.

[13] U. Naumann, J. Lotz, K. Leppkes, M. Towara, Algorithmic differentiation of numerical methods: Tangent and adjoint solvers for parameterized systems of nonlinear equations, ACM Transactions on Mathematical Software (TOMS) 41 (4) (2015) 26.

[14] E. T. Phipps, D. M. Gay, Automatic differentation of c++ codes with sacado., Tech. rep., Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States) (2006).

[15] A. Walther, A. Griewank, Getting started with adol-c., in: Combinatorial scientific computing, 2009, pp. 181–202.

[16] I. Charpentier, J. Gustedt, Arbogast: Higher order automatic differentiation for special functions with modular c, Optimization Methods and Software (2018) 1–25.

[17] F. Biscani, Multiplication of sparse laurent polynomials and poisson series on modern hardware architectures, arXiv preprint arXiv:1004.4548.

[18] F. Biscani, Parallel sparse polynomial multiplication on modern hardware architectures, in: Proceedings of the 37th International Symposium on Symbolic and Algebraic Computation, ACM, 2012, pp. 83–90.

[19] B. Z. Dunham, High-order automatic differentiation of unmodified linear algebra routines via nilpotent matrices, Ph.D. thesis, University of Colorado at Boulder (2017).

[20] Y. Gao, Y. Wu, J. Xia, Automatic differentiation based discrete adjoint method for aerodynamic design optimization on unstructured meshes, Chinese Journal of Aeronautics 30 (2) (2017) 611–627.

[21] B. Mohammadi, O. Pironneau, Mesh adaption and automatic differentiation in a cad-free framework for optimal shape design, International Journal for Numerical Methods in Fluids 30 (2) (1999) 127–136.

[22] B. Mohammadi, B. Mohammadi, Optimal shape design, reverse mode of automatic differentiation and turbulence, in: 35th Aerospace Sciences Meeting and Exhibit, 1997, p. 99.

[23] P. Hovland, B. Mohammadi, C. Bischof, Automatic differentiation and navier-stokes computations, in: Computational Methods for Optimal Design and Control, Springer, 1998, pp. 265–284.

[24] J. R. Martins, P. Sturdza, J. J. Alonso, The complex-step derivative approximation, ACM Transactions on Mathematical Software (TOMS) 29 (3) (2003) 245–262.

[25] W. Squire, G. Trapp, Using complex variables to estimate derivatives of real functions, Siam Review 40 (1) (1998) 110–112.

[26] I. L. Kantor, A. S. Solodovnikov, Hypercomplex numbers: an elementary introduction to algebras, Springer, 1989.

[27] R. E. Spall, W. Yu, Imbedded dual-number automatic differentiation for computational fluid dynamics sensitivity analysis, Journal of Fluids Engineering 135 (1) (2013) 014501.

[28] J. Conway, D. Smith, On Quaternions and Octonions: Their Geometry, Arithmetic, and Symmetry., AK Peters, 2003.

[29] J. Turner, Quaternion-based partial derivative and state transition matrix calculations for design optimization, in: 40th AIAA Aerospace Sciences Meeting & Exhibit, 2002, p. 448.

[30] G. B. Price, An introduction to multicomplex spaces and functions, M. Dekker, 1991.

[31] J. F. Monsalvo, M. J. García, H. Millwater, Y. Feng, Sensitivity analysis for radiofrequency induced thermal therapies using the complex finite element method, Finite Elements in Analysis and Design 135 (2017) 11–21.

[32] J. Grisham, A. Akbariyeh, W. Jin, B. H. Dennis, B. P. Wang, Application of the semi-analytic complex variable method to computing sensitivities in heat conduction, Journal of Heat Transfer 140 (8) (2018) 082006.

[33] J. Garza, H. Millwater, Multicomplex newmark-beta time integration method for sensitivity analysis in structural dynamics, AIAA Journal 53 (5) (2015) 1188–1198.

[34] A. Montoya, H. Millwater, Sensitivity analysis in thermoelastic problems using the complex finite element method, Journal of Thermal Stresses 40 (3) (2017) 302–321.

[35] A. Voorhees, H. Millwater, R. Bagley, Complex variable methods for shape sensitivity of finite element models, Finite elements in analysis and design 47 (10) (2011) 1146–1156.

[36] A. Montoya, R. Fielder, A. Gomez-Farias, H. Millwater, Finite-element sensitivity for plasticity using complex variable methods, Journal of Engineering Mechanics 141 (2) (2014) 04014118.

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

[38] J. A. Fike, Multi-objective optimization using hyper-dual numbers, Ph.D. thesis, PhD thesis, Stanford university (2013).

[39] D. L. Whitfield, J. C. Newman, W. K. Anderson, Step-size independent approach for multidisciplinary sensitivity analysis, Journal of aircraft 40 (3) (2003) 566–573. doi:https://doi.org/10.2514/2.3131.

[40] C. Burg, J. N. III, Computationally efficient, numerically exact design space derivatives via the complex taylor's series expansion method, Computers & Fluids 32 (3) (2003) 373 – 383. doi:https://doi.org/10.1016/S0045-7930(01)00044-5.
URL http://www.sciencedirect.com/science/article/pii/S0045793001000445

[41] J. Kim, D. G. Bates, I. Postlethwaite, Nonlinear robust performance analysis using complex-step gradient approximation, Automatica 42 (1) (2006) 177 – 182. doi:https://doi.org/10.1016/j.automatica.2005.09.008.
URL http://www.sciencedirect.com/science/article/pii/S0005109805003237

[42] L. I. Cerviño, T. R. Bewley, On the extension of the complex-step derivative technique to pseudospectral algorithms, Journal of Computational Physics 187 (2) (2003) 544 – 549. doi:https://doi.org/10.1016/S0021-9991(03)00123-2.
URL http://www.sciencedirect.com/science/article/pii/S0021999103001232

[43] B. P. Wang, A. P. Apte, Complex variable method for eigensolution sensitivity analysis, AIAA journal 44 (12) (2006) 2958–2961. doi:https://doi.org/10.2514/1.19225.

[44] H. Millwater, D. Wagner, A. Baines, A. Montoya, A virtual crack extension method to compute energy release rates using a complex variable finite element method, Engineering Fracture Mechanics 162 (2016) 95–111.

[45] A. Montoya, D. Ramirez-Tamayo, H. Millwater, M. Kirby, A complex-variable virtual crack extension finite element method for elastic-plastic fracture mechanics, Engineering Fracture Mechanicsdoi:https://doi.org/10.1016/j.engfracmech.2018.09.023.
URL http://www.sciencedirect.com/science/article/pii/S0013794418306775

[46] D. Wagner, A finite element-based adaptive energy response function method for curvilinear progressive fracture, Ph.D. thesis, The University of Texas at San Antonio, ProQuest Dissertations Publishing, publication number 10845946, document ID 2091375474 (2018).

[47] D. Wagner, M. J. Garcia, A. Montoya, H. Millwater, A finite element-based adaptive energy response function method for 2d curvilinear progressive fracture, International Journal of Fatigue 127 (2019) 229–245.

[48] M. R. Brake, J. A. Fike, S. D. Topping, Parameterized reduced order models from a single mesh using hyper-dual numbers, Journal of Computational and Applied Mathematics 371 (2016) 370–392.

[49] F. Fujii, M. Tanaka, T. Sasagawa, R. Omote, Computational two-mode asymptotic bifurcation theory combined with hyper dual numbers and applied to plate/shell buckling, Computer Methods in Applied Mechanics and Engineering 325 (2017) 666–688.

[50] D. Ramirez-Tamayo, A. Montoya, H. R. Millwater, A new complex-valued thermal fracture approach for evaluating the structural integrity of aircraft structures, in: 2018 AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, 2018, p. 0649.

[51] A. M. Aguirre-Mesa, D. Ramirez-Tamayo, M. J. Garcia, A. Montoya, H. Millwater, A stiffness derivative local hypercomplex-variable finite element method for computing the energy release rate, Engineering Fracture Mechanics 218 (2019) 106581.

[52] Y. Imoto, N. Yamanaka, T. Uramoto, M. Tanaka, M. Fujikawa, N. Mitsume, Fundamental theorem of matrix representations of hyper-dual numbers for computing higher-order derivatives, JSIAM Letters 12 (2020) 29–32.

[53] A. M. Aguirre-Mesa, M. J. Garcia, H. Millwater, Multiz: A library for computation of high-order derivatives using multicomplex or multidual numbers, ACM Transactions on Mathematical Software (TOMS) 46 (3) (2020) 1–30.

[54] M. Aristizabal, D. Ramirez-Tamayo, M. Garcia, A. Aguirre-Mesa, A. Montoya, H. Millwater, Quaternion and octonion-based finite element analysis methods for computing multiple first order derivatives, Journal of Computational Physics 397 (2019) 108831.

[55] W. R. Hamilton, Xi. on quaternions; or on a new system of imaginaries in algebra, The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science 33 (219) (1848) 58–60.

[56] W. R. Hamilton, Researches respecting quaternions: first series, Transactions of the Royal Irish Academy 21 (part 1) (1848) 199–296.

[57] J. Vince, Quaternions for computer graphics, Springer Science & Business Media, 2011.

[58] M. Geradin, A. Cardona, Kinematics and dynamics of rigid and flexible mechanisms using finite elements and quaternion algebra, Computational Mechanics 4 (2) (1988) 115–135.

[59] I. Romero, The interpolation of rotations and its application to finite element models of geometrically exact rods, Computational mechanics 34 (2) (2004) 121–133.

[60] H. Zhong, R. Zhang, N. Xiao, A quaternion-based weak form quadrature element formulation for spatial geometrically exact beams., Archive of Applied Mechanics 84 (12).

[61] J. R. R. A. Martins, P. Sturdza, J. J. Alonso, The complex-step derivative approximation, ACM Trans. Math. Softw. 29 (3) (2003) 245–262. doi:10.1145/838250.838251.
URL http://doi.acm.org/10.1145/838250.838251

[62] J. Baez, The octonions, Bulletin of the American Mathematical Society 39 (2) (2002) 145–205.

[63] K. Imaeda, M. Imaeda, Sedenions: algebra and analysis, Applied mathematics and computation 115 (2-3) (2000) 77–88.

[64] D. Ramirez-Tamayo, A. Montoya, H. Millwater, A virtual crack extension method for thermoelastic fracture using a complex-variable finite element method, Engineering Fracture Mechanics.

[65] A. Aziz, M. Torabi, Thermal stresses in a hollow cylinder with convective boundary conditions on the inside and outside surfaces, Journal of thermal stresses 36 (10) (2013) 1096–1111.

[66] H. R. Millwater, S. Shirinkam, Multicomplex taylor series expansion for computing high order derivatives, International Journal of Applied Mathematics 27 (4) (2014) 311–334.

[67] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, K. Smith, Cython: The best of both worlds, Computing in Science & Engineering 13 (2) (2011) 31–39.

[68] Python, Python vers. 3.7.4, Computer software (2020).
URL https://www.python.org

[69] Anaconda, Anaconda software distribution vers. 2019.10, Computer software (2019).
URL https://anaconda.com

[70] E. Jones, T. Oliphant, P. Peterson, {SciPy}: open source scientific tools for {Python}.

[71] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, v. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, A. Scopatz, Sympy: symbolic computing in python, PeerJ Computer Science 3 (2017) e103. doi:10.7717/peerj-cs.103.
URL https://doi.org/10.7717/peerj-cs.103

[72] S. Van Der Walt, S. C. Colbert, G. Varoquaux, The numpy array: a structure for efficient numerical computation, Computing in Science & Engineering 13 (2) (2011) 22.

[73] D. Joyner, O. Čertík, A. Meurer, B. E. Granger, Open source computer algebra systems: Sympy, ACM Communications in Computer Algebra 45 (3/4) (2012) 225–234.

[74] F. Smailbegovic, G. N. Gaydadjiev, S. Vassiliadis, Sparse matrix storage format, in: Proceedings of the 16th Annual Workshop on Circuits, Systems and Signal Processing, 2005, pp. 445–448.

[75] J. B. White, P. Sadayappan, On improving the performance of sparse matrix-vector multiplication, in: Proceedings Fourth International Conference on High-Performance Computing, IEEE, 1997, pp. 66–71.

[76] X. S. Li, M. Shao, A supernodal approach to incomplete LU factorization with partial pivoting, ACM Trans. Mathematical Software 37 (4).

[77] T. A. Davis, Algorithm 832: Umfpack v4. 3—an unsymmetric-pattern multifrontal method, ACM Transactions on Mathematical Software (TOMS) 30 (2) (2004) 196–199.

[78] S.-S. Developers, Scikit-sparse sparse matrix cholesky decomposition for python, Computer Science (2017).
URL `https://github.com/scikit-sparse/scikit-sparse`

[79] F. Hecht, New development in freefem++, J. Numer. Math. 20 (3-4) (2012) 251–265.

[80] T. Dupont, J. Hoffman, C. Johnson, R. C. Kirby, M. G. Larson, A. Logg, L. R. Scott, The fenics project, Chalmers Finite Element Centre, Chalmers University of Technology, 2003.

[81] C. Geuzaine, J.-F. Remacle, Gmsh: A 3-d finite element mesh generator with built-in pre-and post-processing facilities, International journal for numerical methods in engineering 79 (11) (2009) 1309–1331.

[82] C. B. Sullivan, A. Kaszynski, PyVista: 3d plotting and mesh analysis through a streamlined interface for the visualization toolkit (VTK), Journal of Open Source Software 4 (37) (2019) 1450. `doi:10.21105/joss.01450`.
URL `https://doi.org/10.21105/joss.01450`

[83] B. Daya Reddy, Introductory functional analysis: With applications to boundary value problems and finite elements, Texts in Applied Mathematics 1 (27) (1998) ALL–ALL.

[84] W. J. Schroeder, B. Lorensen, K. Martin, The visualization toolkit: an object-oriented approach to 3D graphics, Kitware, 2004.

[85] J. Sokolowski, J.-P. Zolesio, Introduction to shape optimization, in: Introduction to Shape Optimization, Springer, 1992, pp. 5–12.

[86] M. H. Sadd, Elasticity: theory, applications, and numerics, Elsevier Science, 2014.

[87] O. C. Zienkiewicz, R. L. Taylor, P. Nithiarasu, The finite element method for fluid dynamics; 7th ed., Elsevier Science, 2013.

[88] P. Gaskell, M. Savage, J. Summers, H. Thompson, Stokes flow in closed, rectangular domains, Applied Mathematical Modelling 22 (9) (1998) 727–743.

[89] A. M. Aguirre-Mesa, M. J. Garcia-Ruiz, M. Aristizabal, D. Wagner, D. Ramirez-Tamayo, A. Montoya, H. Millwater, An efficient approach to solve the system of equations of hypercomplex finite element methods using a block forward substitution scheme, Article under development.

[90] Y. Tian, Matrix representations of octonions and their applications, Advances in Applied Clifford Algebras 10 (1) (2000) 61–90.