UNIVERSITY OF LEEDS

Sustainable research software engineering practices for fluctuating finite element analysis

Jarvellis Rogers

Submitted in accordance with the requirements for the degree of PhD Computer Science

The University of Leeds

Faculty of Engineering

School of Computing

May 2025

Abstract

Research software is often developed with a focus on the output over the software's quality, and made by researchers untrained in formal software development techniques. Conditions such as these lead to an unsustainable environment of rapid software decay, and the wasteful re-invention of software for similar research purposes. The field of research software engineering (RSE) was coined to focus on the development and maintenance of sustainable research software.

Fluctuating Finite Element Analysis (FFEA) is a numerical method and software suite that uses continuum mechanics to simulate mesoscopic biomolecules. As research software, FFEA is subject to the same conditions which threaten software decay. This research applies sustainable RSE principles to FFEA to analyse and mitigate for this threat.

FFEA is evaluated to determine the major bottlenecks in the workflow impairing the software's sustainability. Results show these issues stem from the difficulty and time cost of setting up FFEA experiments, especially due to the current meshing algorithm producing unstable outputs that disrupt setup and frequently terminate FFEA simulations due to physics errors.

Two solutions are developed to attempt to solve these problems. The first extends the features of a new meshing algorithm started by Gravett (2022) which divides voxel representations of molecules into tetrahedra to create a consistent, stable geometry. This research adds new features to the meshing algorithm, and compares it to the previous meshing algorithm to show that it produces meshes more reliably with fewer steps in the process.

The second solution works towards automatic generation of FFEA KOBRA models for rod-shaped biomolecules through using machine learning algorithms to predict the presence of rod structures. Machine learning algorithms were able to categorise isosurface meshes of biomolecules through analysing the curvatures across the surface. This proved the concept and highlights necessary steps required to achieve full automation.

Intellectual Property

The candidate confirms that the work submitted is their own and that appropriate credit has been given where reference has been made to the work of others.

This copy has been supplied on the understanding that it is copyright material and that no quotation from the thesis may be published without proper acknowledgement.

© 2025 The University of Leeds, Jarvellis Rogers

Signed

Acknowledgements

The author wishes to express gratitude towards the FFEA development team, especially for the direct assistance from Ryan Cocking, Molly Gravett, Ben Hanson, Oliver Harlen, Sarah Harris, Joanna Leng, Daniel Read, Tom Ridley, Rob Welch, and Changqing Zheng. Their contributions to, and expert knowledge of the FFEA software was invaluable to this research.

The author is also grateful for the organisational and project management assistance from their supervision team, David Head and Mark Walkley, as well as from their postgraduate research administrator, Victoria Masters. Their support ensured the completion of this research during a difficult time.

Finally, the author is grateful for the support of their wife, Maiev Hart, for their assistance in proof reading, and who's knowledge of biology was of frequent help during the process of this interdisciplinary research.

The LaTeX template and stylesheet for this document was created by Daniel Wise of the University of Leeds in 2019.

Contents

Glossary **xiv**

Acronyms **xx**

1 Introduction **1**

1.1 Research scope 4

1.2 Methodology 6

2 Background research **10**

2.1 Research software engineering and software sustainability 10

2.1.1 Research software engineering 10

2.1.2 Research software sustainability 11

2.1.3 Open issues in research software sustainability 17

2.2 Computational structural biology sustainability 18

2.2.1 Structural biology and molecular dynamics 18

2.2.2 Computational solutions 20

2.2.3 Sustainability of computational solutions to structural biology – key challenges 24

2.3 Finite element method 25

2.4 Fluctuating finite element analysis software suite 28

2.5 Chapter conclusions 31

3 Evaluation of modern novel research software – A case study of fluctuating finite element analysis. Part I: Quantitative analysis. **33**

3.1 Static code analysis 34

3.1.1 Results . . . 36
3.2 Dynamic code analysis . . . 37
3.2.1 Memory analysis results . . . 42
3.2.2 Cache analysis results . . . 50
3.2.3 Threading analysis results . . . 54
3.2.4 Heap analysis results . . . 57
3.2.5 Dynamic code analysis conclusions . . . 60
3.3 Soak testing . . . 60
3.3.1 Results . . . 62
3.4 Quantitative analysis conclusions . . . 66

4 Evaluation of modern novel research software – A case study of fluctuating finite element analysis. Part II: Qualitative analysis and evaluation conclusions. 68

4.1 Survey design . . . 69
4.2 Survey results . . . 70
4.2.1 User responses . . . 71
4.2.2 Developer responses . . . 89
4.2.3 Participant final remarks . . . 96
4.3 Analysis . . . 97
4.3.1 User work practices . . . 98
4.3.2 User experience . . . 100
4.3.3 User and algorithm performance . . . 102
4.3.4 Collaboration and communication . . . 103
4.3.5 Developer work practices . . . 104
4.3.6 Developer experience . . . 105
4.3.7 Qualitative analysis conclusions . . . 106
4.4 Identifying the bottleneck in the workflow . . . 108

5 Fluctuating finite element analysis tetrahedral meshing 112

5.1 File formats for FFEA tetrahedral meshing . . . 113
5.2 FFEA tetrahedral meshing process . . . 115
5.2.1 Limitations of the FFEAtools meshing algorithm . . . 117

5.3 An alternative method of meshing 119
5.3.1 Algorithm process 120
5.3.2 Limitations of the alternative method of meshing and conclusions 122

6 Marching diamond crystal lattice tetrahedral meshing algorithm 124
6.1 Coarsening the mesh 125
6.2 Interpolation for a smoothed mesh 127
6.3 Stability check 128
6.4 Bug fixing and code tidying 129
6.5 Feature-complete algorithm process 130
6.6 Summary of algorithm improvements 136

7 Analysis of the marching diamond crystal lattice tetrahedral meshing algorithm 138
7.1 Comparative analysis 138
7.1.1 Static mesh quality analysis 140
7.1.2 Simulation performance analysis 151
7.1.3 Simulation result analysis 153
7.2 Current limitations 165
7.3 Comparative analysis conclusions 166

8 Towards automated KOBRA rod identification 170
8.1 Classification algorithms 171
8.1.1 K-nearest neighbours 172
8.1.2 Decision trees 174
8.1.3 Support vector machines 176
8.1.4 K-fold cross-validation 179
8.2 Electron density map classification 179
8.2.1 Results 180
8.3 Curvature cluster analysis classification 182
8.3.1 Validation of method 183
8.3.2 MDCL-TMA as input 188
8.3.3 Isosurfaces as input 191

8.4 Chapter conclusions and future prospects 198

9 Conclusions and future prospects 201

9.1 Summary of progress 201

9.1.1 Marching diamond crystal lattice tetrahedral meshing algorithm 203

9.1.2 Machine learning for classifying biomolecule geometry 206

9.1.3 Generic applications to research software sustainability 207

9.2 Limitations of this research 208

9.3 Future prospects 210

9.4 Final remarks 212

References 213

A Qualitative Assessment of the Fluctuating Finite Element Analysis (FFEA) Software Questionnaire 227

B Datasets for rod classification 235

List of Figures

1.1 Generic methodology of this research 7

2.1 The cycle of reinventing research software 12

2.2 The four pillars of research software engineering 16

2.3 Examples of molecular dynamics visual representations 19

2.4 Example of a coarse-graining process on an all-atom model of dipalmitoylphosphatidylcholine 22

2.5 Example visualizations of finite element analysis results 28

2.6 SARS-CoV-2 viral envelope in blob format and myosin 5a in KOBRA rod format for FFEA 30

3.1 Example of the leak summary tables from Valgrind's Memcheck's log files 42

3.2 Example of a loss record from Valgrind's Memcheck's log files 43

3.3 Logarithmic Bar charts showing the number of blocks with memory leaks in FFEA 2.7.2. 45

3.4 Logarithmic bar charts showing the number of bytes lost to memory leaks in FFEA 2.7.2. 46

3.5 Logarithmic bar charts showing the range of bytes directly and indirectly lost by memory errors in FFEA 2.7.2 (disregarding `Bede_MB2`). 48

3.6 Logarithmic bar charts showing the range of bytes directly and indirectly lost by memory errors in FFEA 2.7.2 (including `Bede_MB2`). 50

3.7 `ffea.cpp` lines 164 & 165 pointing to the class `World()` 50

3.8 Example of cache usage table from Valgrind's Callgrind function 51

3.9 Example of function-level counts for cache usage by Valgrind's Cachegrind/Callgrind 51

3.10 Bar chart showing the cache miss rates in FFEA 2.7.2. 52
3.11 `World.cpp` lines 4460-4484 containing the function `calc_blob_corr_matrix()` . . 54
3.12 Example of Valgrind's Helgrind's reports on a possible data race 55
3.13 Stacked bar chart showing the sizes of possible data races in FFEA 2.7.2. 56
3.14 Example of a histogram of heap memory consumption produced by Valgrind's Massif tool 57
3.15 Example of the heap usage tables for each snapshot produced by Valgrind's Massif 58
3.16 Bar chart showing the useful and extra heap memory consumption for FFEA 2.7.2. 59
3.17 Percentage bar chart showing the ratio of useful to extra heap memory consumption for FFEA 2.7.2. 59
3.18 Example results from a soak test for FFEA 2.7.2. 62
3.19 Bar chart showing the soak test wall clock times in FFEA 2.7.2. 63
3.20 Bar chart showing the soak test wall clock times in FFEA 2.7.2. with 32GB RAM limit 65
3.21 Bar chart showing the comparison of average wall clock times between the two sets of soak tests with RAM limits of 256GB and 32GB. 65

4.1 User statement answers for FFEAtools. 83
4.2 User statement answers for the FFEA runner. 85
4.3 User statement answers for the FFEA visualizer. 87
4.4 Average user statement answers. 88
4.5 Developer statement answers for FFEAtools. 93
4.6 Developer statement answers for the FFEA runner. 94
4.7 Developer statement answers for the FFEA visualizer. 95
4.8 Average developer statement answers. 96

5.1 Example of a dynein motor protein mesh created with FFEAtools 115
5.2 Example of mesh sliver-element inversion during a simulation timestep 118
5.3 Voxel slicing via marching diamond crystal lattice 119
5.4 Alternating interior regular tetrahedra of adjacent voxels sliced by MDCL 120
5.5 Flowchart of marching tetrahedra FFEA meshing algorithm process 122

6.1 Example of an input image zoomed using `ndimage.zoom` 126

6.2 Comparison of groEL MDCL-TMA mesh without and with interpolation 127
6.3 Example of interpolation of one voxel in MDCL-TMA 128
6.4 MDCL-TMA Python prototype command-line user warning of potential instability for meshes coarsened under 5Å. 129
6.5 Comparison of original MRC data with overlaid bugged output MRC data 129
6.6 Example of the MDCL-TMA process rendered in 2D 131
6.7 Flowchart of MDCL-TMA process 134
6.8 Flowchart of MDCL-TMA sub-process "Iterate over voxels" 135

7.1 GroEL meshed using FFEAtools and MDCL-TMA viewed in ParaView 144
7.2 Human cytoplasmic dynein-1 meshed using FFEAtools and MDCL-TMA viewed in ParaView 144
7.3 MDCL-TMA meshes viewed in ParaView 145
7.4 Ratio of minimum to maximum edge lengths of meshed objects 146
7.5 Quality-fitness of meshed objects 147
7.6 Comparison of frequency histograms of mesh quality-fitness 148
7.7 Example of a trirectangular tetrahedron 148
7.8 Soak test wall clock times of meshed objects 153
7.9 Comparison of overlaid FFEAtools meshes and MDCL-TMA meshes for groEL and the stalks of human cytoplasmic dynein-1, viewed in ParaView 156
7.10 Strain energy over time during 1ns FFEA simulations for meshed objects 157
7.11 Strain energy over time during 1ns FFEA simulations for meshed objects (cont.) 158
7.12 Strain energy over time during 1ns FFEA simulations for groEL meshed with FFEAtools to 8Å and with MDCL-TMA to 11Å. 159
7.13 Strain energy over time during 1ns FFEA simulations for groEL meshed with FFEAtools to 5Å, 8Å, and 12Å. 160
7.14 Centroid deviation over time during 1ns FFEA simulations for groEL 160
7.15 Centroid deviation over time during 1ns FFEA simulations for groEL split into runs 161
7.16 Average root-mean-square deviation from repeated 1ns FFEA simulations of meshed objects 162

7.17 Centroid deviation over time during 1ns FFEA simulations for groEL with MDCL-TMA surface mesh 163
7.18 Centroid deviation over time during 1ns FFEA simulations for groEL, with MDCL-TMA surface mesh, split into runs 164
7.19 SARS-CoV-2 virion meshed in FFEAtools and MDCL-TMA 165

8.1 Example of the k-nearest neighbours method for a 2D binary classification problem 173
8.2 Example of a decision tree for choosing to wear a coat (a binary classification problem) 174
8.3 Three different linear classification lines for the same dataset 176
8.4 Dataset and classification lines from Figure 8.3 with new data 176
8.5 Dataset from Figure 8.3 with maximum margin linear classifier 177
8.6 Scatter plots of sphere convergence tests for curvature analysis 185
8.7 Scatter plots of prolate spheroid convergence tests for curvature analysis 185
8.8 Sphere and prolate spheroid curvature heat maps viewed in ParaView 186
8.9 Scatter plots of torus curvatures 186
8.10 Torus curvature heat map 187
8.11 Scatter plots of MDCL-TMA mesh principal curvatures 189
8.12 Scatter plots of MDCL-TMA mesh mean and Gaussian curvatures 190
8.13 2D example of MDCL-TMA mesh showcasing issue with curvature analysis . . . 191
8.14 2D examples of how different molecule structures fill the volume of a bounding box 193

List of Tables

2.1 Results of the frequency of features of sustainability mentioned by interview participants . . . 13

3.1 `FFEA_linter` results for FFEA 2.7.2 . . . 36

3.2 Meshes of each system used as datasets for FFEA_profiler . . . 40

3.3 Mesh and FFEA simulation data for each FFEA_profiler dataset . . . 40

3.4 System information of the machines used to profile FFEA[1] . . . 41

3.5 Memcheck results for number of blocks with memory leaks in FFEA 2.7.2 . . . 44

3.6 Memcheck results for number of bytes lost to memory leaks in FFEA 2.7.2. . . . 46

3.7 Memory errors in FFEA 2.7.2 (disregarding `Bede_MB2`). . . . 47

3.8 Memory errors in FFEA 2.7.2 (including `Bede_MB2`). . . . 48

3.9 Threadcheck results for number of thread errors in FFEA 2.7.2. . . . 55

3.10 Sizes of possible data races in FFEA 2.7.2 . . . 56

3.11 Heap memory usage in FFEA 2.7.2 . . . 58

3.12 Results of soak tests on FFEA 2.7.2. . . . 62

3.13 Results of soak tests on FFEA 2.7.2. with 32GB RAM limit. . . . 64

5.1 List of file formats used in FFEA tetrahedral meshing . . . 113

7.1 Static analysis results of meshed objects . . . 142

7.2 Simulation performance analysis results of meshed objects when ran in 48-hour FFEA simulations. . . . 152

7.3 Results of 1ns FFEA simulation measurements of meshed objects. . . . 155

7.4 Mesh details and results of 1ns FFEA simulations of groEL MDCL-TMA surface mesh at 11Å. . . . 163

8.1 K-fold cross-validation scores for rod classification from electron density map point cloud data 181
8.2 The radii and the number of segments across planes for meshes of test objects for curvature analysis validation. 184
8.3 K-fold cross-validation scores for rod classification from biomolecule isosurfaces (small dataset) 195
8.4 K-fold cross-validation scores for rod classification from biomolecule isosurfaces (large dataset) 197

B.1 Dataset of biomolecule electron density maps used for all experiments in Chapter 8.236
B.2 Dataset of biomolecule electron density maps used for the second series of experiments in §8.3.3 237

Glossary

Actin-tropomyosin Proteins that regulate the functions of actin filaments, such as muscle contraction.

Adenosine Triphosphatase (ATPase) An essential enzyme which utilises the energy released from breaking the phosphate bond of adenosine triphosphate to perform other cellular chemical reactions.

Angstrom (Å) A unit of length. 1Å = 0.1nm.

Biomolecule A molecule utilised in a biological process(es).

Biophysics An interdisciplinary science that studies biological phenomena using the theories and methods of physics.

Capsid The protein shell of a virus.

Cell (Biology) The basic structural unit of organisms.

Cellulose A chain compound of glucose that is a common structural component of cell walls in green plants (and other organisms that use similar cell structures).

Centroid The mean position of all points in a geometric system.

Chromatophore The cells responsible for generating colour in the skin and iris that are commonly studied in epidemiology and used as a tool for drug discovery.

Clathrin A protein that forms structural cages as part of the formation of coated vesicles.

Coarse-grain A simplification of a model by increasing the minimum size of model components.

Continuum Mechanics The study of the mechanical behaviours of materials of continuous

matter.

Contour Level / Isolevel The surfacing threshold of point cloud data geometry for the creation of an isosurface.

Cryo-Electron Microscopy (Cryo-EM) An imaging technique that uses electron microscopy on cryogenic samples.

Cryogenics The creation and study of very low temperatures.

Cytoplasm The solution that fills a cell.

Degrees of Freedom (Mechanics) The number of motions in a body that may vary independently.

Dengue Virus A mosquito-borne virus that causes the widespread disease dengue fever.

Density The quantity of mass per unit of volume.

Deoxyribonucleic Acid (DNA) A double-stranded polymer carrying genetic instructions for biological functions and processes.

Dielectric Electrically insulating due to having no free electrons.

Dihederal Angle The angle between intersecting planes, or those connected by an edge.

Dipalmitoylphosphatidylcholine (DPPC) An organic compound which can be used in pulmonary surfactants to ease the effort of breathing.

Displacement (Geometry) The difference between the final and initial position of a point translated through space.

Dynamics The mechanical study of forces and their effects on motion.

Dynein A family of motor proteins that converts the stored chemical energy of adenosine triphosphate to mechanical work. Used in functions such as mitosis.

Elasticity (Physics) The ability of matter to resist deformity and to return to its original shape after deformation.

Electron Cryotomography An imaging technique that uses electron microscopy on cryogenic samples to record a series of 2D images as the sample is tilted to yield a 3D reconstruction. Also known as cryo-electron tomography.

Electron Density Map A 3D map of the density of electrons in a crystal structure. Also the name of a file format that contains this information.

Element Inversion An FFEA error where a node of a mesh element passes through a face or edge of that element, resulting in non-physical behaviour.

Entropy (Information) The amount of information provided by the potential outcomes of a variable.

Entropy (Physics) The thermal energy per unit of temperature in a system that cannot be transferred.

FFEA Runner The component of the FFEA suite that contains the FFEA algorithm and computes FFEA trajectories from an FFEA input file containing a system and its conditions, primarily written in C++.

FFEA Suite The holistic FFEA software, i.e. the FFEA runner, the FFEA visualiser, and FFEAtools combined.

FFEA Visualiser The plugin for PyMOL that visualises FFEA simulations from an FFEA input file, primarily written in Python.

FFEAtools An independent Python package for the FFEA suite containing analysis tools and tools for assisting users in the creation of FFEA files and simulations.

Finite Element Method (FEM) A numerical method for solving engineering and mathematical model problems by solving partial differential equations. Simplifies a model or system by subdividing into elements connected to points in a mesh called nodes with respective equations attached. Analysis with this method is called finite element analysis.

Fluctuating Finite Element Analysis (FFEA) A software package designed to perform continuum mechanics simulations of proteins and other globular macromolecules on the mesoscale by combining finite element analysis with stochastic thermal noise.

Gene Sequencing The process of determining the order of nucleotide molecules in nucleic acids.

Glucagon A hormone produced by the pancreas that raises the concentration of glucose and fatty acids in the bloodstream.

Graphene An assembly of carbon atoms in a hexagonal lattice.

GroEL A protein (commonly found in bacteria) used in protein folding.

Hemoglobin The protein found in red blood cells which transports oxygen.

Human Epidermal growth factor Receptor 2 (HER2) A human protein of scientific interest due to its role in the development of some types of breast cancer.

Human Immunodeficiency Virus 1 (HIV-1) The more common, virulent, and transmissible of the two major types of the virus that causes acquired immune deficiency syndrome (AIDS).

Hyperspectral Pertaining to information from the electromagnetic spectrum.

Isosurface A surface created from points of a set constant value in 3D continuous data.

Isotropic Having uniform properties in all directions.

Kinetic Energy The energy of an object due to motion.

Kinetochore A protein structure which helps attach chromosomes to microtubules.

Kirchoff Biological Rod Algorithm (KOBRA) A model for representing the dynamics of elongated biomolecules in FFEA.

Lactic Acid An organic acid that is a waste product of anaerobic (i.e. without oxygen) respiration.

Leucocyte White blood cells. Responsible for protecting the body against foreign materials such as infections.

Macromolecule A molecule with a large number of atoms.

Marching Diamond Crystal Lattice Tetrahedral Meshing Algorithm (MDCL-TMA) A method for the generation of computational methods by slicing voxels into five tetrahedra with alternating inner tetrahedra.

Mechanics The study of the relationships between forces, matter, and motion.

Mesoscale Materials in the size range between the nanoscale and microscale. Generally in the range between approximately 100-1000nm (or 1000-10000Å).

Microtubule Polymers of the protein tubulin that forms part of the cytoskeleton for structure and intracellular transport of proteins.

Molecular Dynamics A method of analysing the spatial location of atoms and molecules through computer simulations.

Myosin Motor proteins involved in motility processes such as muscle contraction.

Neuron Also known as a nerve cell. Sends and receives signals across the nervous system of an organism to allow it to react to stimuli and control motor functions.

Nuclear Division Cycle 80 Complex (NDC80C) A microtubule receptor at the kinetochore.

Nucleic Acid Biomacromolecules that contain genetic information.

Potential Energy The stored energy of an object due to its position (such as within a force field) or condition (such as under mechanical stress).

Protein A biomacromolecule composed of amino acids that perform a vast number of essential biological functions.

Quality-Fitness The ratio of a tetrahedron's inradius to circumradius multiplied by 3. Results in a value between 0-1, with 1 representing the tetrahedron's volume fitting perfectly to that of a regular tetrahedron.

Residue (Chemistry) A leftover product of a chemical reaction.

Ribonucleic Acid (RNA) A single-stranded polymer carrying genetic instructions for biological functions and processes.

Rigid Body A solid object with zero or negligible deformation.

Rod Dynamics Explosion An FFEA error where a KOBRA model has gained an infinite or very large energy value, causing non-physical behaviour.

Severe Acute Respiratory Syndrome Coronavirus 2 (SARS-CoV-2) The virus responsible for the respiratory illness COVID-19.

Software Decay / Rot The deterioration of the quality of a given software.

Software Sustainability A term with a contested definition (see §2.1). Generally refers to

software aspects such as longevity, maintainability, and source code readability. This thesis defines it as "the qualities of software which impact its usability, maintainability, and longevity".

Stator The stationary component of a rotary system such as a motor.

Stiffness (Mechanics) The ability of an object to resist deformation.

Stimulus Something which causes activity in an object or system.

Strain (Mechanics) The relative deformation of physical matter under applied forces.

Stress (Mechanics) The forces present during the deformation of physical matter.

Structural Biology The study of biomolecules' structure and dynamics.

Super-Resolution Microscopy A series of imaging techniques in optical microscopy that can image samples at higher resolutions than traditional microscopy, which is limited by the diffraction limit.

Synapse The connective structure between neurons that passes electrical or chemical signals.

Thermal Noise The random motion of free electrons in a conducting material caused by heat. Also known as thermal agitation.

Thermodynamics The study of the relationships between heat, work, temperature, and energy.

Topography The study of geometric surfaces.

Topology The study of geometric properties that are preserved under deformations, those that do not change the number of holes in a geometry.

User Retention The number of users who continue to use a product or system.

van der Waals Force A weak, non-chemical and non-electric bonding force between atoms or molecules.

Viscoelasticity The behaviour of materials with simultaneous fluid and elastic properties.

Viscosity A measurement of a fluid's ability to resist deformation.

Yeast A single-cell fungus important to humans for the process of fermentation.

Acronyms

AMBER Assisted Model Building with Energy Refinement.

API Application Programming Interface.

ATPase Adenosine TriPhosphatase.

BASH Bourne-Again SHell.

CAD Computer-Aided Design.

CAM Computer-Aided Manufacturing.

CCP Collaborative Computational Project.

COVID COronaVIrus Disease.

CPU Central Processing Unit.

Cryo-EM Cryo-Electron Microscopy.

CSV Comma-Separated Values.

CUDA Compute Unified Device Architecture.

DNA DeoxyriboNucleic Acid.

DNN Deep Neural Network.

DPPC DiPalmitoylPhosphatidylCholine.

DT Delta Time.

EI Element Inversion.

EMDB Electron Microscopy Data Bank.

ERR ERRor.

FEA Finite Element Analysis.

FEM Finite Element Method.

FF-DT Feature Fusion and multi-layered gradient boosting Decision Tree.

FFEA Fluctuating Finite Element Analysis.

FFmpeg Fast Forward Moving Picture Experts Group (software).

GAMer Geometry-preserving Adaptive MeshER.

GB GigaByte.

GCC GNU Compiler Collection.

GDB GNU DeBugger.

GNU GNU's Not Unix! (recursive acronym).

gprof GNU PROFiler.

GPU Graphics Processing Unit.

GUI Graphical User Interface.

HER2 Human Epidermal growth factor Receptor 2.

HIV Human Immunodeficiency Virus.

HPC High-Performance Computing.

I/O Input/Output.

ID InDex (or unique IDentifier).

IDE Integrated Development Environment.

k-NN k-Nearest Neighbours.

KB KiloByte.

KOBRA KirchOff Biological Rod Algorithm.

MB MegaByte.

MD Molecular Dynamics.

MDAnalysis Molecular Dynamics Analysis (software).

MDCL Marching Diamond Crystal Lattice.

MDCL-TMA Marching Diamond Crystal Lattice Tetrahedral Meshing Algorithm.

MIC Many Integrated Core.

MRC Medical Research Council (file format).

N/A Not Applicable.

NDC80C Nuclear Division Cycle 80 Complex.

OBJ wavefront OBJect (file format).

PATH Not an acronym. Linux environment variable.

PC Personal Computer.

PDB Protein Data Bank.

PNG Portable Network Graphics.

PointCNN Point Convolutional Neural Network.

POSIX Portable Operating System Interface.

PyMOL A pseudo-acronym. A Python-based molecular dynamics visualization tool.

QF Quality-Fitness.

QUARK A pseudo-acronym. A protein structure prediction tool.

RAM Random Access Memory.

RBF Radial Basis Function.

RCSB Research Collaboratory for Structural Bioinformatics.

RDE Rod Dynamics Explosion.

RMSD Root Mean Square Deviation.

RNA RiboNucleic Acid.

RSE Research Software Engineer(ing).

SARS-CoV-2 Severe Acute Respiratory Syndrome CoronaVirus 2.

SCWRL Side Chains With Rotamer Library (software).

SI Système International (units).

STL STereoLithography (file format).

SVM Support Vector Machines.

UCSF Chimera University of California, San Francisco Chimera (software).

UI User Interface.

UV Not an acronym. Represents the u and v axes of 2D textures for a 3D model (where the x, y, and z axes labels are already in use to describe the geometry of the 3D model).

UX User eXperience.

VMD Visual Molecular Dynamics.

VTK Visualisation ToolKit.

WCT Wall Clock Time.

XML eXtensible Markup Language.

Chapter 1

Introduction

> Computational biophysics... should not let the speed and availability of computing power set the upper bound on the size and complexity of molecular dynamics systems. We cannot depend on the exponential growth of computing speed to solve ever more difficult problems.
>
> *Robert Welch (2020)*

Over the past few decades, structural biology has been benefiting from the combination of computational methods (such as simulation) and traditional physical "real-world" experimentation to enhance scientific understanding and discovery (Kmiecik et al. 2016; Gravett, Cocking et al. 2021; Zhou and Liu 2022). This has allowed researchers to investigate the physical and chemical properties of biomolecular systems without always needing to resort to expensive and time-consuming real-world experiments, and without needing to manually perform complex calculations such as gene sequencing and analysing physical forces across an entire system.

While these computational methods have been a boon to biophysics researchers, they also have to contend with computational problems that can arise from trying to design and develop these methods. These can be challenging problems for many researchers to understand and solve, as they are typically experts in their own field, not necessarily in software development, and may have had no training in this field despite requirements or expectations to produce research software. These conditions, which are the same for research software in most fields (even sometimes computer science), often lead to an environment of rapid software decay, meaning research software is in a constant cycle of wasteful re-invention as tools need to be re-made each time the same or similar research topics are tackled (Cohen, Katz, Barker, Haines et al. 2018;

Souza et al. 2019).

The work in this thesis uses, as an exemplar, the research software tool Fluctuating Finite Element Analysis (FFEA), which implements the FFEA numerical method to simulate large biological macromolecules (Solernou et al. 2018), in particular those in the mesoscale (a length scale of approximately 100-1000nm), which other traditional methods of molecular dynamics are unable to simulate the dynamics of accurately, and are unable to produce simulations within a reasonable time frame. FFEA is a novel piece of research software, which has been developed over a decade by numerous contributors, but as a piece of research software it is subject to the same conditions that contributes to the decay of research software. This research aims to apply sustainable research software engineering (RSE) principles to FFEA, in order to prevent this work from experiencing software decay and poor user retention.

Chapter 2 begins this work by reviewing appropriate literature for background research. This starts by exploring the field of RSE and software sustainability, to try and get as clear of an understanding as possible as to what makes software sustainable, and to have a good understanding of these principles to apply to FFEA. The research topics of structural biology and molecular dynamics are then introduced, to gain an understanding of the general aims and applications of these fields, and then how computational methods have been applied to these fields to expand upon their capabilities. This is done to understand the existing computing strategies implemented for these fields, and their limitations. This chapter then closes with an explanation of FFEA and the finite element method it is based on.

Before work can be started on improving the sustainability of FFEA, it is necessary to know exactly what are the current problems with the software. If work is immediately started on trying to apply the sustainable RSE principles explored in Chapter 2, it would be very possible to overlook issues that should be of a higher priority to address. Therefore, Chapters 3 and 4 are dedicated to a thorough evaluation of the current version of FFEA to identify these issues and the major bottlenecks in the workflow of using the software. Chapter 3 details quantitative analysis of FFEA to determine potential problems in the performance and structure of the FFEA source code. This is done via static code analysis through linting, dynamic code analysis from running datasets in the FFEA runner executable, and soak testing. Chapter 4 then details qualitative analysis through surveying users and developers of FFEA to determine potential problems in the user experience, and to gain subjective opinions on the performance of FFEA

to compare with the quantitative performance results in the prior chapter.

The result of this evaluation identifies the setup of FFEA experiments to be the main bottleneck in the workflow, especially in regards to generating meshes, due to the large number of steps requiring trial-and-error parametrisation. Outputs of each step in FFEA mesh generation can also frequently produce unstable geometries that cause failures in subsequent steps, and the unstable geometry of a final mesh can frequently result in physics errors that prematurely terminate FFEA simulations. The rest of the thesis is dedicated to trying to find solutions to the issues causing this difficult and overly-long workflow.

One potential solution to the problematic workflow for setting up FFEA experiments is to develop a new meshing algorithm with stability in mind. Gravett (2022) began work on a new meshing algorithm for use in their research, and Chapter 5 details this work, as well as explaining the current process of generating meshes with FFEAtools (a suite of supplementary Python tools for FFEA) so that these algorithms can be compared. The new meshing algorithm as presented by Gravett (2022) generates tetrahedral meshes of a consistent geometry by diagonally slicing voxels. While this new meshing algorithm was successfully applied to their research, it was not in a feature-complete state ready for a wider audience to use. Chapter 6 details the development work on the new meshing algorithm (tentatively named the "Marching Diamond Crystal Lattice Tetrahedral Meshing Algorithm" or MDCL-TMA) performed as part of this thesis, to develop any missing features (such as interpolation to produce a smooth mesh surface) and to improve the usability of the tool. Chapter 7 then compares the feature-complete prototype of MDCL-TMA to the FFEAtools mesh generation method, to see if MDCL-TMA improves upon mesh generation for FFEA systems. This comparative analysis is done through comparing the quality of static meshes produced by each algorithm, comparing the performance of FFEA simulations using meshes generated with both methods, and comparing the results of FFEA simulations using both types of mesh.

FFEA meshes come in two formats; tetrahedral meshes for globular molecules, and Kirchoff Biological Rod Algorithm or KOBRA (Welch 2020) meshes for rod structures in molecules. Utilising KOBRA meshes also comes with difficulties in setting up FFEA experiments, especially when it is needed to combine them with other structures, as this is a manual process that uses text fields instead of a GUI with instant feedback. Therefore, another potential solution to this general workflow problem is to try and develop a feature for automating the genera-

tion of KOBRA meshes. Chapter 8 works towards this goal by developing machine learning models to try and classify biomolecule geometric data into two categories, those that contain rod structures, and those that do not, to see if this geometry can be discerned algorithmically. Multiple attempts of using different formats of biomolecule geometric data as input to machine learning algorithms are performed to try and prove this concept, and to develop potential future strategies for achieving the final goal of full automation.

This thesis then closes with Chapter 9, which presents the conclusions to this research through summarising current progress made and discussing future prospects to potentially improve upon this progress.

Due to the involvement of interdisciplinary research, this thesis also contains a glossary of terms and acronyms at the start to ensure the reader is familiar with the terminology used throughout this work.

1.1 Research scope

Shaw (2003) poses three questions that software engineering papers should answer to indicate well-presented results; What, precisely, was your contribution? What is your new result? Why should the reader believe your result? Theisen et al. (2017) indicated the importance of these questions by noting the impact of whether or not they were answered on review committees' acceptance rate of papers (see §2.1.2 for further detail).

This research builds on the established methodology presented by Shaw (2003) and applies it specifically to research software. The aim of this particular thesis is to address the impact of sustainable qualities and sustainable design practices on the user experience, maintainability, and longevity of research software. Considering the framework of Shaw's questions, and the specific topics of this research, three research questions have been devised that this thesis aims to answer:

1. How can research software be systematically evaluated in order to gain a holistic understanding of its sustainable qualities and their impacts on software usability, maintainability, and longevity?

2. When it is understood what the current sustainable software qualities of a piece of research software are, how could these sustainable software qualities be improved?

3. Will improving the sustainable software qualities of research software improve the user experience, maintainability, and the software longevity?

Answering question 1 is a necessity before attempting any kind of improvement to the sustainability of a piece of software. First of all, it would be difficult to attempt any kind of improvement to software without understanding its current state; to understand its current issues and how to prioritise them, and what qualities are either necessary or beneficial for the software and should *not* be changed (or at least should be preserved in some form). This also provides a record of the "before" state of the software to compare to the "after" state once changes have been made, so that the reader can understand exactly what has been contributed and what the new results are, as per Shaw's questions.

Question 2 prompts the researcher to consider what could be contributed to research software development to improve the sustainable software qualities. This question provides the limits of the scope of this research – it is an aim of this research to seek changes to research software that improve the sustainable software qualities of research software. We do *not* seek to improve any other aspects of software (such as result accuracy, capability of its functions, or marketability), and we are specifically addressing *research* software, not software designed for any other purpose (such as creative, communication, or recreational software).

Finally, question 3 prompts the researcher to conduct a comparative analysis of the research software before and after changes were made. It needs to be clearly shown how the changes to the research software correlate with improving its sustainable software qualities, so that as per Shaw's third question the reader can clearly understand the validity of the results.

To narrow the scope of this research project, this thesis applies these research questions through a case study of an exemplar of this type of research software – Fluctuating Finite Element Analysis (Solernou et al. 2018). Attempting to apply these questions to research software as a whole would require deep studies of multiple pieces of research software, which is beyond the time-frame of this research. This case study will test the effectiveness of this approach and the methodology used, and develop guidelines and best practice to apply to other research software. If so, similar studies to this one can then be conducted to compare and contrast results to get closer to more holistic answers to these questions.

As this research aims to discuss the importance of software sustainability this thesis must be

upheld to the standards of well-presented software engineering results, and thus will aim to answer Shaw's base questions as well.

1.2 Methodology

Software engineering currently lacks a consensus on the definition of software sustainability (Venters, Kocak et al. 2021). Without being able to agree on the nature of the very topic, studies are required to produce evidence on the impact of features considered to be related to sustainability, such as the quality, lifespan, and user interaction of software. The collection of evidence produced by such studies will hopefully allow software engineers to get closer to a consensus. This research aims specifically to look at this problem from the perspective of research software.

The methodology employed by this research is one of a case study. As previously mentioned, this approach is used to test the effectiveness of the general methodology (see Figure 1.1), so that after experiments have been conducted the results can be analysed to determine any flaws and limitations in the approach to this research. The methodology can then be adjusted as appropriate to apply it further and conduct similar studies on other research software to compare the results, in order to work towards the higher-level goal of obtaining holistic answers to the research questions (§1.1), and to understand the best sustainability practices to apply to research software.

For this study, FFEA is used as the exemplar case of software created for research purposes. FFEA is a novel research application with a small user base that has been primarily developed by PhD students, some of which were not trained in software engineering. The quality of the code and the documentation is inconsistent, having been developed by different people without a coding style guide for the project. Such quality issues that can lead to software sustainability problems are a common result of ad-hoc development of research software (Baxter et al. 2012; Cohen, Katz, Barker, Hong et al. 2021). Therefore, this makes for a sufficient and apt software suite to investigate the sustainable qualities and practices of research software.

The diagram in Figure 1.1 outlines the generic methodological approach to this research, and explains where each step of the methodology is contained within this thesis.

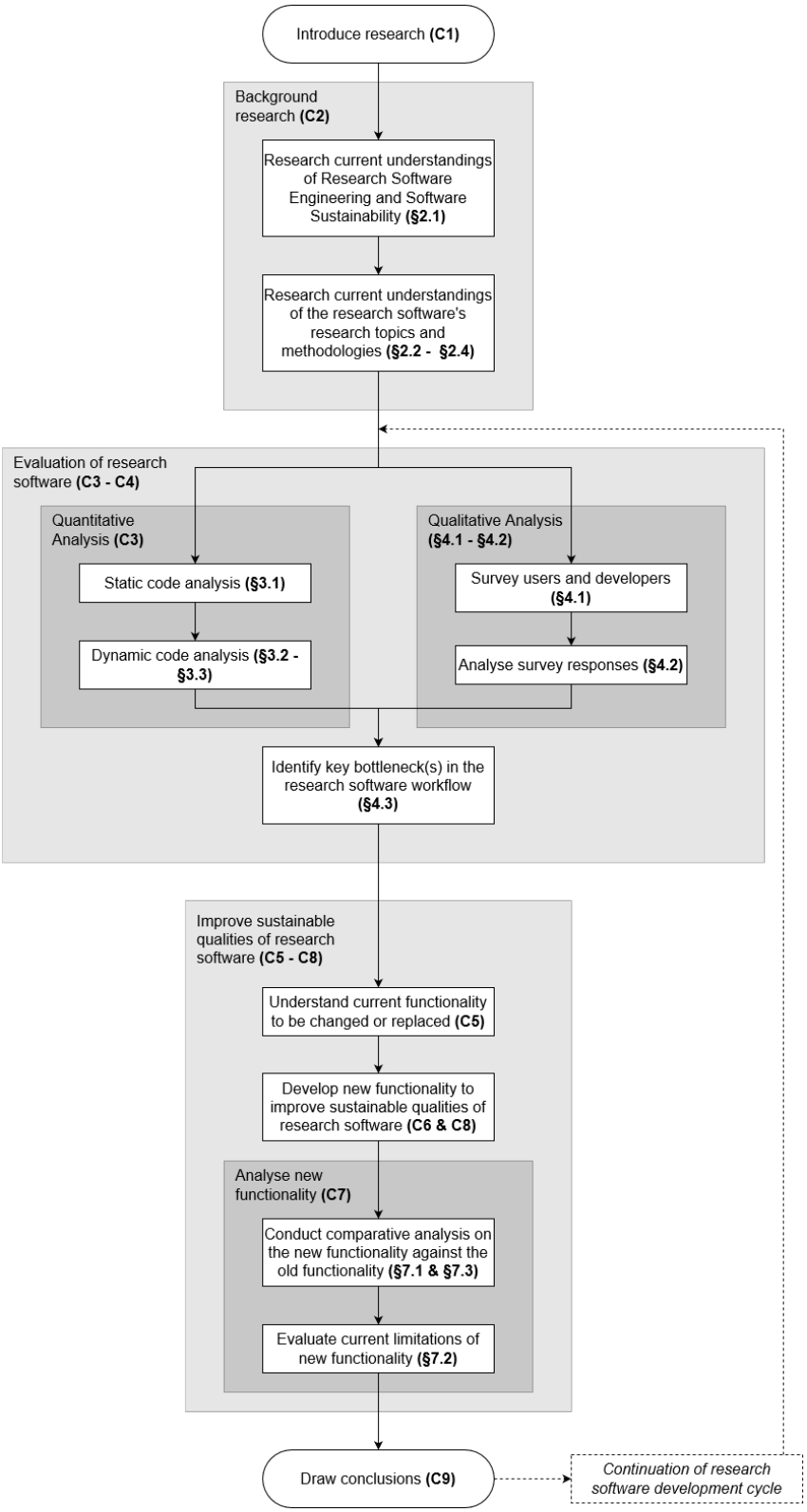

Figure 1.1: Generic methodology of this research. Each step shows the chapter (C) or section (§) where it is applied in this thesis in **bold**.

For the specifics of how this methodology is applied to the case study of FFEA, this begins with secondary research of relevant literature in Chapter 2 with a section to explore our current

understandings of RSE and software sustainability. This helps to showcase how research software is typically developed, and how flaws with these typical methods lead to certain software sustainability issues. The same chapter also introduces the concept of structural biology and molecular dynamics (the main research topics of experiments that use FFEA) and how software engineering principles have been applied to these fields.

A holistic evaluation of the current version of FFEA is performed through quantitative evaluation of the source code in Chapter 3 and qualitative evaluation of the user and developer experience in Chapter 4. The quantitative analysis applies static code analysis techniques in order to understand the quality of the written source code, and applies dynamic code analysis to understand the quality of the software's performance as it runs. The qualitative analysis employs a user and developer survey to understand the current relationship between the software and those that interact with it; to highlight both strengths and weaknesses of the research software to know what should and should not be altered. These analyses are done in order to identify key bottlenecks in the software's common workflow in §4.4, to see if they stem from issues with code performance or from issues with the design of its functionality.

After evaluating the research software to understand its sustainable qualities, the next task is to develop changes to the software that improves these sustainable qualities. In the generic approach to this research, the first step is to understand the current functionality that is being altered or replaced to understand why the software utilises it and what the current flaws are, so that in the next step a new functionality can be developed that addresses these flaws while ensuring it keeps its utility. Once a new functionality has been developed, it needs to be analysed to understand how well it has improved the research software's sustainable qualities. This is done by conducting a comparative analysis against the old functionality to measure improvements and note any potential new flaws, as well as evaluating the limitations of this functionality to decide if future development is needed and why.

In this specific case study of FFEA, the stability and ease of use of meshes during creation and simulation were key issues found from this evaluation, so new functionality was devised to attempt to improve this part of the workflow. Two ideas were devised; one aiming to improve FFEA's tetrahedral meshing, and one aiming to improve FFEA's rod meshing. Chapter 5 begins by discussing the current method of generating tetrahedral meshes for FFEA and the flaws with this method, as well as an attempt by Gravett (2022) to create a more stable method

of tetrahedral mesh generation for FFEA. Chapter 6 then explains how Gravett's meshing algorithm was further developed as part of this research to a usable state. The new tetrahedral meshing algorithm is then analysed against the old meshing algorithm in Chapter 7 in order to validate its results and test if improvements were made to the stability of objects meshed for FFEA. Chapter 8 provides a proof of concept for improving rod mesh generation for FFEA by utilising classification machine learning algorithms to detect elongated structures in molecular data.

In the final step of the methodology, conclusions are drawn in Chapter 9 to attempt to answer the research questions posed in §1.1 and by Shaw (2003). By doing this the generic methodology can be evaluated to see its potential in being applied to other research software, and what alterations and additions can be made to the method to improve its thoroughness and applicability. These conclusions also presents an overview of the specific application of the methodology to FFEA, to clarify how this method has highlighted sustainable quality issues with the software and the effectiveness of changes made to its functionality during this research. This overview will help future development as the development cycle of the software begins again.

Chapter 2

Background research

Before conducting research on making improvements to FFEA, this chapter discusses relevant literature. It begins by detailing researchers' experiences and studies on research software engineering (RSE), as a generalised look into issues with research software (especially in relation to sustainability) may provide useful insights on how to analyse and improve such software. Structural biology and molecular dynamics are then presented as the research topics which FFEA as research software is designed for, with literature to provide brief explanations of them to the reader, how computer science and software engineering assist researchers of these topics, and how computational limitations can impact this assistance. This chapter then closes with a more detailed explanation of the FFEA numerical method, following literature on the finite element method which it is based on. In regards to the general methodology as outlined in Figure 1.1, this chapter makes up the "Background research" stage.

2.1 Research software engineering and software sustainability

2.1.1 Research software engineering

As computing tools and digital methods have increased in general usage, academia has been increasingly utilising software for research practices, taking advantage of the increasing processing power of machines to produce more precise simulations of real-world systems for experimentation (Hansson 2015). At the Software Sustainability Institute's 2012 Collaborations Workshop, a discussion began around issues specific to the development of software for academic purposes (Baxter et al. 2012). The key issue noted at this workshop was a lack of a professional des-

ignation for the development of this type of software, requiring academics to create software themselves for their research, regardless of their understanding of software engineering. The attendants of this discussion proposed the term "research software engineer" (RSE) as a career title for this form of software development.

Research software engineers fill in the need for software engineers that also have an understanding of research practices and the specific domain of the research topic (Baxter et al. 2012). They must understand how to read, and sometimes contribute to the writing of, academic literature, and they must employ scientific methods to develop and validate software if it is to be appropriate for use in scientific research. Software engineers with this knowledge base were uncommon at the time due to this lack of a designated career path, and of institutional representation. Since coining this term however, it has been widely accepted by the software engineering community, leading to the creation of new workshops, conferences, career positions, and research associations for the field (Hettrick 2016).

2.1.2 Research software sustainability

Research software is often primarily or even solely focused on the goal of creating an output, such as answers to given research questions, with less attention focused on the quality of the software itself (Baxter et al. 2012; Cohen, Katz, Barker, Haines et al. 2018; Carver et al. 2021). This leads to unsustainable software that is not maintained and rapidly decays (Souza et al. 2019) (in this thesis, sustainability refers to aspects of software, not aspects of ecological impact). This is often because the developers are researchers who are not software engineers, and therefore have little-to-no training in relevant tools and practices (Baxter et al. 2012; Cohen, Katz, Barker, Haines et al. 2018). In recent years, there has been an increase in higher education integrating sustainable practices into course material in response to the United Nations's sustainable development goals (UNESCO 2023), but this is of a limited scope and courses rarely address sustainable practices for fields focusing on technical skills such as computer science (Venters, Capilla et al. 2022).

These issues tend to lead to a cycle of "reinvention of the wheel", where software is created and disposed of frequently for similar research tasks. Figure 2.1 elaborates on how this cycle typically occurs. When researchers require software for their problem domain, they will create a new piece of research software with development focused on solving this problem. When the software produces results for the research it was designed for, its immediate use is now fulfilled,

so further development and maintenance of the software ceases, and eventually the software decays to an unusable or inaccessible state. Then, when a similar research topic begins that could have made use of this now-decayed software, the researchers have no choice but to make a new pieces of research software from scratch, and thus the cycle of decay begins anew.

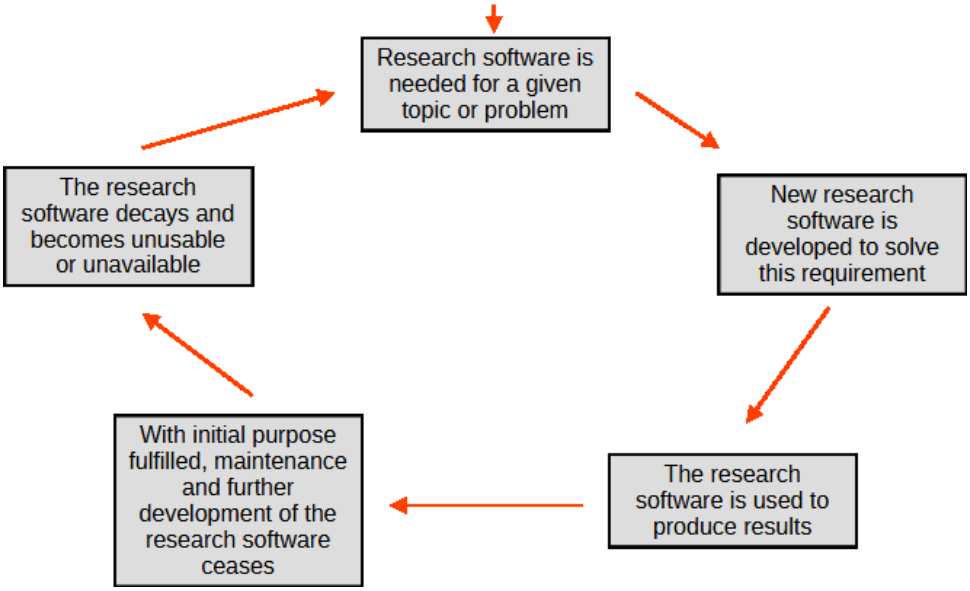

Figure 2.1: The cycle of reinventing research software. Software decay is further exacerbated by a lack of software engineering training for developers of this software, and by a lack of sustainability standards for developers to reference.

What makes a piece of software sustainable is subjective, and different users and organisations will stop using software for different reasons. There is currently no consensus on the definition of software sustainability, the various researchers that have tried to define this present contradictory ideas (Venters, Kocak et al. 2021). With this lack of a strict definition, to gain an idea of sustainable practices for RSE, Souza et al. (2019) produced a survey for RSEs asking them which features of research software are important to keeping it sustainable. They noted some consensus on the meaning of sustainability, but varying views on the best ways to achieve it. Overall software quality and connection to a community are the more common points mentioned, however. There were fourteen key features mentioned by those surveyed, the frequencies of which are shown in Table 2.1. Having the software be well documented, openly available, actively maintained, with readable code, and plans for future upkeep were the features most-often mentioned.

FREQUENCIES OF THEMES PER STUDY PHASE

	Phase 1	Phase 2	Total
Intrinsic sustainability			
Documented	8	6	14
Testable	5	2	7
Readable	5	7	12
Modular	2	3	5
Standardized	2	-	2
Useful	2	2	4
Scalable	1	1	2
Extrinsic sustainability			
Openly available	6	6	12
Shared/co-owned	3	4	7
Resourced	6	4	10
Actively maintained	6	7	13
Independent of infrastructure	1	6	7
Supported	4	-	4
Version-controlled	-	4	4

Table 2.1: Results of the frequency of features of sustainability mentioned by participants in the interviews by Souza et al. (2019). There were nineteen participants in total. Three were female and sixteen were male. Experience in RSE ranged from eighteen months to twenty years. Phase 1 consisted of nine developers at a single UK university. Phase 2 consisted of ten developers from four UK institutions, one of which is the same university as in phase 1.

Venters, Kocak et al. (2021) note that attempts to define software sustainability tend to focus on the technical side of software development, such as the maintainability and extensibility of the software itself. However, they put forward that it has been argued that software sustainability should apply to "*both the system and its wider contexts*". This is corroborated by the survey results produced by Souza et al. (2019), with extrinsic features of sustainability receiving roughly the same amount of attention from participants as intrinsic sustainability features. There is some difference between what the two groups of researchers consider sustainability features intrinsic to the software itself, with Souza et al. (2019) considering maintenance to be extrinsic, while Venters, Kocak et al. (2021) discuss maintenance as an intrinsic factor. Regardless of this difference, there would still be roughly-equal mentions of intrinsic and extrinsic sustainability features by the participants of the survey conducted by Souza et al. (2019).

While not specific to RSE, Theisen et al. (2017) discuss how to write quality research papers about software engineering, revisiting an earlier paper by Shaw (2003), who posed three questions that software engineering papers should answer to be indicative of well-presented results:

1. **What, precisely, was your contribution?** - State what questions the software can answer, and why readers should care about the software.

2. **What is your new result?** - State what new knowledge the software contributes for the

reader to use, what previous work is the software built on, and is the software a superior alternative to existing similar software and why.

3. **Why should the reader believe your result?** - State the standard that should be used to evaluate claims made and give concrete evidence to show that the result satisfies these claims.

Theisen et al. (2017) replicated the study Shaw (2003) performed to analyse the abstracts of papers submitted to the International Conference on Software Engineering in order to investigate the perceptions of the review committees based on these questions. They noted in papers with a produced artefact (for example a technique or tool) reviewers held negative opinions towards these papers if they did not discuss the availability of the artefact or any data utilised. Many publications have since adopted open data policies to address this, where the publisher will recommend that data used in given research is publicly archived, however it is rare for publications to *require* open data before research can be published (Jackson 2021).

The most important factor found by Theisen et al. (2017), however, is the researcher's choice of validation technique. Submissions of novel approaches with minimal evidence for their results have a lower acceptance rate. As this thesis covers the topic of software engineering, these ideas will be considered throughout to ensure good quality information is presented, as the quality of documentation is important to the sustainability of software (Souza et al. 2019). If this research is to assert the importance of software sustainability, then it must be held to the same standards.

Some commonalities arise in the literature. Cohen, Katz, Barker, Haines et al. (2018), Jiménez et al. (2017) and Souza et al. (2019) mention the importance of community to RSE, promoting some form of possibilities of collaboration, whether direct or through forums. Jiménez et al. (2017), Shaw (2003), Souza et al. (2019) and Theisen et al. (2017) discuss the importance of the availability of the software. Jiménez et al. (2017) and Souza et al. (2019) specifically mention the value of software being open-source (i.e the source code is openly available for users to inspect and modify), and how that increases the discoverability of the software to cement its future usage. Jiménez et al. (2017), Shaw (2003) and Theisen et al. (2017) mention the importance of transparency (i.e. evidencing what actions have been performed to obtain results), although have different discussions about how to make software transparent. Jiménez et al. (2017) focus on the people involved and their responsibilities, and Shaw (2003) and Theisen et al. (2017)

discuss how developers should ensure to answer why their claims and results should be believed through concrete evidence.

The questions posed by Shaw (2003) essentially restate the scientific method but apply it to software. In practice, these questions are attempting to hold the design of a piece software to the same standards as the scientific results it produces, to ensure its reliability and validity when presented.

Sustainability in regards to software is a contested issue with no clear definition or strategies. Reviews and surveys have helped to qualitatively evaluate this problem and allow researchers to identify aspects that could be considered software sustainability which are important to members of the scientific community, but consensus cannot be achieved from this to develop a list of software sustainability best practices. In the meantime however, RSEs can consider these subjective views on sustainability to develop and implement strategies for improving and maintaining sustainability, and then evaluate the effectiveness of their strategies in order to validate and refine these ideas.

Software sustainability has been important to research software engineering from its inception, as a key issue the field aimed to address was the frequent decay of academic software due to the unsustainable environment they were typically developed in (Baxter et al. 2012). Research software also has its own specific sustainability requirements in that the needs of its research topic will evolve over time, likely in ways too complex to fully deduce from the software's inception, so research software must be created in a sustainable manner from the start so that it can be actively maintained to keep updated and stay relevant (Carver et al. 2021).

Cohen, Katz, Barker, Hong et al. (2021) have stressed the importance of sustainability to RSE in their framework "the four pillars of research software engineering". The general aim of this framework was to highlight important elements to provide support for the field, with the four central ideas being software development, community, training, and policy (see Figure 2.2). Software sustainability is directly mentioned as part of the "software development" pillar, emphasising the importance of ensuring research software is of high quality for valid research and is maintainable for future use as the needs of the research topic change over time.

While the other three pillars do not explicitly mention sustainability, they are designed to create a stable and supported career environment for RSEs. This is done by highlighting the import-

ance of communities for RSEs to discuss research and the needs of their field, the importance of training to bring new RSEs into the field in such a way that all are prepared for their career, and the importance of advocating policies in research-based intuitions to understand the value of RSEs and ensure they are properly supported in their role. These points are put up for consideration as the authors argue that sustainable software requires development in a sustainable environment (Cohen, Katz, Barker, Hong et al. 2021). This point is also backed by Carver et al. (2021) who states that a stable work environment with adequate training and support for its RSEs is necessary to provide guidance through best software development practices to create sustainable software, and that it is necessary for ensuring the needs for software maintenance is met to keep it updated.

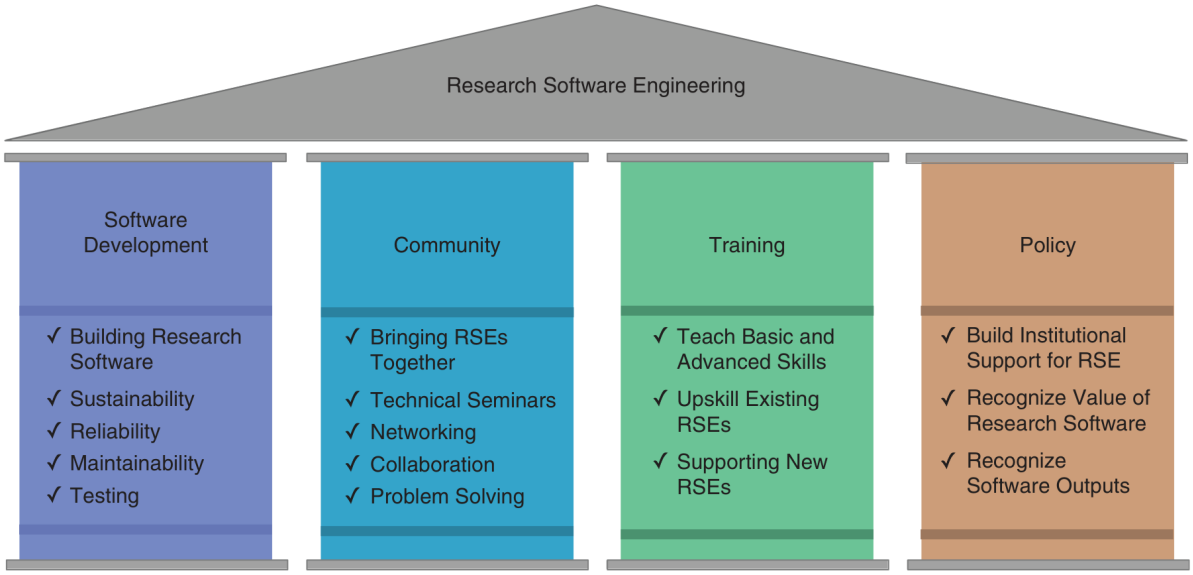

Figure 2.2: The four pillars of research software engineering. Diagram by Cohen, Katz, Barker, Hong et al. (2021).

Much of the work centred around research software sustainability has been very general to the needs and practices of the field as a whole, which is understandable due to it being a young field with a focus on developing stable career paths for practitioners. Specific examples of RSE cases would have been helpful to include in this literature review for this research, however little of relevant substance was found from looking through the resources of research groups with RSE as a key topic, such as the Software Sustainability Institute (2024), the Workshop on Sustainable Software for Science: Practice and Experiences (2017), and Better Scientific Software (2024). High-Performance Computing (HPC) groups such as Archer often publish case studies to showcase their capabilities and applications (Archer 2024), but it is not yet

common practice to discuss the software engineering challenges of running research software on HPC, and how RSE principles were applied to achieve this.

2.1.3 Open issues in research software sustainability

Now that the field of RSE has been accepted by the software engineering community, it is the opinion of the author of this thesis that RSEs need guidance on the best practices to ensure robust and sustainable research software for their specific field. As previously mentioned by Baxter et al. (2012), a research software engineer must understand general research practices *and* the specific domain of the research topic in order to appropriately apply scientific methods correctly to the development and validation of their software.

But how does a research software engineer develop these sustainable best practices beyond the general for their specific research domain, and how do they do so when software engineering in general lacks a consensus for the definition of software sustainability? The authors of the four pillars framework themselves have questioned how generally applicable their framework is when faced with different research contexts, and that understanding this is necessary to completing their framework (Cohen, Katz, Barker, Haines et al. 2018). Without studies conducted on cases of research software for specific fields and topics on improving their sustainability, there is no evidence to base definitions and best practices on through critical analysis of employed methods to test their effectiveness. This thesis aims to be one such study.

Even though we cannot define software sustainability for the whole field of software engineering or RSE, we must choose and stick to a definition in this research for clarity to the reader. Therefore, in this thesis, software sustainability is defined as **the qualities of software which impact its usability, maintainability, and longevity**. This definition is chosen to encompass quality features that affect the experience of a software's users and developers, as well as affects on the lifespan of the software itself, attempting to gain a holistic understanding of the impact of these affects. The term "usability" is included in this definition to probe the various factors on how the software is interacted with, such as user experience, workflows, and accessibility concerns. "Maintainability" is included to drive analysis of the developer experience, and how the quality of the software changes with time and with updates to the software. Finally, "longevity" is included to to cover aspects that impact how long the software will remain in use, such as user retention, software decay, and the ease of access to the source code, the executable,

and documentation.

2.2 Sustainability of computational solutions to structural biology

2.2.1 Structural biology and molecular dynamics

Structural biology is the study of biological materials and their physical and chemical properties (Liljas et al. 2016). The analysis of material properties of biological matter has allowed researchers to study biological objects in more depth and with clearer distinctions. For example, the discovery of nucleic acid occurred during experiments by Freidrich Miescher (Liljas et al. 2016; Pal 2019). While attempting to study proteins in leucocytes (i.e. white blood cells), Miescher noticed an unknown substance precipitating from the cells' nuclei. Upon examination, the substance was found to not have the chemical structure of proteins (instead containing large quantities of phosphorous), and was thus found to be a different substance (Pal 2019).

Miescher's discovery has been invaluable to the field of biology, with DNA and RNA both being classes of nucleic acids which are now highly studied. However, as the study of DNA progressed it became difficult to continue without the aid of machines to perform some of the complex computations required for such techniques as DNA sequencing to describe the chemical structure (Staden 1979). The declining cost of computer components across the latter half of the 20th century gave more DNA sequencing researchers access to computational solutions. This allowed for the development of faster, more efficient DNA sequencing strategies, and this is but one method of applying computational solutions to structural biology problems.

Molecular dynamics is a computational method for analysing atoms and molecules in simulated motion in order to study their structures and dynamics (i.e. the effects of forces on motion). A computational approach allows researchers to investigate molecular physics without the need of setting up difficult and expensive real-world experiments. While *in silico* experiments cannot replace the validity of real-world experimentation, this allows a researcher to do preliminary studies on a theory or hypothesis before making more costly commitments (Zhou and Liu 2022).

As computer science and technology have advanced, molecular dynamics has benefited from more powerful and efficient computing resources to allow for experiments of greater complexity, such as with larger objects (or larger numbers of objects) at longer timescales (Zhou and Liu

2022). As the field of computer graphics has evolved, graphical user interfaces (GUIs) have allowed researchers to visually observe their data and results (Pal 2019; Gravett, Cocking et al. 2021; Zhou and Liu 2022). The results of molecular dynamics experiments can be visualized with a number of representations to assist a researcher in results analysis, and to communicate experimental findings to a wider audience. Some of these representations include line representation (used for showing chemical structures and atomic bonds), ribbon representation (used to visually simplify the structure of chain molecules, such as nucleic acids and proteins, which fold to form 3D structures), and surface representation (used to represent surface interactions and molecule geometry) (Pal 2019). Figure 2.3 shows examples of these representations.

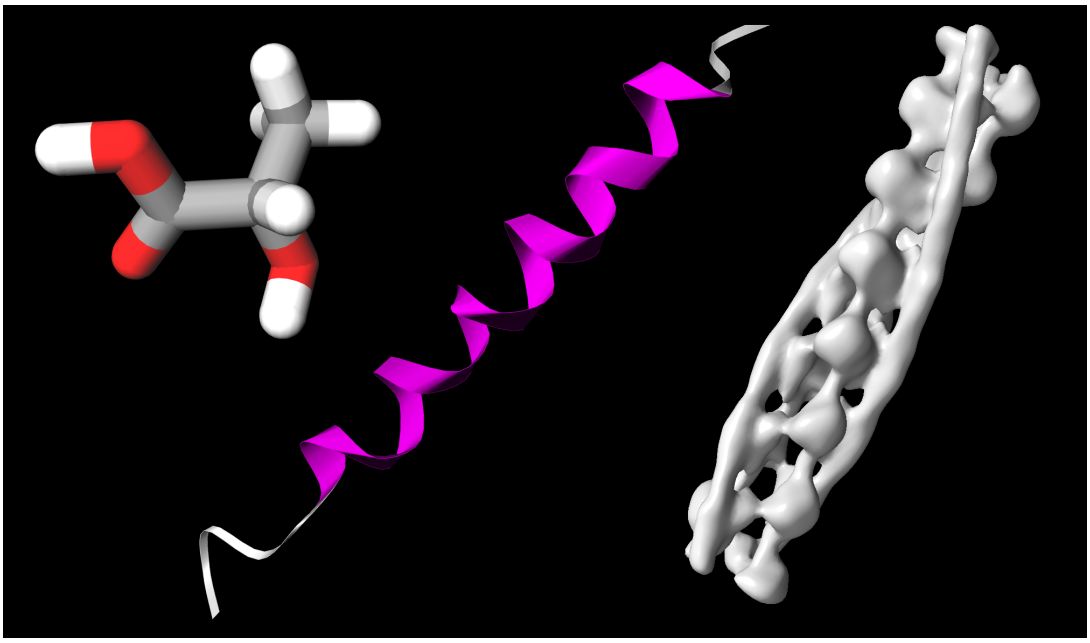

Figure 2.3: Examples of molecular dynamics visual representations. From left-to-right, lactic acid in line representation (visualized in MolView), glucagon in ribbon representation (visualized in MolView), and actin-tropomyosin in surface representation (visualized in UCSF ChimeraX).

Visualization for molecular dynamics has also assisted researchers in building models for simulated experiments (Gravett, Cocking et al. 2021). UCSF ChimeraX (Pettersen et al. 2021) is a molecular graphics tool for analysing molecular structures, and a common use is to fit surface or mesh models to atomistic models. ChimeraX also supports third-party extensions, which can be used by a researcher to develop model building tools for their experiments. Blender (Blender Online Community 2023) is a general-purpose 3D modelling software, however it too supports third-party extensions, one of which being a GUI for the molecular meshing tool GAMer (Lee et al. 2020). GAMer converts molecular structure datasets (such as from microscopy or atom-

istic data) into geometric meshes, and with the Blender extension BlendGAMer a user can interactively analyse and edit these meshes.

2.2.2 Computational solutions

Molecular dynamics is a computational method, and as such is limited by computational resources. As there is an increase in the size of input data, the amount of calculations performed, and the length of experiments, the computational cost of these experiments will also increase, and performance limits will eventually be reached. Conventional molecular dynamics simulations usually have an upper length scale limit of around 100nm, and an upper timescale limit of around 100ns (Zhou and Liu 2022). The length limit prevents molecular dynamics from simulating mesoscale objects (length scales of approximately 100-1000nm), which includes many desirable biomacromolecules such as proteins and nucleic acids. The timescale limit prevents molecular behavioural analysis, as some molecular actions (such as the molecular relaxation of some systems) take longer than 100ns. While these are not hard limits, and some molecular dynamics methods have been able to move into these difficult time/length scales (see §2.4), it is still an on-going process to extend the range of scales that molecular dynamics can simulate (Gravett, Cocking et al. 2021).

Molecular dynamics, therefore, requires computational solutions for maintenance, improvements, and supplementary tools. This section will discuss what computational solutions have been attempted to directly further the capabilities of molecular dynamics.

A case for the impact of optimisation on molecular dynamics can be found in the work of Stone et al. (2013). In it, they adapt the molecular dynamics software VMD (Humphrey et al. 1996) to run on GPU-accelerated petascale supercomputers, with the goal of being able to visualize the dynamics of large biomolecular systems of interest to the field of medicine, such as viruses, which are in the mesoscale and unreachable through a personal computer (as of the time of their writing). This was inspired by earlier successes such as that of D'Agostino et al. (2014) using CUDA to speed up the generation of molecular surfaces (for their dataset the speed-up was by a factor of 88). Stone et al. used the supercomputer Blue Waters (Supercomputing Applications 2023) to accelerate VMD, and with it were able to render photo-realistic atomic simulations of HIV-1's capsid (64 million atoms) and the structure of a chromatophore (2.4 million atoms).

A similar project was conducted by Peng et al. (2020) whereby they optimised the molecular

dynamics software AMBER (Kollman et al. 2023) through parallel acceleration of the CPU and Intel's Xeon Phi MIC coprocessor[1] (Rahman 2013) to use on the supercomputer Tianhe-2 (Liao et al. 2014). They noted that an issue with molecular dynamics is the high amount of computing resources necessary, which hampers current technology's ability to process simulations of systems with millions of atoms on the microsecond-level. Their main goal was to work towards a solution for this. With their frameworks they were able to speed up the original software by factors between 25 and 33, beginning work towards more reasonable times for producing atomistic simulations of macromolecules.

Optimisation has been a key focus of the computational solutions for improving molecular dynamics, with the objective being to reduce the amount of computational resources required per element of a molecular dynamics system, and therefore increasing the timescale of simulations and the size of systems which can be simulated. This is not, however, the only computational approach to improving molecular dynamics.

Machine vision has been used to improve the analysis of molecular image data, especially data from microscopy (Gravett, Cocking et al. 2021). These tools use algorithms to process molecular image data by identifying molecular components, automating some of these time-consuming and error-prone manual tasks (Al-Azzawi, Ouadou, Tanner et al. 2019; Wagner et al. 2019). Theoretically, beyond just image classification, such machine vision tools could be used to automate the setup of molecular dynamics simulations by analysing input data, and processing this data based on the results of classification and identification (such as generating meshes of input molecule data) to be passed into simulations (Gravett, Cocking et al. 2021).

Similarly, artificial neural network techniques have been developed to predict structural properties of molecules. Graph neural networks are one such technique used in this field (Wang et al. 2022), which processes irregularly-shaped data encoded as a graph, such as molecular structures. This technique has been successfully applied to problems in the simulation, screening, design, and synthesis of materials (Reiser et al. 2022).

One approach to improving the capabilities of molecular dynamics simulations themselves is to simplify a complex system with a coarse-grained model. A coarse-grained model reduces the number of discrete data points by grouping them into single elements (or "beads") of a dataset,

[1] Unfortunately the Xeon Phi was discontinued in July 2020, therefore MIC parallelisation is no longer an option moving forward.

with the intent of that element reacting to stimuli as the combined data would (Kmiecik et al. 2016; Liu et al. 2021). The traditional use of coarse-graining in molecular dynamics is to reduce the complexity of atomistic models by grouping atoms together in single elements as continua (i.e. continuous matter) (see Figure 2.4), thus significantly reducing the number of points in a molecular dataset to apply calculations to (Kmiecik et al. 2016).

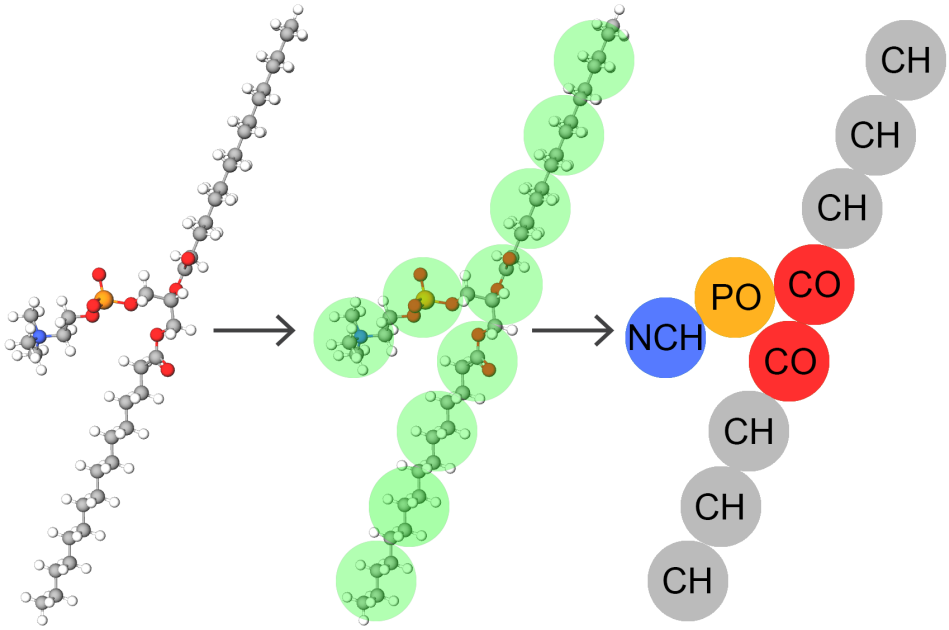

Figure 2.4: Example of a coarse-graining process on an all-atom model of dipalmitoylphosphatidylcholine (DPPC). From left-to-right, the original all-atom model of DPPC containing 130 atoms, the selection of coarse-grain beads, and the new coarse-grain model of DPPC containing 10 elements.

Coarse-grained models are one method that allows molecular dynamics simulations to reach into the mesoscale to simulate macromolecules. Liu et al. (2021) presents a case where they were able to produce molecular dynamics simulations of mesoscalar graphene structures (which are made up of layers of hexagonal lattices of carbon atoms) by coarse-graining the models. They noted that all-atom molecular dynamics models struggled to simulate large graphene structures with good performance, limiting the ability to analyse a material of current significant interest to material science. They also noted that existing coarse-graining models for graphene had various flaws, such as how square coarse-graining models cannot account for interactions between layers of graphene, and how hexagonal coarse-graining models still had too high a computational cost for modelling mesoscalar graphene structures, as it could only reduce the number of data points by a fixed factor of four. Their solution was to create a multilayer coarse-graining model which first coarsens the atoms across the planes of graphene, and then coarsens those planes to create

thicker layers of coarsened graphene models. Compared to all-atom models, these multilayer coarse-grained graphene models were able to produce simulations of a recorded approximate 1-2 orders of magnitude increase in length scale, and an approximate 3-4 orders of magnitude reductions in computational time cost.

The coarse-graining methodology is not without its limitations. Coarse-graining a model of a physical object requires geometric calculations, and as noted with the example of graphene just discussed (Liu et al. 2021), sometimes these geometric calculations are incompatible with, or unsuitable for certain molecular structures. The wrong coarse-graining model applied to an object can alter the structure of the object in undesirable ways, or it could simply be an inefficient model that does not reduce the computational complexity of the model enough for quality experimentation.

Coarse-graining will also always sacrifice accuracy, as it is a simplification of a more complex model. For example, coarse-graining reduces the degrees of freedom of an object (which in mechanics is the number of motions in a body that may vary independently). A reduction in degrees of freedom will decrease the computational cost as the number of calculations performed at a given time decreases also. However, this can have impacts on certain physical properties such as thermodynamics, as reducing the degrees of freedom affects the entropy of a system, and thus will introduce inaccuracies into calculations that use entropy. This accuracy reduces as the coarseness of a model increases, and therefore there is a balance when creating coarse-grained models between increasing computational efficiency as much as possible, and decreasing the accuracy of certain physics calculations as little as possible (Kmiecik et al. 2016).

Methods for alleviating the limitations of the coarse-graining approach are being developed however. For example, Pan et al. (2023) noted that using coarse-grained models for mesoscalar graphene simulations caused a significant inaccuracy in the total kinetic and potential energy in the system. The researchers attempt an alternative modelling method with atom sampling in order to mitigate these inaccuracies while still maintaining the lower computational cost beneficial from coarse-graining. In atom sampling, an element of the model is made of only a single sampled atom in that bead, and has the mass of only that single atom, rather than the element being the sum total of the atoms present in that bead. A Tersoff sampling model (the Tersoff potential is an interatomic energy potential used to describe the structural and energy properties of carbon (Tersoff 1988)) based on the atom sampling method was then developed

to predict mechanical properties of the system. This method was tested against a Tersoff coarse-grained model to see if the atom sampling method or coarse-grained method produce results more accurate to a Tersoff all-atom model. The researchers recorded that the atom sampling method was able to reproduce more accurate mechanical behaviours than the coarse-graining method for graphene, including the stress-strain relationships and the deformation of the material.

2.2.3 Sustainability of computational solutions to structural biology – key challenges

The previous subsection cites seven pieces of research on a specific computational solution to a structural biology problem (Stone et al. 2013; D'Agostino et al. 2014; Al-Azzawi, Ouadou, Tanner et al. 2019; Wagner et al. 2019; Peng et al. 2020; Liu et al. 2021; Pan et al. 2023). This presents a handful of cases to briefly consider their approaches to the sustainability of their research software, in line with the ideas presented in §2.1. This will focus on sustainability themes with objective answers, such as if open access is provided, as the sustainability themes with subjective answers such as source code readability cannot be reliably answered by just a single person.

All papers satisfied the questions posed by Shaw (2003), outlining their contribution to a solution to a particular problem, clearly stating their results with comparisons to previous methodologies, and evidencing these results. This suggests that modern paper review processes tend to require or request this information before research can be considered for publication. This seems in line with the findings of Theisen et al. (2017), stating that submitted research with poor validation have a low acceptance rate.

Out of these seven research papers, only Al-Azzawi, Ouadou, Tanner et al. (2019) provided open access to the source code. The exact code for the tool in this paper has not been maintained since publication and only provides minimal documentation, but their research has evolved into the development of subsequent tools such as DeepCryoPicker (Al-Azzawi, Ouadou, Max et al. 2020) and Auto3DCryoMap (Al-Azzawi, Ouadou, Duan et al. 2020). Many of the other papers explained how their computational solution worked, such as by providing mathematical explanations or pseudocode, but they did not allow access to the source code itself for readers to understand exactly how their methods were implemented. This in turn means that most

of these tools are undocumented, and it cannot be verified whether or not they are being maintained. The one exception in this group of research papers is the work done by Wagner et al. (2019), which provides a link to the closed-source software (only accessible to the copyright holders) with detailed documentation and evidence of further development and maintenance after the publication of this paper. The availability of data in these papers is a little better, with D'Agostino et al. (2014), Al-Azzawi, Ouadou, Tanner et al. (2019) and Wagner et al. (2019) stating where readers can access the data used for their experiments. Publication open data policies may be improving access to data, but this loose requirement still means many papers will not provide this information, and a lack of similar policies for source code means that there is currently little incentive for researchers to provide this. A change in policy to require open-access to all information may not be easy or viable however, as it will have to contend with proprietary software where information is protected by copyright.

2.3 Finite element method

The finite element method (FEM) (sometimes referred to as finite element analysis or FEA in practice) is a numerical method for approximating mathematical models through enabling approximate solutions to complex differential equations. These computational models are simplifications of the original continuous data and are made up of a finite number of discrete elements connected at shared points or "nodes" (akin to vertices in a lattice) (Musa et al. 2014; Logan 2022). This discretized model allows an analyst to find solutions to physics problems that are too difficult or costly to solve with real-world data and experiments, as the model only needs to solve equations that lie at these discrete elements, rather than for a continuous system.

The following paragraphs explains the process of FEM. For ease of explanation, it will be assumed this is in relation to the stress and strain of a structural problem.

The FEM process begins by dividing the problem domain into small discretized elements (Musa et al. 2014), with the specifics of element size and connections being decided by the analyst to suit their problem (similar to coarse-graining for molecular dynamics discussed earlier in §2.2.2). This creates an approximation of the domain of the problem (i.e. a discretized model), which can be formatted as a mesh of the structure. In an FEM problem there will be an unknown field variable to be evaluated, which in this case is the displacement. After a mesh model has been created of the structure or system to be analysed, the analyst then needs to choose a

displacement function; a simple polynomial equation used to describe the displacement of a given element based on the known values at the nodes of that element (Musa et al. 2014; Logan 2022). The same displacement function is used for each element of the mesh for a given problem.

With this displacement function the analyst then needs to define the relationship between the displacement and the other mechanics of the system (Musa et al. 2014; Logan 2022). In this case, the analyst defines the relationship of strain and displacement, and stress and strain (Logan 2022). For a one-dimensional deformation problem, the relationship between strain and displacement would be defined by

$$\epsilon_x = \frac{du}{dx}, \tag{2.1}$$

where in the x direction, ϵ_x is the strain, and u is the displacement. The relationship between stress and strain would be defined by Hooke's law for normal stress as per

$$\sigma_x = E\epsilon_x, \tag{2.2}$$

where σ_x is the stress in the x direction, and E is the modulus of elasticity.

These relationships define the material behaviour of the system, and are used to derive the equations for each finite element. For this case, the analyst derives the matrix and equations for element stiffness (Musa et al. 2014; Logan 2022). These equations show the responses of an element to a given load, and is defined by

$$[k][d] = [f], \tag{2.3}$$

where $[k]$ is the element stiffness matrix, $[f]$ is the vector of the forces at each node of an element, and $[d]$ is the vector of unknown displacements or nodal degrees of freedom.

The equations derived from the previous step are then assembled to obtain the global equations (Musa et al. 2014; Logan 2022) (sometimes called the total equations (Logan 2022)). In this case, the global equation is defined by

$$[K][D] = [F], \tag{2.4}$$

where $[F]$ is the vector of global nodal forces, $[K]$ is the global stiffness matrix, and $[D]$ is the vector of unknown *and* known displacements or nodal degrees of freedom.

With the global equations the analyst can now solve for the unknown values in $[D]$ (Musa et al. 2014; Logan 2022), known as the primary unknowns. Equation 2.4 can be expanded to

$$\begin{bmatrix} K_{11} & K_{12} & \dots & K_{1n} \\ K_{21} & K_{22} & \dots & K_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ K_{n1} & K_{n2} & \dots & K_{nn} \end{bmatrix} \begin{bmatrix} D_1 \\ D_2 \\ \vdots \\ D_n \end{bmatrix} = \begin{bmatrix} F_1 \\ F_2 \\ \vdots \\ F_n \end{bmatrix}, \tag{2.5}$$

where n is the total number of displacements or nodal degrees of freedom, which shows a set of simultaneous algebraic equations. This linear system of equations can be solved for D via an elimination or iterative method (Logan 2022).

Once the primary unknowns are derived, the analyst can then obtain important secondary unknowns (Musa et al. 2014; Logan 2022). In this case for a structural problem, this would be to solve for the element strains and stresses. This can be solved by using the displacements determined by Equation 2.5 and (for one-dimensional problems) computing them from Equations 2.1 and 2.2. With all the desired unknowns calculated, the analyst can then analyse and interpret the results of their finite element analysis in order to understand the solution to the given problem (Logan 2022). In the case of a structural problem, this might be determining the locations of large stresses and strains within a structure, so that a structural engineer can understand mechanical load limits, and perhaps how to improve the structure to increase the maximum loads.

The ability to make mathematical models of physical objects and matter makes it a very useful tool for structural and material property analysis, such as the analysis of stress, vibrations, buckling, impacts, heat transfer, force fields, and fluid flow (Logan 2022). The results of these analyses can easily be visualized onto representations of the objects or systems in question, as each element can be labelled and graphed with the values of calculated unknowns to show the distribution of these values (see Figure 2.5 for examples).

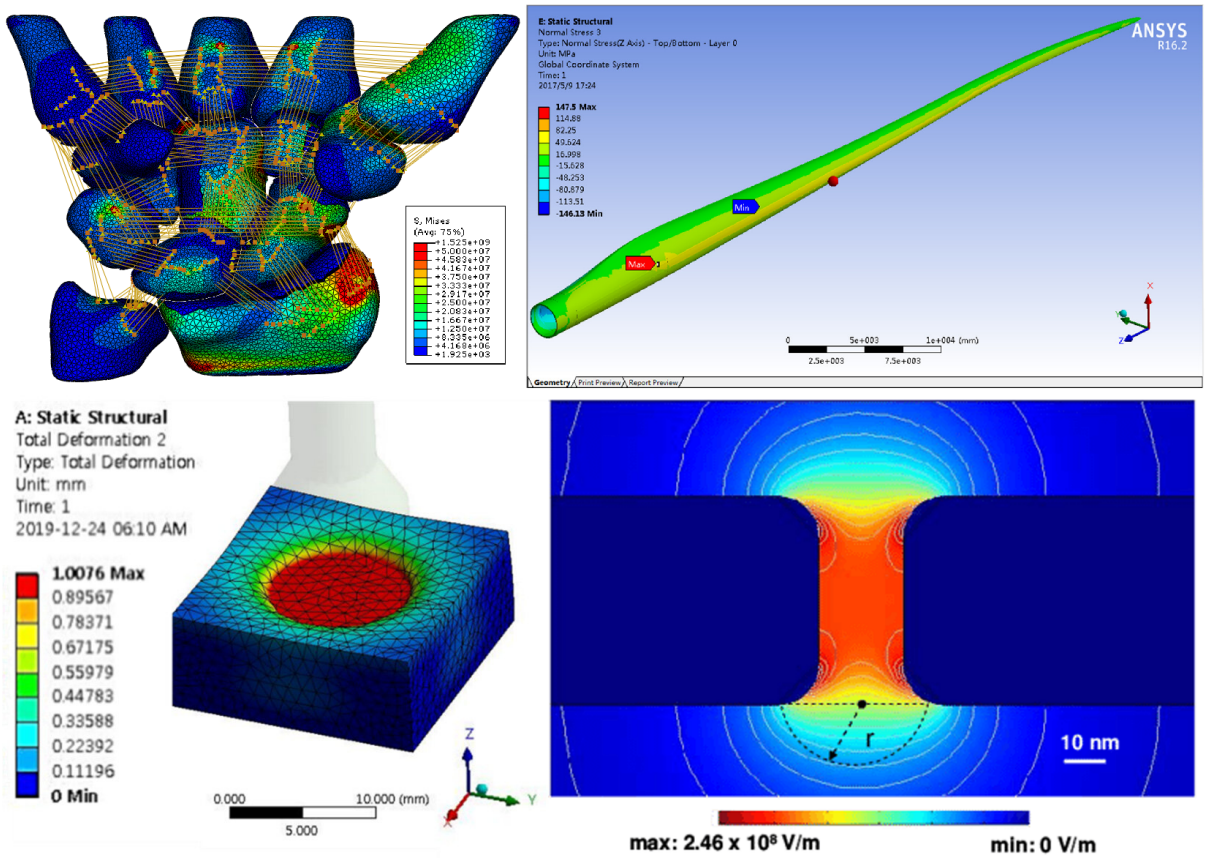

Figure 2.5: Example visualizations of finite element analysis results. Left-to-right and top-to-bottom: Finite element model of the human wrist showing von Mises stresses (Gíslason and Nash 2012). Finite element model of a wind turbine blade showing tensile and compressive stresses (He et al. 2017). Finite element model of smart magnetorheological fluid immersed into porous polyurethane foam to calculate repulsive forces during deformation (Park et al. 2020). Finite element analysis simulation of the electric field in and around a 20nm gap between electrodes (Hegg and Lin 2008).

2.4 Fluctuating finite element analysis software suite

Fluctuating finite element analysis (FFEA) is a numerical method that uses continuum mechanics to simulate mesoscopic biological macromolecules (such as proteins and nucleic acids) by combining the finite element method with stochastic thermal noise. This method was developed with the goal of meeting a need for analysis of these macromolecules as advances in biophysical tools allowed for an increased collection of information about the structure and dynamics of matter at this length scale. Other techniques for analysis of the mesoscale designed before this treat macromolecules as rigid bodies, and thus cannot be used for simulating highly dynamic systems such as molecular motors (Solernou et al. 2018).

The fluctuating finite element method introduces this stochastic thermal noise as it is designed

to simulate objects at a small enough length scale that thermal fluctuations are non-negligible, and will cause deformations in the viscoelastic materials that are used to represent biomolecules (Richardson et al. 2014). The general formula used to describe the motion of the continuum model, which defines the material behaviour of an FFEA system, is defined by

$$\rho \frac{du}{dt} = \nabla \cdot \sigma + f + \nabla \cdot \pi, \tag{2.6}$$

where ρ is density, u is displacement, t is time, σ is continuum stress, f are additional forces from electrostatic or van der Walls interactions in the system, and π is stress due to thermal noise.

The FFEA method can be utilised by users with the FFEA software suite (Cocking et al. 2023). This software suite consists of three components; the FFEA runner, FFEAtools, and the FFEA visualizer. The FFEA runner is a C++ implementation of the FFEA method, which will simulate the molecular dynamics of biological objects in a fluid subject to stochastic thermal noise. FFEAtools is a Python toolsuite package designed to assist FFEA users with setting up FFEA systems and analysing simulation results. The FFEA visualizer is a plugin for PyMOL (a Python-based molecular visualization tool (Schrödinger 2023)), which loads in the data from FFEA simulations produced by the FFEA runner, and visualizes the results.

A review of coarse-grained models in structural biology by Hafner et al. (2019) showcased many recent novel techniques and how they have enabled new approaches for understanding the mechanics of the formation and biophysical functions of biological systems. However, they also noted in their conclusion a need to combine structures in molecular dynamics (as opposed to simulating a single system in isolation) in order to approach biological complexity and function. FFEA has begun allowing for some of this research, such as simulations of cytoplasmic dynein on the microtubule (Hanson 2018). Solernou et al. (2018) also stated that FFEA was a response to a need for novel techniques that analyse information about mesoscale macromolecules generated by new advances in biophysical tools such as cryo-electron microscopy, electron cryotomography, and super-resolution microscopy. They also showed that an advantage of FFEA is the ability to build continuum structures from low-resolution data, meaning it is able to create surface profiles from electron density maps for meshing.

FFEA uses two formats of representation for biomolecules. Most biomolecules, ones globular

in shape, use "blob" (BioLogical OBject) representation. Blobs are a tetrahedral mesh representation of a biomolecule, with each element of the mesh being subject to the mechanics of the simulation (Equation 2.6). The alternative format is "rod" or KOBRA representation. KOBRA (Kirchoff Biological Rod Algorithm) is an algorithm designed to represent the dynamics of elongated "rod-shaped" biomolecules, such as their elasticity (Welch 2020). Figure 2.6 shows a comparison of the two representation formats.

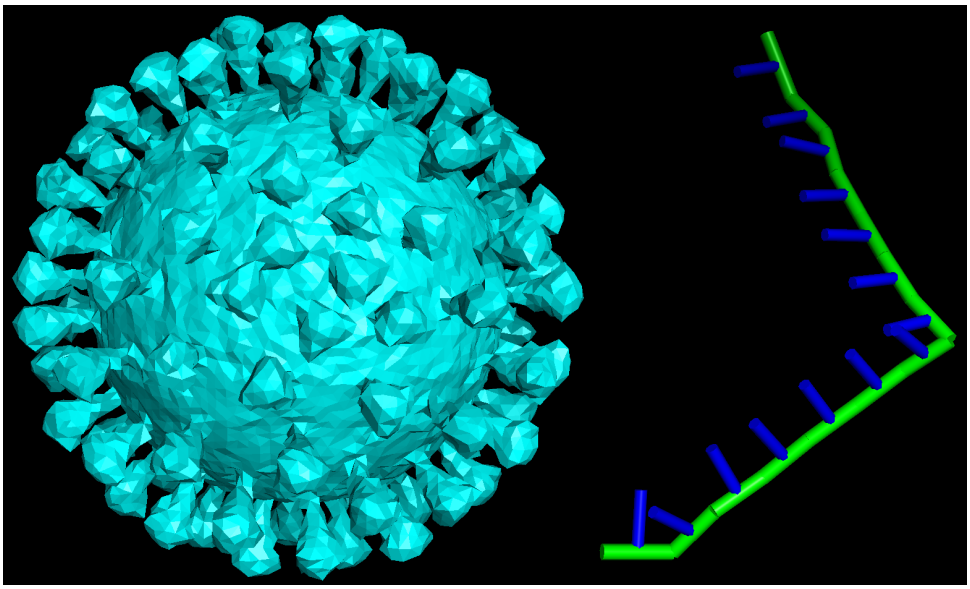

Figure 2.6: SARS-CoV-2 viral envelope in blob format (left) and myosin 5a in KOBRA rod format (right) for FFEA, viewed in PyMOL. In KOBRA representation, the green components represent the elements of a rod-shaped biomolecule, and the blue components represent material axes.

As mentioned at the start of this section, FFEA was developed to analyse and solve biophysical problems for biomolecules at a length scale (the mesoscale) that other molecular dynamics software have not been able to accurately simulate in a reasonable time frame. Since its initial development, FFEA has been applied to some of these problems. For example, Hanson (2018) applied FFEA to modelling mesoscopic cytoplasmic dynein, with the goal of understanding how it "walks" along a microtubule. A validated FFEA model was used to demonstrate potentially responsible mechanisms, such as what allows for dynein's flexibility, and the range of interaction potential energy values that would be necessary for dynein to stay associated with a microtubule track via surface diffusion. Hanson explains that understanding the functions of molecular motors could be of use to medical science for problems such as drug delivery and health enhancements. Richardson et al. (2014) applied FFEA (alongside elastic network modelling) to work towards validating cryo-EM reconstructions of ATPase rotary motors, which show

a seemingly-counterintuitive connection between stators and rotating subdomains of the motor, hindering the rotation. The two models are used to discuss a potential role for this connection by severing it and demonstrating through simulation how this state changes the collective dynamics by being dominated by the local motion of the disconnected stator. The exact reason for this connection is undetermined here, but potential reasons are presented for further study to validate the cryo-EM models.

2.5 Chapter conclusions

This chapter has discussed literature on the topic of research software engineering, with specific attention paid to the problems with sustainability in the field. As research software is primarily focused on an output, and often developed by researchers untrained in software development, much research software rapidly decays and will need re-developing for future research on the same or similar topics. This problem is exacerbated by the lack of consensus on the definition of software sustainability, however, qualitative studies allow us to see what features of sustainability are important to the scientific community, such as good documentation, active maintenance, data and code being openly available, and code being easy to read.

While there is no agreed-upon definition of software sustainability, for the purposes of this research this thesis will define software sustainability as "the qualities of software which impact its usability, maintainability, and longevity".

This chapter has also introduced the numerical method and research software suite Fluctuating Finite Element Analysis (FFEA), which uses continuum mechanics to simulate the dynamics of mesoscopic molecules that many traditional molecular dynamics methods are unable to simulate accurately in a reasonable time frame. This method has been applied to better understand the mechanics of large macromolecules for potential uses in fields such as medicine, and to assess the validity of other molecular models.

As FFEA is a piece of research software, it is subject to all the potential problems that come with research software, and without due maintenance could be left in an unsustainable state, dissuading future use. This should not be allowed to happen, as it would be a waste of over a decade of work from numerous contributors. In the following chapters, sustainable RSE principles will be applied to FFEA. This will first be used to determine what exactly are potential

sustainability problems with FFEA, instead of trying to improve the sustainability of FFEA right away, potentially missing and neglecting significant contributors to software decay. Once the sustainability issues and the bottlenecks in the workflow are properly identified, subsequent chapters will then work towards solutions to these problems.

Chapter 3

Evaluation of modern novel research software – A case study of fluctuating finite element analysis. Part I: Quantitative analysis.

The literature on research software engineering discussed in the previous chapter explains and demonstrates issues in unsustainable software practices that can cause rapid software decay. The goal-oriented nature of research software, and their development being done by researchers regardless of software engineering training, often leads to unmaintained software that can rot to the point of being unusable, meaning a lot of redundant re-creations of software designed for similar tasks.

If a piece of research software is in danger of software rot, how might one avert this danger and develop the software into a more sustainable version of itself? Before that question can be answered for a piece of software, it needs to be known what exactly is causing issues contributing to software decay. Is the bottleneck in the software's use in the performance of the code, is it in the way users interface with the software, is it caused by a flaw in the design of the software, or something else entirely? If software engineers were to do a cursory analysis of the software and start fixing the first seemingly-serious issue found, it is entirely possible that a more critical flaw could go unnoticed when it should have been made a higher priority. In particularly bad cases,

it could render that premature maintenance useless (such as in cases where re-writing some or all of the software is required), and thus increase the time and financial cost of developing the software.

Quantitative and qualitative evaluation is required in order to get a more holistic understanding of a piece of software and its potential flaws. In other words, the condition and performance of the software's code needs to be understood, as does the human experience with using and developing the software. This chapter, and the one following, shows how such an evaluation can be performed, using FFEA version 2.7.2 as a case study. In the general methodology as outlined by Figure 1.1, these chapters make up the "Evaluation of research software" stage of this research.

Due to the lack of consensus among the FFEA development team as to what the major issues were that limited the FFEA 2.7.2 software suite, an evaluation was conducted to analyse the code, and analyse user and developer opinions of the software. This was written up as an internal developer report in November 2021 (Rogers 2021a). This chapter details the quantitative findings of this evaluation relevant to this thesis — the static and dynamic code analysis conducted to detect potential issues in FFEA's source code, and then this analysis is used to identify which of these issues may impact FFEA's sustainable software qualities. This is the "Quantitative Analysis" substage of the general methodology as outlined by Figure 1.1.

Quantitative analysis was performed using a toolset developed for this project in Python called `ffea-dev-tools` (Rogers 2021b). This toolset contains two tools, `FFEA_linter` and `FFEA_profiler`. Each of these tools are documented, so more detailed information on how they work can be found at the `ffea-dev-tools` repository. This chapter will detail the analysis of the FFEA software suite code using these tools, plus additional analysis from stress testing.

3.1 Static code analysis

Static code analysis was performed using `FFEA_linter`, a Python script for linting the entire source code's C, C++, CMake, and Python files to showcase any programming and stylistic errors. This tool outputs log files of the linting results of each source code file, and a general log file summarising the entire source code's linting results.

Linting tools are useful for helping developers to keep the source code consistent throughout

the files to make it easier for the current and future development team to collaborate, and to notify them of any code smells (such as deprecated features) without having to check for such issues manually. As highlighted by the study on software sustainability factors by Souza et al. (2019) (§2.1.2), one of the more valued factors by RSEs was the readability of the code. This method allows us to compare the source code of FFEA to relevant programming language standards, which will help determine how well it adheres to agreed-upon coding conventions for legibility, and to show the FFEA developers which parts of the code could be written clearer. An alternative method would be for the development team to create their own coding style guidelines to adhere to for consistency in the source code, but it would cost less time to use established coding standards that are more universally familiar to programmers and have existing linting tools written for them.

The source code for FFEAtools (the Python package of tools for the analysis and for assisting the creation of FFEA simulations) is currently implemented in Python 2. Python 2 has reached its end-of-life and is no longer supported or maintained as of January 2020, meaning that any new or remaining issues will no longer be fixed (such as potential security issues), and many Python-based tools such as libraries will start to only support Python 3 (Python Software Foundation 2020). This was identified as a potential major issue for the future of FFEA, with moving the FFEAtools code to Python 3 being seen by some developers as a critical task (Welch 2020). To assist with this, an additional feature of `FFEA_linter` uses a fixer to automatically convert all the Python files in the FFEA software suite's source code from version 2 to 3 and produces reports detailing the changes made that can guide a developer to do the conversion. The tool will then use linting software on the new Python 3 code. These are *not* ready-to-go code files. Automatically-generated code in general often contains imperfections that can cause bugs that either slow down or halt the software.

The `FFEA_linter` tool makes use of four pieces of software in the following ways:

- **Clang-Tidy** - A linting tool for C and C++ (The Clang Team 2023). For each C and C++ file (including header files) in the FFEA software suite's source code, it will produce a log file containing a list of errors and warnings found within the source code file, as well as the line they were found at. Errors point towards source code that cause perceivable problems at runtime, whereas warnings only *might* cause issues and tend to be stylistic issues. The C and C++ source code are linted to the standards the FFEA software suite

was originally written under; C18 and C++11.

- **CMakeLint** - A linting tool for CMake (tkruse 2019). For each CMake file (including the `CMakeLists.txt` file) in the FFEA software suite's source code, it will produce a log file containing a list of errors found within the source code file, as well as the line they were found at (unlike Clang-Tidy there is no differentiation between a warning and error).
- **PyLint** - A linting tool for Python (Logilab 2014). For each Python file in the FFEA software suite's source code (as well as any files generated by 2to3) it will produce a log file. At the top of the log file it will show specific issues and the line they occur at. Afterwards, it generates a report of the analysis of the source code file, showing statistics and metrics of the file, issues with duplication, and the errors by type and amount of occurrence. At the bottom of each file, PyLint scores the source code out of 10, allowing for an easy method of comparison as the source code is developed.
- **2to3** - A code translation program that reads Python 2 code and applies fixers to automatically or semi-automatically translate it into Python 3 code (The Python Standard Library 2023) (for its purposes with FFEA it will be a semi-automatic or guided process). For each Python file in the FFEA software suite, it will produce a new Python file containing the Python 3 code, and a log file containing the report from 2to3.

3.1.1 Results

The following table shows the results produced when running `FFEA_linter` on FFEA 2.7.2:

Table 3.1: `FFEA_linter` results for FFEA 2.7.2 as recorded by the individual linting tools used for each programming language

Language	C	C++	CMake
Number of Files	4	143	6
Linting Results	Total Errors: 3 Total Warnings: 155	Total Errors: 131 Total Warnings: 2435	Total Errors: 152

Language	Python 2	Python 3
Number of Files	154	
Linting Results	Average Score: -11.09/10	Average Score: -16.11/10

The C and the C++ files are probably the highest priority for standardisation, with a total of 134 errors found, as well as a huge number of warnings for potential errors. The amount of

warnings are a potential concern as it will take a long amount of time to fix.

The CMake results almost entirely point to simple readability errors, mainly issues with white-spaces. There appear to be no errors that could cause problems with the functionality of the source code, so this is likely the lowest priority for standardisation of the languages used.

The Python results have quite damning scores, however simply cleaning the individual files will not resolve the problem. Python 2 source code cannot be put into a Python 3 project structure and remain functional, so standardising the FFEAtools code in FFEA 2.7.2 would waste time. A change to the project structure is required first. There are also possible problems with the results obtained from PyLint and 2to3. Due to the inconsistent encoding of whitespaces (the latter likely due to developers of FFEAtools using a variety of text editors and IDEs on different operating systems that encode whitespaces differently), in some circumstances, they break. In the internal report where this evaluation was conducted, developers were advised to view Python linting results with caution as is possible that the linter and fixer have not correctly or fully analysed the source code.

While the FFEAtools code requires a new project structure, a migration of Python 2 to Python 3 by way of restructuring and converting the Python code is a risky process as it requires changing more than one code issue at a time, making it a more complex task more likely to introduce errors that are more difficult to source. It was suggested that FFEAtools should undergo full re-implementation rather than each file to be automatically converted to fix this problem.

3.2 Dynamic code analysis

Dynamic code analysis was performed using `FFEA_profiler`, a Python script for running dynamic profiling software on the FFEA runner using a series of datasets to measure performance and spot problems in the source code, such as memory leaks and cache misses. This can be used to determine the sustainability of FFEA from the perspective of resources; to see if there are any issues in the source code which cause disruptions to workflows and thus damaging user retention due to being difficult to work with. This will also help us note any inefficiencies with the source code that cause the software to run for longer than necessary or inefficiently use more computational resources than necessary.

Dynamic profiling requires the FFEA runner to be executed, therefore, a series of datasets have been collected that cover each of the FFEA runner's major features. This includes utilising both of FFEA's meshing methods; tetrahedral meshes and KOBRA meshes (see §2.4). This is done to test the reliability of FFEA for the many kinds of research cases a user may work with, as the reliability of software is key factor in the first pillar of research software engineering (Figure 2.2). These datasets are split into three categories:

1. **blob** - These datasets contain tetrahedral meshes of biological objects that most users would be working with.
2. **many-blob** - These datasets contain large numbers of blob objects in a single simulation. The FFEA runner has a separate mode for running simulations that contain a large number of blobs that is more efficient for this situation than the standard runner (called by the command `ffea_mb`), and therefore this category is for testing that mode.
3. **rod** - A recently added feature of the FFEA software suite has been the inclusion of a rod model (KOBRA) for representing slender biomacromolecules (Welch 2020). As this utilises new parts of the source code, these datasets are used to test this.

For each of these categories, there are three datasets, with the general idea of there being a small, medium, and large dataset in terms of computation required to run them (the soak tests performed on these datasets in §3.3 provide some further insight into their computational intensity).

The format of each dataset is an FFEA script (a file with the format `.ffea`) and any input files it requires, mainly geometric files of the model like `.vol`, `.surf`, and `.top`, or files containing the physics parameters such as `.vdw` files. The FFEA scripts themselves are text-based files with a similar format to the XML markup language. These files contain the system which details the models and how they may interact with each other, and the simulation parameters such as boundary conditions. Each script is set up to execute a valid FFEA simulation, however, some changes are made to the time parameters.

There are three time-based parameters in an FFEA script; `num_steps` is the simulation length, `check` is the number of steps before the FFEA runner outputs a frame of the simulation (which will contain the coordinates and energy measurements of the simulation objects), and `dt` (delta time) is the time step. For the purpose of profiling, each dataset has the same `num_steps` and

`check` parameters, 1000 and 100 respectively. The `dt`, however, will be kept to whatever is appropriate for the simulation, as different simulations can handle different time steps based on the magnitude of forces and the size of the shortest edge (there is a function in FFEAtools that can calculate the maximum time step a mesh can handle). The simulations have been made so short because they are being used for measurement of the software rather than to obtain good results from the simulations. This is especially necessary as profiling software can cause massive slowdowns. Further information on how FFEA files work can be found in the software's documents (Cocking et al. 2021a).

The following is a list of each of the nine datasets used for `FFEA_profiler`.

- `blob1` (`B1`) - An individual dynein protein.
- `blob2` (`B2`) - An individual SARS-CoV-2 viral envelope.
- `blob3` (`B3`) - Two dynein proteins with collisions on a microtubule.
- `many-blob1` (`MB1`) - A system of 50 colliding soft spheres in a containing box.
- `many-blob2` (`MB2`) - A system of 500 colliding soft spheres in a containing box.
- `many-blob3` (`MB3`) - A system of 1000 colliding soft spheres in a containing box.
- `rod1` (`R1`) - An individual NDC80C protein.
- `rod2` (`R2`) - An individual myosin protein with a bend.
- `rod3` (`R3`) - A sphere connected to a microtubule via NDC80C with three legs (this is to test the software when it is performing blob-rod interactivity).

Table 3.2 provides a screenshot of each dataset when viewed in the FFEA visualizer plugin for PyMOL (except for `MB3` which had to be loaded in ParaView, as PyMOL was unable to load that many meshes). Table 3.3 provides some parameter information on the datasets used, showing the number of elements and nodes the meshes have, the dimensions of the bounding box around the system, the timestep (DT), and whether or not they include some form of interactivity between objects such as collisions and connections.

Table 3.2: Meshes of each system used as datasets for FFEA_profiler. The `B` and `MB` datasets consist of tetrahedral meshes, datasets `R1` and `R2` consist of KOBRA meshes, and dataset `R3` combines tetrahedral and KOBRA meshes.

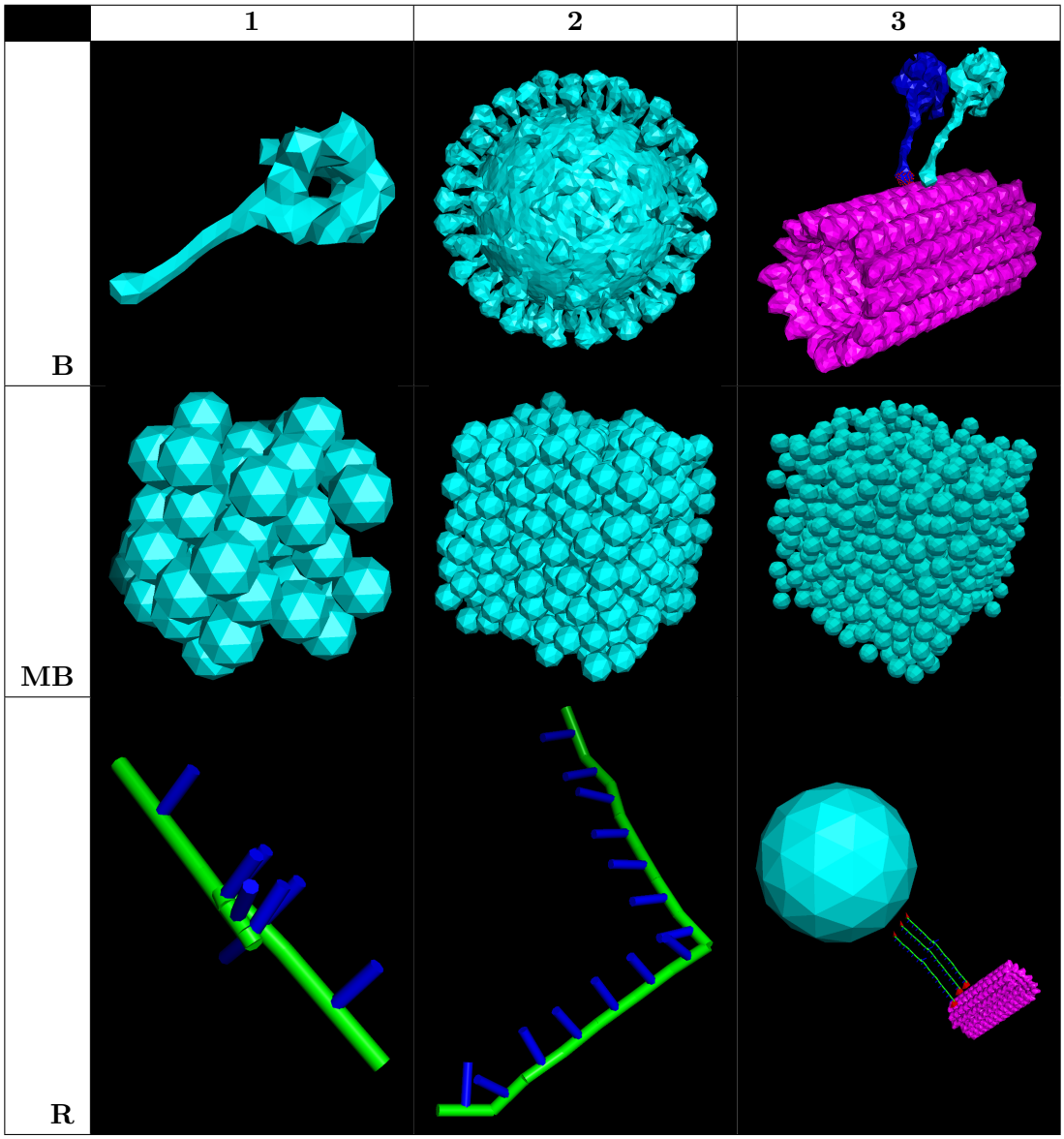

Table 3.3: Mesh and FFEA simulation data for each FFEA_profiler dataset

Dataset	Number of Elements	Number of Nodes	Dimensions (Å)	DT (s)	Interactions
B1	825	1580	226.28 x 256.36 x 126.41	1e-13	No
B2	59532	102480	1822.65 x 1787.31 x 1832.75	1e-13	No
B3	42059	81159	358.77 x 544.37 x 550.79	1e-12	Yes
MB1	2200	5750	2968.73 x 3049.83 x 2950.43	1e-9	Yes
MB2	22000	57500	6603.29 x 6598.14 x 6615.14	1e-9	Yes
MB3	220000	575000	8339.62 x 8341.62 x 8337.94	1e-9	Yes
R1	12	13	101.47 x 147.15 x 110.09	1e-11	No
R2	15	16	134.51 x 117.14 x 360.44	3e-13	No
R3	38942	74682	1856.97 x 973.61 x 981.45	3e-13	Yes

`FFEA_profiler` uses the toolsuite Valgrind (Valgrind Developers 2022a) to perform the profiling on FFEA. This includes such tools as a memory-management profiler, a cache profiler, a heap profiler, a thread debugger etc. These tools produce text outputs that can be written to log files with detailed explanations of problems within the source code, and for some tests provides a `.out` file containing tables and graphs of the results. The following list is of the functions that `FFEA_profiler` performs:

- **Memcheck** - Uses Valgrind's Memcheck tool to profile the memory usage of FFEA and detect errors such as memory leaks.
- **Cachecheck** - Uses Valgrind's Cachegrind tool to profile the cache usage of FFEA and detect errors such as cache misses.
- **Callcheck** - Functionally identical to Cachecheck, but it also generates call-graphs of FFEA.
- **Threadcheck** - Uses Valgrind's Helgrind tool to profile the threading of FFEA and detect errors such as data races.
- **Heapcheck** - Uses Valgrind's Massif tool to profile the heap of FFEA to measure how much heap memory is used.
- **Normal Runs** - Runs FFEA normally with no profiling tool a specified number of times. This can be used to run a dataset multiple times and get an average of the time taken. This is included as a quick check tool and is not used in this evaluation.

These tests were ran on two systems; a general-purpose but high-end PC, and an HPC supercomputer. This was to see how they perform on the two kinds of systems the FFEA software suite are commonly used on. Table 3.4 provides the specifications for the two machines used.

Table 3.4: System information of the machines used to profile FFEA[1]

Specification	**PC**	**N8 Bede**
System Name	Dell Precision 3551	N8 Bede "infer" node
Operating System	CentOS Linux 8.3.2011	Red Hat Enterprise Linux Server 7.6
Platform	4.18.0-240.22.1.el8_3.x86_64	4.14.0-115.29.1.el7a.ppc64le
Processing	Intel i7-10850H CPU @2.7GHz	2 IBM POWER9 CPUs @2.9GHz
Graphics Processing	Nvidia Quadro P620 GPU	4 NVIDIA Tesla T4 16G PCIe
Memory	32GB DDR4 RAM	256GB DDR4 RAM

[1] N8 Bede hardware specifications are taken from `https://n8cir.org.uk/supporting-research/facilities/bede/hardware/`.

It must be noted that in FFEA 2.7.2 at the time dynamic analysis was conducted (June 2021) there is an error in the rod source code with running FFEA scripts through calling them via a path as opposed to running scripts directly from their containing directories[2]. This causes an issue with running Valgrind's profiling tool on N8 Bede, meaning that results can not be obtained for the rod datasets on there until this issue is fixed. Results that could not be obtained through this will be designated as "N/A" on results tables.

3.2.1 Memory analysis results

With the results from running Memcheck, information on memory leaks, the amount of memory lost during those leaks, and the number of memory errors caused by these leaks can be extracted. During the running of these profiling tools, it must be noted that there was an error that would repeatedly occur while running the dataset `mb3` on Bede. It appeared to be finding many small errors in the Boost C++ libraries which FFEA utilises, clogging up the results until the profiling tools would stop reporting. Therefore, on the results tables, this will be denoted by "ERR". Please also note that all bar charts in this subsection use a base 10 logarithmic scale due to the huge range of effects of memory leaks. Logarithmic tick marks are provided on the appropriate axes.

There are two main parts to the log files that Valgrind's Memcheck produces that were used to extract this memory information. The bottom of the log file shows two summaries that provide the totals of leaks and errors that occur in memory, an example of which can be seen in Figure 3.1.

```
==37723== LEAK SUMMARY:
==37723==    definitely lost: 16 bytes in 2 blocks
==37723==    indirectly lost: 0 bytes in 0 blocks
==37723==      possibly lost: 112,080,098 bytes in 273,095 blocks
==37723==    still reachable: 857,880 bytes in 3,469 blocks
==37723==                       of which reachable via heuristic:
==37723==                         newarray           : 5,516,936 bytes in 12 blocks
==37723==         suppressed: 0 bytes in 0 blocks
==37723== Reachable blocks (those to which a pointer was found) are not shown.
==37723== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==37723==
==37723== ERROR SUMMARY: 85 errors from 85 contexts (suppressed: 0 from 0)
```

Figure 3.1: Example of the leak summary tables from Valgrind's Memcheck's log files, showing the four types of data leaks, how many bytes and blocks are lost in these leaks, and the total number of errors found.

[2]This is issue #58 in the FFEA Bitbucket issue tracker, which can be found at `https://bitbucket.org/FFEA/ffea/issues/58/assertions-in-rod_structurecpp-can-cause`.

As shown in the summary tables, Memcheck denotes four different types of leaks (Valgrind Developers 2022b), which are as follows:

- **Definitely Lost (D)** - No pointers can be found to this block, and it is lost. Calls to malloc here cannot have been freed.
- **Indirectly Lost (I)** - No pointers can be found to this block because all the blocks that point to it are lost themselves. These are indirectly caused by the definitely lost blocks, therefore these will be fixed if the root causes are fixed.
- **Possibly Lost (P)** - Pointers *can* be found to this block, but at least one is an interior-pointer (a pointer to the middle of an allocation). Programmers can deliberately use interior-pointers and thus these are not a problem if that is the case, however, they can also be the result of a random value in memory that happens to point to a block it should not.
- **Still Reachable (SR)** - A start-pointer(s) has been found to this block, which the programmer may have freed before the program exit. These kinds of blocks are very common and usually not of concern, and are simply provided for full context.

The other part of Memcheck's log files used is the loss records as shown in Figure 3.2. These detail a recording of each memory loss, showing the number of blocks affected, the size of the memory loss, its target, and the causes. This can include libraries used rather than just the source code of the profiled software.

```
==37723== 8 bytes in 1 blocks are definitely lost in loss record 4 of 95
==37723==    at 0x4C33943: operator new[](unsigned long) (vg_replace_malloc.c:579)
==37723==    by 0x4F356E: SparsityPattern::create_sparse_matrix() (SparsityPattern.cpp:112)
==37723==    by 0x4B83E4: Blob::init() (Blob.cpp:462)
==37723==    by 0x46DE43: World::read_and_build_system(std::vector<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>
>, std::allocator<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > > >) (World.cpp:2661)
==37723==    by 0x46F949: World::init(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >, int, int, bool)
(World.cpp:438)
==37723==    by 0x44ABD3: main (ffea.cpp:169)
```

Figure 3.2: Example of a loss record from Valgrind's Memcheck's log files, which details per individual memory loss the total number of bytes and blocks lost, the target line of code of the memory, and the lines of code which are causing the leaks.

Table 3.5 shows the number of blocks in the code that contain these types of memory leaks. The most important memory leaks to take note of are the ones that cause blocks to be definitely lost, which can be seen in the columns "D". These are the memory leaks definitely caused by an error in the source code that a programmer should fix.

Table 3.5: Memcheck results for number of blocks with memory leaks in FFEA 2.7.2.

Dataset	**Number of Blocks with Memory Leaks**							
	PC				N8 Bede			
	D	I	P	SR	D	I	P	SR
B1	0	0	30,101	1,588	2	31,658	0	5
B2	0	0	2,948,924	10	1	1	2,948,903	5
B3	2	0	273,095	3,469	3	1	276,532	6
MB1	0	0	149,749	8	1	143,738	5,988	6
MB2	0	0	1,497,049	8	66	1,467,027	29,933	7
MB3	0	0	2,994,049	8	ERR	ERR	ERR	ERR
R1	1	42	0	7	N/A	N/A	N/A	N/A
R2	1	43	0	7	N/A	N/A	N/A	N/A
R3	2	0	102,124	10	N/A	N/A	N/A	N/A

Figure 3.3 plots Table 3.5 onto logarithmic bar charts. When it comes to blocks that are definitely lost, it can be seen that for the dataset `Bede_MB2` it produced far more memory leaks to a total of affecting 66 blocks, in comparison to the other datasets that affected 0-3. It is expected for *indirectly* lost blocks to have a very wide range in number, as different blocks can have different levels of responsibility for other blocks. A direct memory leak in one block might not responsible for any other blocks, and another may be responsible for hundreds or thousands of others. However, for the direct leaks, this result is less expected and this may be an outlier, but it cannot definitely be dismissed as so, so further analysis will take this into consideration.

From the results for the blocks that are possibly lost, it appears that the rod code never uses interior-pointers, as only the dataset `R3` shows possibly lost blocks, which is the rod dataset that also contains blobs. This does not necessarily show there is a problem in the blob source code, however, if it is determined that these interior-pointers are a problem then this narrows down where they are. As for the still reachable blocks, it is expected for there to be a few of these as denoted by the majority of datasets that record approximately the same amount of 5-10 blocks, but future developers working on memory issues were advised to look at the blocks pointed to by `PC_B1` and `PC_B3` due to the large jump in the amount of these blocks.

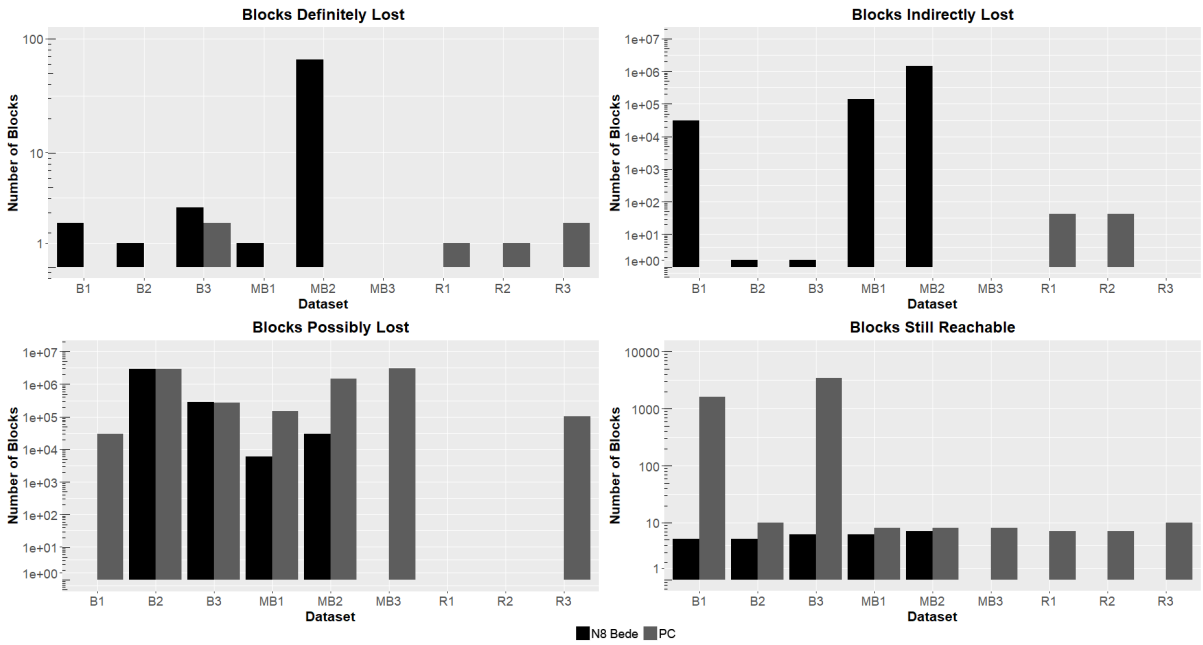

Figure 3.3: Logarithmic bar charts showing the number of blocks with memory leaks in FFEA 2.7.2., separated by type of memory leak.

Table 3.6 shows the number of bytes lost to the different types of memory leaks which is then plotted onto logarithmic bar charts in Figure 3.4. This shows the extent to which memory leaks are causing issues.

While the blocks that are directly lost do not cause much memory loss (only up to a few KB), it can be seen that they in turn cause a lot of indirect memory loss. On PC, these lost blocks can cause around 254-363KB of lost memory, and on Bede this can reach MBs of lost memory (up to around 213MB if `Bede_MB2` is not an outlier).

The datasets that record possibly lost blocks are usually recording MBs of potentially lost memory, as high as around 475MB for `B2`. The report recommended investigating if interior-pointers are intentional in the FFEA runner source code, because if they are not then they are causing major losses.

The still-reachable blocks are usually only recording a size of around 2-6KB which seems an acceptable range. However, for the aforementioned `PC_B1` and `PC_B3` where they divert from the norm, they are in the hundreds of KB, so while a lower priority than the other memory leaks, it would still be appropriate to investigate these blocks.

Table 3.6: Memcheck results for number of bytes lost to memory leaks in FFEA 2.7.2.

Dataset	**Bytes Lost to Memory Leaks**							
	PC				N8 Bede			
	D	I	P	SR	D	I	P	SR
B1	0	0	5,515,963	400,352	2,056	5,902,751	0	1,920
B2	0	0	475,147,095	5,784	48	31	475,141,280	1,920
B3	16	0	112,080,098	857,880	64	55	112,926,023	2,488
MB1	0	0	21,714,535	6,192	2,008	20,827,367	883,460	2,488
MB2	0	0	223,265,935	6,192	3,752	213,014,351	10,283,892	2,528
MB3	0	0	462,411,935	6,192	ERR	ERR	ERR	ERR
R1	2,008	363,469	0	2,640	N/A	N/A	N/A	N/A
R2	2,008	254,475	0	2,640	N/A	N/A	N/A	N/A
R3	16	0	85,995,906	3,824	N/A	N/A	N/A	N/A

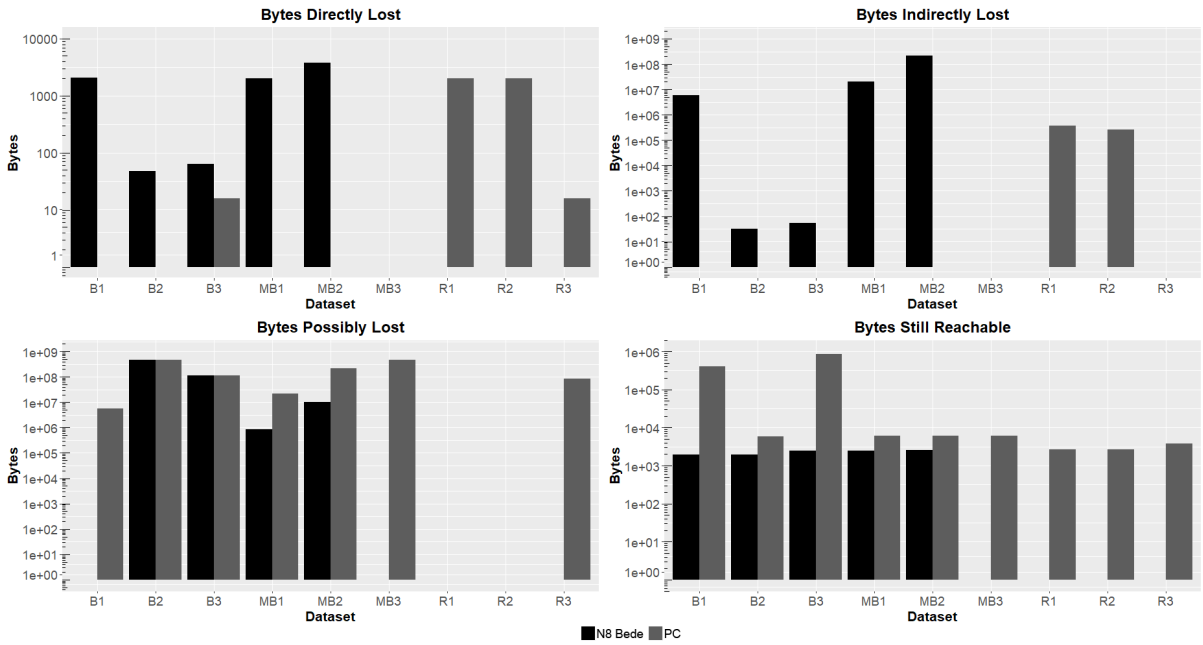

Figure 3.4: Logarithmic bar charts showing the number of bytes lost to memory leaks in FFEA 2.7.2., separated by type of memory leak.

As the direct memory leaks are of the highest priority to fix, a list was created of all the lines of the source code that caused direct leaks. Using the loss records of each dataset, for each of these lines containing errors it can be shown what targets they affect, the datasets they appeared in, and the range of bytes, directly and indirectly, they cause to be lost. These values can be ranges as the same memory leak may appear in different datasets with different recorded loss values. These ranges are the range of potential loss if every found memory leak across all datasets were to occur in one run. Two of these lists have been created, one where it is assumed `Bede_MB2` is an outlier and is therefore disregarded, and one where this is not done. Table 3.7 shows the list

of source code lines that cause memory leaks if `Bede_MB2` is disregarded.

Table 3.7: Memory errors in FFEA 2.7.2 (disregarding `Bede_MB2`).

Error Location	**Affected Targets**	**Datasets Appeared In**	**Bytes Directly Lost by Affected Blocks**	**Bytes Indirectly Lost by Affected Blocks**
Blob.cpp:462	0x4C33943 0x40872DC	PC_B3 PC_R3 Bede_B3	16	0
Blob.cpp:463	0x4C33943 0x40872DC	PC_B3 PC_R3 Bede_B3	16	0
ffea.cpp:165	0x4C326DF 0x4085CBC	PC_R1 PC_R2 Bede_B1 Bede_MB1	2,056 - 4,016	254,475 - 21,190,836
ffea.cpp:169	0x4C33943 0x4085CBC 0x40872DC	PC_B3 PC_R3 Bede_B1 Bede_B2 Bede_B3	64	31 - 55
SparsityPattern.cpp:112	0x4C33943 0x40872DC	PC_B3 PC_R3 Bede_B3	16	0
World.cpp:438	0x4C33943 0x4085CBC 0x40872DC	PC_B3 PC_R3 Bede_B1 Bede_B2 Bede_B3	64	31 - 55
World.cpp:2209	0x4085CBC	Bede_B1 Bede_B2 Bede_B3	48	31 - 55
World.cpp:2661	0x4C33943 0x40872DC	PC_B3 PC_R3 Bede_B3	16	0

Figure 3.5 plots the ranges of bytes lost by these errors (ignoring `Bede_MB2`) on a logarithmic bar chart. In this case, most of these errors only cause 10s of bytes of memory loss, which is insignificant. However, the line `ffea.cpp:165` stands out as one which can indirectly cause between around 254KB to 21MB of memory loss. It would seem that this line is definitely a major problem that needs addressing, but the results that include `Bede_MB2` must first be considered.

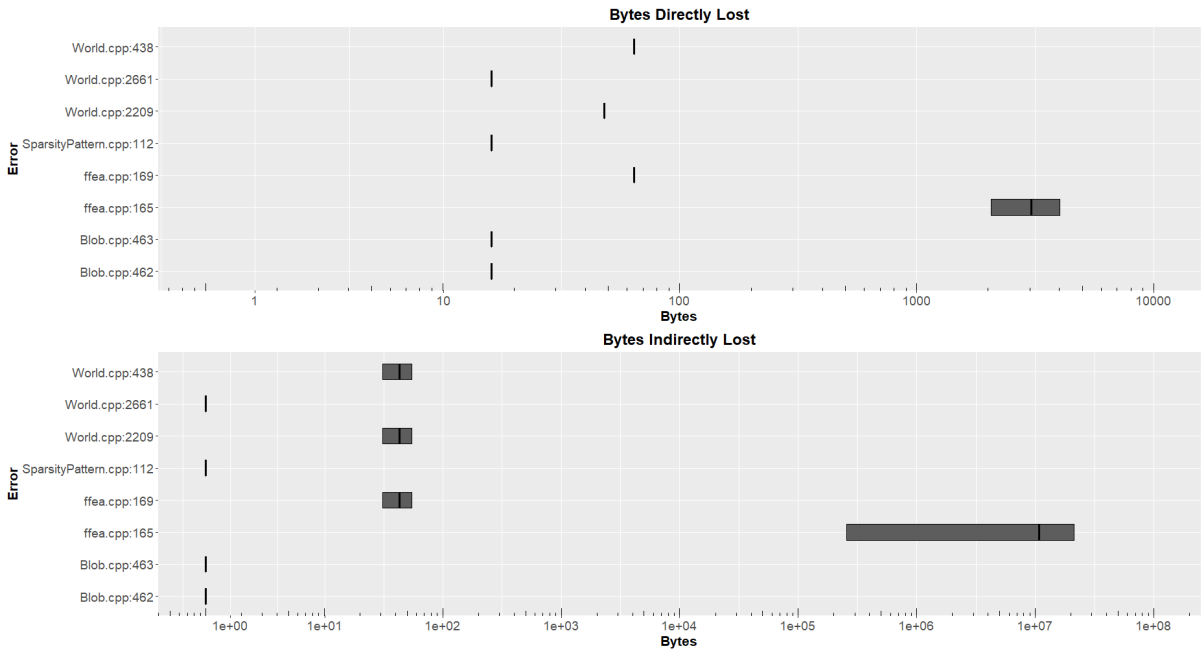

Figure 3.5: Logarithmic bar charts showing the range of bytes directly and indirectly lost by memory errors in FFEA 2.7.2 (disregarding `Bede_MB2`). The grey boxes show the full recorded range of bytes lost, and the black lines show the median bytes lost for that error.

Table 3.8 lists the source code lines that cause memory errors and details their effects, and Figure 3.6 plots the range of bytes they lose, including results from `Bede_MB2`. By considering the possible outlier, more lines appear that show problematic memory losses up to the MBs, however, `ffea.cpp:165` still remains the largest potential offender with a recorded memory loss of up to just under 186MB.

Table 3.8: Memory errors in FFEA 2.7.2 (including `Bede_MB2`).

Error Location	**Affected Targets**	**Datasets Appeared In**	**Bytes Directly Lost by Affected Blocks**	**Bytes Indirectly Lost by Affected Blocks**
Blob.cpp:347	0x4087CD0	Bede_MB2	872	12,315,834
Blob.cpp:370	0x40866B0	Bede_MB2	96	424,618
Blob.cpp:379	0x40872DC	Bede_MB2	184	2,972,814
Blob.cpp:462	0x4C33943 0x40872DC	PC_B3 PC_R3 Bede_B3 Bede_MB2	16 - 480	0 - 10,192,664
Blob.cpp:463	0x4C33943 0x40872DC	PC_B3 PC_R3 Bede_B3 Bede_MB2	16 - 128	0 - 1,698,736
Blob.cpp:3989	0x4087CD0	Bede_MB2	872	12,315,834

Error Location	**Affected Targets**	**Datasets Appeared In**	**Bytes Directly Lost by Affected Blocks**	**Bytes Indirectly Lost by Affected Blocks**
ffea.cpp:165	0x4C326DF 0x4085CBC	PC_R1 PC_R2 Bede_B1 Bede_MB1 Bede_MB2	2,056 - 4,016	254,506 - 185,773,154
ffea.cpp:169	0x4C33943 0x4085CBC 0x40872DC 0x40866B0 0x4087CD0	PC_B3 PC_R3 Bede_B1 Bede_B2 Bede_B3 Bede_MB2	1,032 - 1,496	12,740,483 - 22,933,171
NoMassCGSolver .cpp:143	0x40872DC	Bede_MB2	184	2,972,814
parallel.c:171	0x40866B0 0x40872DC 0x4087CD0	Bede_MB2	1,088 - 1,440	14,439,188 - 22,933,116
SparseMatrixTypes .cpp:39	0x40872DC	Bede_MB2	120 - 472	1,698,736 - 10,192,664
SparsityPattern .cpp:112	0x4C33943 0x40872DC	PC_B3 PC_R3 Bede_B3	16	0
SparsityPattern .cpp:126	0x40872DC	Bede_MB2	120 - 472	1,698,736 - 10,192,664
World.cpp:438	0x4C33943 0x4085CBC 0x40872DC 0x40866B0 0x4087CD0	PC_B3 PC_R3 Bede_B1 Bede_B2 Bede_B3 Bede_MB2	1,032 - 1,496	12,740,483 - 22,933,171
World.cpp:2209	0x4085CBC	Bede_B1 Bede_B2 Bede_B3	48	31 - 55
World.cpp:2655	0x40866B0 0x40872DC 0x4087CD0	Bede_MB2	1,088 - 1,440	14,439,188 - 22,933,116
World.cpp:2661	0x4C33943 0x40872DC 0x40866B0 0x4087CD0	PC_B3 PC_R3 Bede_B3 Bede_MB2	984 - 1,448	12,740,452 - 22,933,116

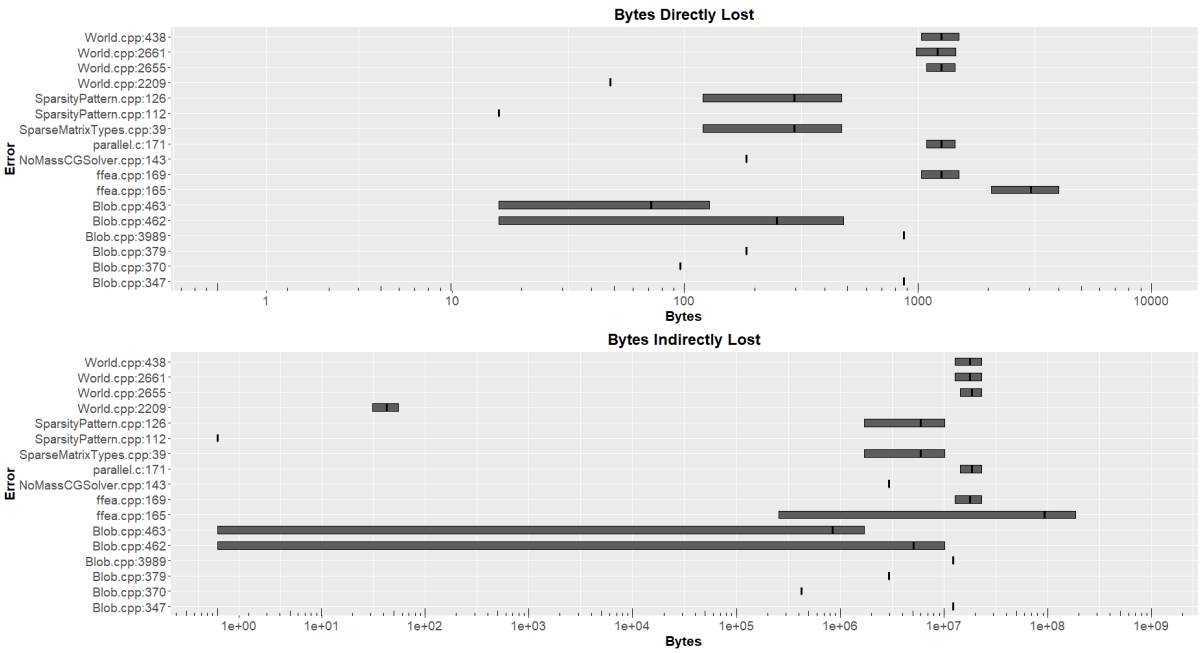

Figure 3.6: Logarithmic bar charts showing the range of bytes directly and indirectly lost by memory errors in FFEA 2.7.2 (including `Bede_MB2`). The grey boxes show the full recorded range of bytes lost, and the black lines show the median bytes lost for that error.

As `ffea.cpp:165` is definitely seen to be a major issue, the report recommended considering it the highest priority issue for the memory in FFEA 2.7.2. Looking at the source code this can be found for `ffea.cpp:165`.

```
164     // Allocate the world
165     world = new World();
```

Figure 3.7: `ffea.cpp` lines 164 & 165 pointing to the class `World()`

Unfortunately, this does not narrow down the exact location of the memory leak in question. It points to a class called `World()` which makes up the majority of the file `World.h`. Fixing this requires analysis of how malloc is used in this header file to figure out where it has gone wrong to then correct it. Developers may also be able to narrow down the location of this memory leak from cross-referencing results from the other checks that include errors in `World.h`.

3.2.2 Cache analysis results

The results of Callcheck produce information on the cache usage of the FFEA runner. Primarily this shows the amounts of requests to cache made by the program, the number of times these resulted in cache misses, and to what type of cache these requests were made to. Similarly

to Memcheck, an error repeatedly would occur when running the dataset `MB3` on Bede, and therefore results tables will denote this again by “ERR”.

The log files produced by Valgrind’s Callgrind (and also Cachegrind) produce tables such as the one in Figure 3.8 which show the cache usage across the entire run. The `.out` files produced show a similar table containing the same information in a different format (similar to the table in Figure 3.9).

```
Bede_B1
I   refs:      7,469,818,707
I1  misses:          276,021
LLi misses:           96,568
I1  miss rate:          0.00%
LLi miss rate:          0.00%

D   refs:      3,529,223,104  (2,472,955,652 rd + 1,056,267,452 wr)
D1  misses:      280,665,823  (  229,335,140 rd +    51,330,683 wr)
LLd misses:      224,325,548  (  175,328,688 rd +    48,996,860 wr)
D1  miss rate:           8.0% (          9.3%   +           4.9%  )
LLd miss rate:           6.4% (          7.1%   +           4.6%  )

LL refs:         280,941,844  (  229,611,161 rd +    51,330,683 wr)
LL misses:       224,422,116  (  175,425,256 rd +    48,996,860 wr)
LL miss rate:            2.0% (          1.8%   +           4.6%  )
```

Figure 3.8: Example of cache usage table from Valgrind’s Callgrind function, showing the referrals, misses, and miss rates of each cache type.

Figure 3.9 shows an example of the function-level counts for cache usage as shown by Callgrind’s `.out` files. For each function that makes calls to cache, this shows the number of calls and cache misses. Read and write calls to cache are denoted by the lowercase “r” and “w” respectively after the cache type, and misses to read and write are denoted by “mr” and “mw” respectively. The amounts of requests and misses also have a percentage next to them that shows how much that function was responsible for the total amount of that type of cache request or miss. The term “responsibility” will be used to refer to this in this section.

```
--------------------------------------------------------------------------------
Ir                      Dr                      Dw                      I1mr              D1mr                    D1mw              ILmr           DLmr                   DLmw                   file:fun
ction
--------------------------------------------------------------------------------
1,635,766,845,994 (27.83%) 456,942,312,850 (18.76%) 396,847,079,816 (44.71%) 4,515,469 (19.92%)    430,052,176 ( 0.29%)  40,753,317 ( 0.86%) 103,773 (12.84%)    67,281,579 ( 0.67%)     200,824 ( 0.01%)  /home/sc
cdrADM/softw/ffea/master_opt_debug/ffea/src/CheckTetrahedra0verlap.cpp:tet_a_tetII(double (&) [3], double (&) [3], double (&) [3], double (&) [3], double (&) [3], double (&) [3], double (&) [3], double (&) [3])
[/home/sccdrADM/softw/ffea/master_opt_debug/ffea/bin/ffea_mb]
1,408,343,239,871 (23.96%) 735,706,771,687 (30.20%)  38,862,386,383 ( 4.38%) 2,171,831 ( 9.58%) 84,599,812,631 (56.83%)      66,972 ( 0.00%)  22,954 ( 2.84%) 1,870,969,068 (18.59%)      39,570 ( 0.00%)  /home/sc
cdrADM/softw/ffea/master_opt_debug/ffea/src/VdW_solver.cpp:VdW_solver::solve(double*) [clone ._omp_fn.0] [/home/sccdrADM/softw/ffea/master_opt_debug/ffea/bin/ffea_mb]
  835,494,385,432 (14.22%) 417,747,192,716 (17.15%) 169,356,970,020 (19.08%)     1,055 ( 0.00%) 29,001,189,088 (19.48%)  14,256,404 ( 0.30%)   1,055 ( 0.13%) 1,008,359,679 (10.02%)      68,334 ( 0.00%)  /home/sc
cdrADM/softw/ffea/master_opt_debug/ffea/src/Face.cpp:Face::checkTetraIntersection(Face*, double*, int, int) [/home/sccdrADM/softw/ffea/master_opt_debug/ffea/bin/ffea_mb]
  304,848,056,212 ( 5.19%) 101,612,604,468 ( 4.17%)  56,452,323,340 ( 6.36%)   452,747 ( 2.00%)        37,312 ( 0.00%)     634,662 ( 0.01%)   2,857 ( 0.35%)          20,607 ( 0.00%)       3,689 ( 0.00%)  /home/sc
cdrADM/softw/ffea/master_opt_debug/ffea/src/Steric_solver.cpp:Steric_solver::do_interaction(Face*, Face*, double*) [/home/sccdrADM/softw/ffea/master_opt_debug/ffea/bin/ffea_mb]
  248,367,335,230 ( 4.23%) 170,257,012,155 ( 6.99%)  62,312,059,538 ( 7.02%)     1,866 ( 0.01%) 14,790,068,512 ( 9.93%)     243,497 ( 0.01%)   1,350 ( 0.17%)     558,632,507 ( 5.55%)      34,164 ( 0.00%)  /home/sc
cdrADM/softw/ffea/master_opt_debug/ffea/include/mat_vec_types.h:VdW_solver::solve(double*) [clone ._omp_fn.0]
```

Figure 3.9: Example of function-level counts for cache usage by Valgrind’s Cachegrind/Callgrind, detailing the amount and types of cache requested and missed by each function called by the analysed software.

The bar chart in Figure 3.10 shows the cache miss rates for each dataset ran, separated by the cache type. The first thing to note was that in all datasets there were zero LLi misses recorded, and therefore this has not been plot on the graph. I1 misses were also very rare with only

`Bede_B3` recording a miss rate of 0.19%. The most commonly occurring cache misses are for D1 which is recorded in every dataset except `PC_R2`. Generally, it seems like the rod code has far fewer cache misses, with the stand-out exception of D1 for `PC_R3` which records the largest cache miss rate of 16.8%.

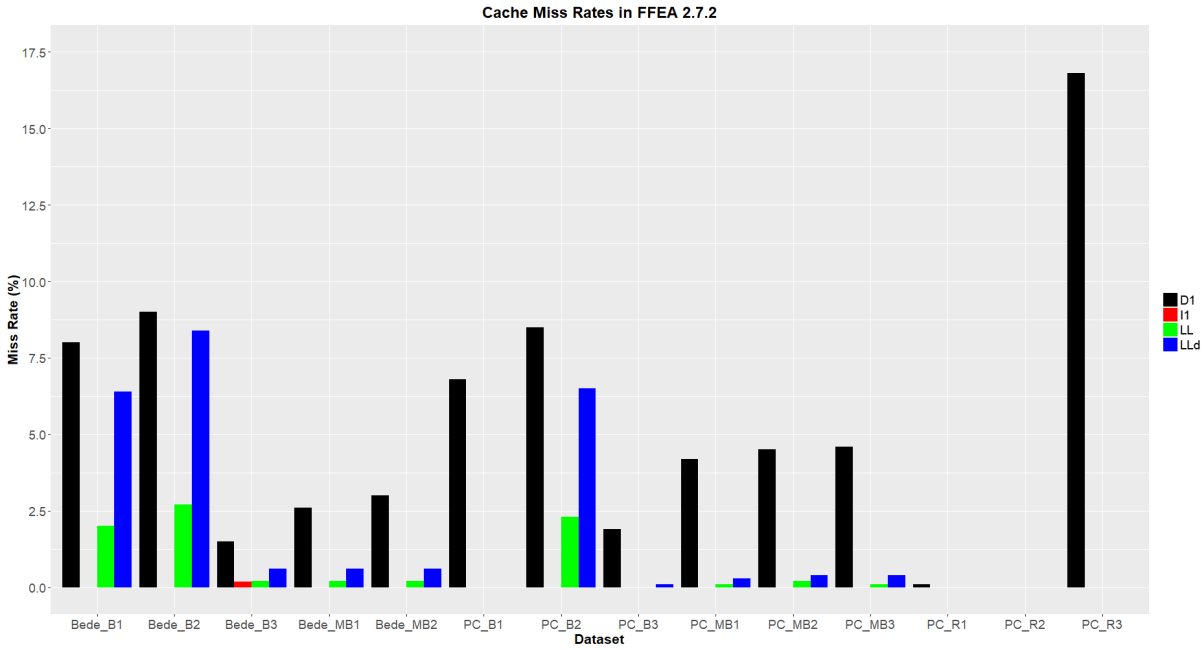

Figure 3.10: Bar chart showing the cache miss rates in FFEA 2.7.2., separated by cache type.

The following is a list of lines in the FFEA 2.7.2 source code that cause cache misses. Listing every single line that causes a cache miss would be infeasible, as some cause very small cache misses. Therefore, this list contains every line that had a responsibility over a threshold of 5% as recorded by the function-level counts detailed in the `.out` files as shown in Figure 3.9. The ones in **bold** are the more significant lines causing cache misses either very frequently or semi-frequently with high levels of responsibility. Some misses are caught during references to header or template files, which is why some will state this with the term "via".

- Blob.cpp:1066
- **Blob.cpp:3826**
- Blob.cpp:3937
- **CheckTetrahedraOverlap.cpp:254**
- Face.cpp:197
- Face.cpp:314
- **Face.cpp:352**
- **Face.cpp:448**
- LJ_matrix.cpp:292

- **mat_vec_fns.cpp:223**
- **mat_vec_fns.cpp:250**
- mat_vec_fns_II.cpp:890
- NearestNeighbourLinkedListCube.cpp:27
- NearestNeighbourLinkedListCube.cpp:27 via LinkedListCube.tpp
- NoMassCGSolver.cpp:205
- NoMassCGSolver.cpp:233
- NoMassCGSolver.cpp:261
- NoMassCGSolver.cpp:278
- rod_math_v9.cpp:1067
- rod_structure.cpp:165
- **SparseMatrixFixedPattern.cpp:85**
- **SparseMatrixFixedPattern.cpp:148**
- **SparseMatrixTypes.cpp:52**
- tetra_element_linear.cpp:223
- **tetra_element_linear.cpp:299**
- tetra_element_linear.cpp:621
- **VdW_solver.cpp:168**
- **VdW_solver.cpp:168 via mat_vec_types.h**
- **VdW_solver.cpp:237**
- **VdW_solver.cpp:237 via mat_vec_types.h**
- VolumeIntersection.cpp:1202/1233[3]
- VolumeIntersection.cpp:1212/1239
- VolumeIntersection.cpp:1216/1241
- VolumeIntersection.cpp:1217/1242
- **World.cpp:4460**

There is a significant enough rate of cache misses that maintenance was considered warranted. While there are many lines with a far worst impact on cache misses, the particular line worth highlighting is `World.cpp:4460`. As mentioned in the section on memory, there is a memory leak surrounding the class `World()` in `World.h` that requires narrowing down. Memory leaks and cache misses *can* have a link (Rasmussen 2014; Shivakumar 2014), which may be something to use for this pinpointing.

`World.cpp` as to be expected includes `World.h` as a main header, and despite how massive `World.cpp` is to the point of having god object issues, this is the only line that shows up with

[3]Callgrind only states the function that caused a miss, rather than the line number, and VolumeIntersection.cpp has a lot of paired functions where it was unable to be discerned which out of each pair was being pointed to, hence pointing to both lines.

significant responsibility for cache misses. Therefore, the memory leak may be in the code in the header file that is utilised by the function at `World.cpp:4460`.

```
4460 void World::calc_blob_corr_matrix(int num_blobs,scalar *blob_corr) {
4461
4462     //calculates blob corrections for periodic interactions
4463     vector3 com,com2;
4464     for(int i = 0; i < num_blobs; ++i) {
4465         //sets blob distance from itself to 0
4466         blob_corr[i*num_blobs*3 + i*3]= 0;
4467         blob_corr[i*num_blobs*3 + i*3+1]= 0;
4468         blob_corr[i*num_blobs*3 + i*3+2]= 0;
4469
4470         active_blob_array[i]->get_stored_centroid(com.data);
4471         for(int j = i+1; j < num_blobs; ++j) {
4472
4473             active_blob_array[j]->get_stored_centroid(com2.data);
4474             blob_corr[i*num_blobs*3 + j*3]= box_dim.x*floor((com2.x-com.x+0.5*box_dim.x)/box_dim.x);
4475             blob_corr[i*num_blobs*3 + j*3+1]= box_dim.y*floor((com2.y-com.y+0.5*box_dim.y)/box_dim.y);
4476             blob_corr[i*num_blobs*3 + j*3+2]= box_dim.z*floor((com2.z-com.z+0.5*box_dim.z)/box_dim.z);
4477
4478             blob_corr[j*num_blobs*3 + i*3] = (-1)*blob_corr[i*num_blobs*3 + j*3];
4479             blob_corr[j*num_blobs*3 + i*3+1] = (-1)*blob_corr[i*num_blobs*3 + j*3+1];
4480             blob_corr[j*num_blobs*3 + i*3+2] = (-1)*blob_corr[i*num_blobs*3 + j*3+2];
4481
4482         }
4483     }
4484 }
```

Figure 3.11: `World.cpp` lines 4460-4484 containing the function `calc_blob_corr_matrix()`, which is the only function responsible for significant cache misses in this code file.

3.2.3 Threading analysis results

Threadcheck uses Valgrind's Helgrind to detect threading errors in the FFEA runner. These come under three main classes of errors:

- **POSIX Misuse (PM)** - A variety of errors that can occur from the misuse of POSIX (Portable Operating System Interface) pthreads functions.
- **Lock Order Violation (LOV)** - Occurs when multiple threads try to access the same locks in an inconsistent order and cause a deadlock.
- **Data Race (DR)** - Occurs when multiple threads access a shared memory location without proper synchronisation to ensure access is restricted to one thread at a time. Helgrind only reports them as *possible* data races due to detection of them being unreliable (Valgrind Developers 2022c).

Table 3.9 shows the count of the three classes of errors that occurred when running FFEA 2.7.2. Immediately it can be seen that for all datasets there were no recorded POSIX misuse or lock order violations. However, there were recordings of possible data races for the blob and many-blob datasets, but only on PC.

Table 3.9: Threadcheck results for number of thread errors in FFEA 2.7.2.

Dataset	**Number of Thread Errors**							
	PC				N8 Bede			
	PM	LOV	DR	Total	PM	LOV	DR	Total
B1	0	0	2,000	601,010	0	0	0	0
B2	0	0	750	10,000,000	0	0	0	0
B3	0	0	212	10,000,000	0	0	0	0
MB1	0	0	732	10,000,000	0	0	0	0
MB2	0	0	782	10,000,000	0	0	0	0
MB3	0	0	780	10,000,000	0	0	0	0
R1	0	0	0	0	N/A	N/A	N/A	N/A
R2	0	0	0	0	N/A	N/A	N/A	N/A
R3	0	0	0	0	N/A	N/A	N/A	N/A

Helgrind reports on data races (as well as the other error classes) through the log files it produces, an example of which can be seen in Figure 3.13. These show the size of the data race in bytes, whether this was during a read or write, and the call stack at the point of detection. It then shows the conflict that is believed to be racing with the first call stack.

```
==12048== Possible data race during read of size 4 at 0x6AE1354 by thread #8
==12048== Locks held: none
==12048==    at 0x557CE1B: UnknownInlinedFun (wait.h:57)
==12048==    by 0x557CE1B: UnknownInlinedFun (wait.h:66)
==12048==    by 0x557CE1B: gomp_barrier_wait_end (bar.c:48)
==12048==    by 0x557A43A: gomp_simple_barrier_wait (simple-bar.h:60)
==12048==    by 0x557A43A: gomp_thread_start (team.c:117)
==12048==    by 0x4C3C4E5: mythread_wrapper (hg_intercepts.c:398)
==12048==    by 0x59B7149: start_thread (in /usr/lib64/libpthread-2.28.so)
==12048==    by 0x5CCBF22: clone (in /usr/lib64/libc-2.28.so)
==12048==
==12048== This conflicts with a previous write of size 4 by thread #1
==12048== Locks held: none
==12048==    at 0x557CE8B: gomp_barrier_wait_end (bar.c:41)
==12048==    by 0x557CE8B: gomp_barrier_wait_end (bar.c:35)
==12048==    by 0x557B149: gomp_simple_barrier_wait (simple-bar.h:60)
==12048==    by 0x557B149: gomp_team_start (team.c:850)
==12048==    by 0x55706D0: GOMP_parallel (parallel.c:169)
==12048==    by 0x4679D4: World::run() (World.cpp:1842)
==12048==    by 0x44B0AE: main (ffea.cpp:187)
==12048==  Address 0x6ae1354 is 68 bytes inside a block of size 192 alloc'd
==12048==    at 0x4C33D31: malloc (vg_replace_malloc.c:380)
==12048==    by 0x556AD7C: gomp_malloc (alloc.c:37)
==12048==    by 0x557A655: gomp_get_thread_pool (pool.h:42)
==12048==    by 0x557A655: get_last_team (team.c:150)
==12048==    by 0x557A655: gomp_new_team (team.c:169)
==12048==    by 0x55706B9: GOMP_parallel (parallel.c:169)
==12048==    by 0x4679D4: World::run() (World.cpp:1842)
==12048==    by 0x44B0AE: main (ffea.cpp:187)
==12048==  Block was alloc'd by thread #1
```

Figure 3.12: Example of Valgrind's Helgrind's reports on a possible data race, detailing the size of the data race in bytes, whether it was during read or write, and the call stack where the data race was detected. It then provides the same details of the conflicting read or write.

Looking at the data race reports in the log files they are all of sizes 1, 2, 4, or 8 bytes. These were counted and placed into Table 3.10 to show the total possible sizes of these data races that were recorded, and these were then plotted onto a stack bar chart in Figure 3.13. The possible data races recorded only produced very minimal totals, with most being around 5KB but with a maximum of just over 15KB.

Table 3.10: Sizes of possible data races in FFEA 2.7.2

Dataset	Possible Size of Data Races During Read (B)	Possible Size of Data Races During Write (B)	Possible Total Size (B)
PC_B1	6,904	8,344	15,248
PC_B2	2,744	2,848	5,592
PC_B3	1,136	384	1,520
PC_MB1	3,746	1,022	4,768
PC_MB2	4,160	1,184	5,344
PC_MB3	4,184	1,160	5,344

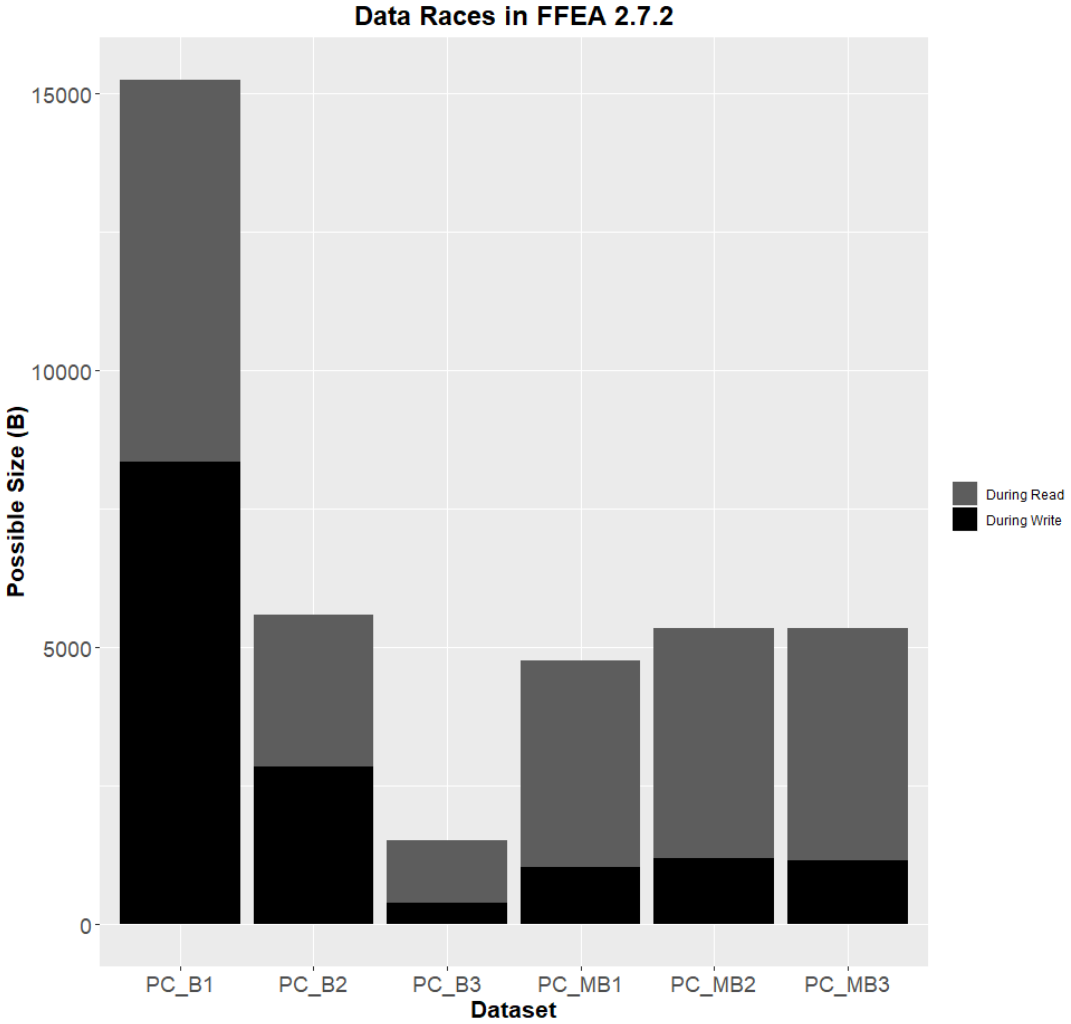

Figure 3.13: Stacked bar chart showing the sizes of possible data races in FFEA 2.7.2. during reads and writes.

Thread errors seem to be of very little concern to FFEA 2.7.2. The data races recorded were of very minimal sizes that only appeared in the blob and many-blob datasets for the PC, and these are only *possible* data races. When Helgrind searches for these potential data races, it will include any that occur due to the libraries used, so out of these possible data races, some are not a result of the FFEA source code. The report therefore concluded that fixing possible data races is of trivial priority.

3.2.4 Heap analysis results

Heapcheck produces information on the heap usage across runs of FFEA, showing the amount of heap used at particular time instances (called snapshots) and showing how much useful heap was assigned and how much extra heap was allocated for bookkeeping and alignment.

Heapcheck uses Valgrind's Massif tool, which produces `.out` files containing the above information. Each of these files starts with a histogram of snapshots of heap memory consumption, an example of which can be seen in Figure 3.14. These use a series of symbols to denote what was occurring at the time, which are the following:

- **:** - These bars represent normal snapshots, where only basic information is recorded.
- **@** - These bars represent detailed snapshots, where the later tables produced by Massif will show allocation trees (see Figure 3.15).
- **#** - These bars represent the peak snapshot where heap consumption was the greatest. These will also always be detailed snapshots.

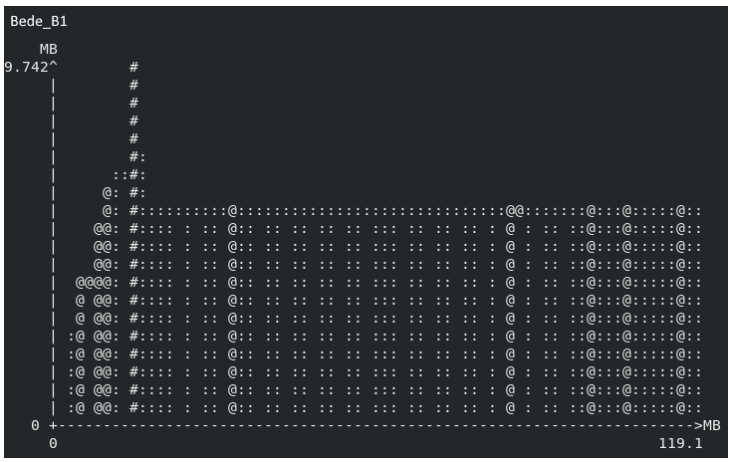

Figure 3.14: Example of a histogram of heap memory consumption produced by Valgrind's Massif tool. The x-axis shows the time, and the y-axis shows the memory consumption.

Below the histogram in the `.out` file is a large table of the heap information of snapshots, an example of which can be seen in Figure 3.15. The undetailed snapshots will simply show basic metrics of heap usage, such as in snapshots 0 and 1 in this example. Detailed snapshots like snapshot 2 show these metrics and then also the full allocation tree during that snapshot of when and where heap memory was allocated and deallocated.

```
--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  0              0                0              0             0            0
  1      2,928,392        2,610,088        2,605,683         4,405            0
  2      4,795,560        4,323,688        3,996,539       327,149            0
92.43% (3,996,539B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->50.07% (2,164,800B) 0x100A5FB7: Blob::load_topology(char const*) (Blob.cpp:2561)
| ->50.07% (2,164,800B) 0x100B74CF: Blob::init() (Blob.cpp:278)
|   ->50.07% (2,164,800B) 0x1006A1A7: World::read_and_build_system(std::vector<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >, std::allocator<std::__cxx11::basic_string<char, std
::char_traits<char>, std::allocator<char> > > >) (World.cpp:2661)
|     ->50.07% (2,164,800B) 0x1006C613: World::init(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >, int, int, bool) (World.cpp:438)
|       ->50.07% (2,164,800B) 0x1003D633: main (ffea.cpp:169)
|
->15.48% (669,128B) 0x100A70A7: Blob::load_surface(char const*, SimulationParams*) (Blob.cpp:2673)
| ->15.48% (669,128B) 0x100B74EB: Blob::init() (Blob.cpp:281)
|   ->15.48% (669,128B) 0x1006A1A7: World::read_and_build_system(std::vector<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >, std::allocator<std::__cxx11::basic_string<char, std::
char_traits<char>, std::allocator<char> > > >) (World.cpp:2661)
|     ->15.48% (669,128B) 0x1006C613: World::init(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >, int, int, bool) (World.cpp:438)
|       ->15.48% (669,128B) 0x1003D633: main (ffea.cpp:169)
|
->07.60% (328,648B) 0x100A5897: Blob::load_nodes(char const*, double) (Blob.cpp:2460)
```

Figure 3.15: Example of the heap usage tables for each snapshot produced by Valgrind's Massif. The tables state the runtime the snapshot was taken, the total heap memory consumption at that runtime, and the amount of useful and extra heap allocated at that runtime (the size of stacks are unrecorded by default as measuring this greatly increases the time Massif takes to profile the heap). When a detailed snapshot is recorded, the table is interrupted to display an allocation tree, showing which lines of code were responsible for allocating heap memory at that runtime.

From the results of Heapcheck, the total heap usage was extracted for each dataset, along with what amount of this was useful heap and what was extra heap, which can be seen in Table 3.11.

Table 3.11: Heap memory usage in FFEA 2.7.2

Dataset	**Useful Heap**	**Extra Heap**	**Total Heap**	**Percentage Extra Heap**
Bede_B1	5,986,195	352,693	6,338,888	5.89
Bede_B2	475,143,279	33,799,233	508,942,512	7.11
Bede_B3	113,009,658	3,014,638	116,024,296	2.67
Bede_MB1	21,796,357	1,441,435	23,237,792	6.61
Bede_MB2	224,061,070	14,549,154	238,610,224	6.49
Bede_MB3	463,925,556	29,093,868	493,019,424	6.27
PC_B1	5,922,884	352,332	6,275,216	5.95
PC_B2	475,159,448	33,024,928	508,184,376	6.95
PC_B3	113,035,385	2,826,354	115,861,712	2.50
PC_MB1	21,862,077	1,453,203	23,315,280	6.65
PC_MB2	224,041,958	14,518,698	238,560,656	6.48
PC_MB3	463,864,444	29,039,540	492,903,984	6.26
PC_R1	368,117	747	368,864	0.20
PC_R2	361,600	1,672	363,272	0.46
PC_R3	86,110,578	840,366	86,950,944	0.98

With these results, they were plotted onto two charts. The first being a stacked bar chart of

the heap consumption in FFEA 2.7.2 (Figure 3.16) and the second showing a percentage bar chart of the ratio of useful to extra heap consumption (Figure 3.17).

A few things can be noted from these results. First is that despite the datasets being designed to be of increasing computational sizes, the `B2` datasets have a far-higher heap consumption than `B3`. Rods also evidently require far less heap than blobs, with `PC_R1` and `PC_R2` being the only datasets where heap memory consumption is in the KBs, and only with the introduction of blob objects in `R3` increasing the usage on PC to MBs. Rods also seem to be more efficient with the heap, with extra heap only making up less than 1% of the total heap, compared to the blob datasets mostly being in the range of 6-7% with a lowest recorded usage of 2.5%.

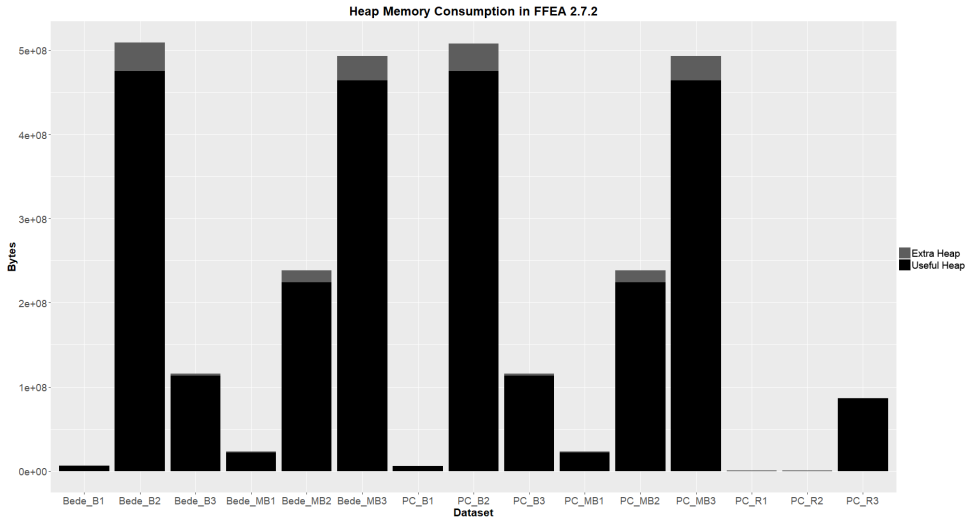

Figure 3.16: Bar chart showing the useful and extra heap memory consumption for FFEA 2.7.2.

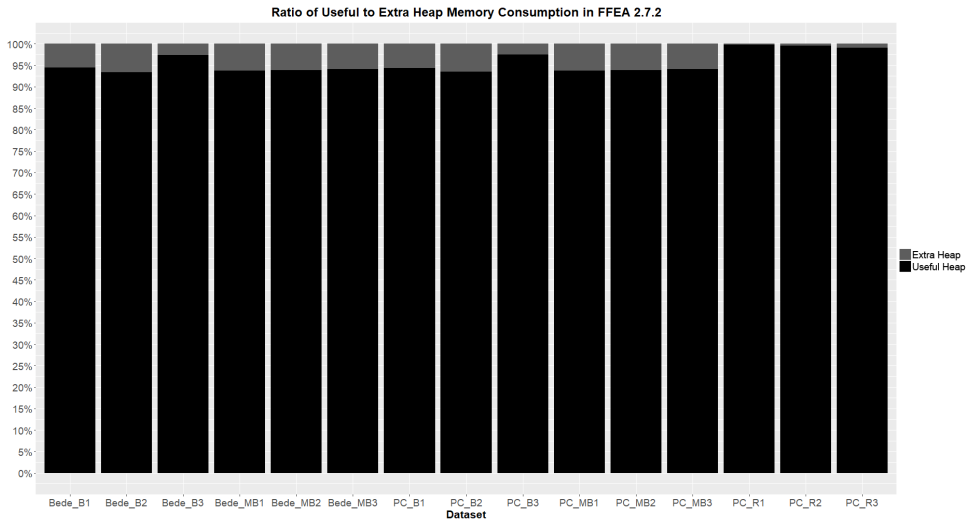

Figure 3.17: Percentage bar chart showing the ratio of useful to extra heap memory consumption for FFEA 2.7.2.

While heap usage in FFEA can likely be improved, the report did not recommended to consider it a top priority. It was noted that the rod code appears to manage heap much more effectively than blob code, so if it ever does become something the developers wish to perform maintenance work on, they can look to the rod code to see how the blob code should change to be more heap efficient.

It was also noted that heap usage is not correlated to the way the datasets define size. When picking these datasets the idea was to have them increase in size based on the amount of computational requirement and the general time it takes for simulations to complete, such as by having the "larger" datasets include interactions between simulated objects. While it could not be determined what heap usage *does* correlate to without further investigation, an initial guess was that it correlates to the size of the mesh, as `B2` has a physically much larger mesh object than the combined size of the meshes used for `B3` (see Table 3.3). This may become a problem later down the line as larger and larger objects are meshed to be used in FFEA, as the difference between the heap usage of `B2` compared to `B1` and `B3` is quite drastically large.

3.2.5 Dynamic code analysis conclusions

From analysing the results of the FFEA_profiler it was recommended that fixing the memory leaks should be considered the highest priority out of the issues discussed, especially the leak that occurs at `ffea.cpp:165`. The leak(s) may be responsible for potential concerns with cache misses and heap usage, which is why they should be concentrated on first.

Despite this recommended priority order, it is still worth using the other profiling results, especially those on cache usage, as they may potentially be useful in cross-referencing parts of the source code to determine likely origins of issues, such as the aforementioned discussion on how the memory leak at `ffea.cpp:165` may be related to the cache misses at `World.cpp:4460`.

3.3 Soak testing

To allow for the analysis of very complex or large biomolecular systems, FFEA simulations can be set to run continuously for large periods of time. However, it is possible for software to encounter errors with functionality when reaching long runtimes, such as through slowly-incrementing arithmetic errors or running out of resource allocation. This is a possibility that can be explored through soak testing, where the software is ran for an extended period of

time (longer than in typical use) to test for errors that result from accumulated and otherwise-unassuming anomalies (Youree 2023). This is a similar method to stress testing, in which the performance of the software is evaluated when given workloads larger than it is expected to conduct in typical use (or with reduced access to computational resources) (Homès 2024). In soak testing, the high load is the long runtime, and other stressful loads can be applied during this long runtime. To ensure that FFEA does not encounter such problems, FFEA must be soak tested under a long runtime to see if it continues to perform optimally and correctly under these conditions.

Soak testing was performed by running each dataset from the `FFEA_profiler` (§3.2) for 48 hours on N8 Bede, three times for each to gain mean results. This was only done on Bede because nine datasets running three times for 48 hours each is a total runtime of 54 days, and while Bede can run multiple tests simultaneously, the PC used for profiling could not. The `check` parameter was set to 1000 and `num_steps` was set to a large enough number to make it effectively infinite for this test. From each test, the number of steps reached, the wall clock time (WCT), the simulation time, and the stop condition were recorded. Note that simulation time is affected by the time step of each dataset, which can be found in Table 3.3, and therefore is not useful for direct comparison to other datasets but *is* useful for comparing against other runs of the same dataset and seeing how computationally-intensive individual steps are. The soak tests encountered three conditions for stopping the simulation, which are as follows:

- **Time Limit (TL)** - The 48-hour time limit imposed by the soak test was reached. This should be considered a successful test.
- **Element Inversion (EI)** - The simulation crashed due to an element inversion. This usually is the result of an element having too high an energy value, often due to an element being too small or thin. However, there may be other less-clear causes.
- **Rod Dynamics Explosion (RDE)** - A rod has gained an infinite (or sometimes just a very large) energy value and caused the simulation to crash.

Figure 3.18 shows an example of the end of one of the output files from soak testing where the results are obtained. The result that will be recorded is the last instance of a full 10,000 steps as that is the last record of the simulation time and wall clock time.

```
step = 124361000
step = 124362000
step = 124363000
step = 124364000
step = 124365000
step = 124366000
step = 124367000
step = 124368000
step = 124369000
step = 124370000 (simulation time = 12437.00ns, wall clock time = 48.000 hrs)

step = 124371000
step = 124372000
step = 124373000
step = 124374000
step = 124375000
step = 124376000slurmstepd: error: *** JOB 244543 ON infer001 CANCELLED AT 2021-07-21T13:02:48 DUE TO
TIME LIMIT ***

step = 124377000
```

Figure 3.18: Example results from a soak test for FFEA 2.7.2., showing the number of simulation steps, the simulation time, and the wall clock time when the simulation terminated. Note that simulation time and wall clock time is only recorded at every ten checkpoints in the FFEA simulation (in this case every 10,000 steps), hence why the recorded results for soak testing comes from the last instance of a full 10,000 step cycle in the FFEA runner.

3.3.1 Results

Table 3.12 presents the results of the soak tests for FFEA 2.7.2. The first run for `R1` was not used for calculating averages due to being a clear outlier. The wall clock times were then plot onto a bar graph in Figure 3.19 for comparison.

Table 3.12: Results of soak tests on FFEA 2.7.2.

Dataset	Steps Reached	Average Steps	WCT (Hrs)	Average WCT (Hrs)	Simulation Time (ns)	Average ST (ns)	Stop Condition
B1	124,370,000	124,156,667	48.000	47.993	12,437.00	12,415.67	TL
	124,070,000		47.974		12,407.00		TL
	124,030,000		48.004		12,403.00		TL
B2	990,000	990,000	47.905	47.777	99.00	99.00	TL
	990,000		47.899		99.00		TL
	990,000		47.528		99.00		TL
B3	20,000	66,667	4.949	16.990	20.00	66.67	EI
	50,000		12.124		50.00		EI
	130,000		33.896		130.00		EI
MB1	6,870,000	6,930,000	47.950	47.952	6,870,000.00	6,930,000.00	TL
	7,090,000		47.941		7,090,000.00		TL
	6,830,000		47.966		6,830,000.00		TL
MB2	130,000	130,000	44.993	45.747	130,000.00	130,000.00	TL
	130,000		46.155		130,000.00		TL
	130,000		46.093		130,000.00		TL
MB3	40,000	40,000	45.642	45.621	40,000.00	40,000.00	TL
	40,000		45.490		40,000.00		TL
	40,000		45.730		40,000.00		TL
R1	1,070,000	87,010,000	0.586	48.002	10,700.00	870,100.00	RDE
	86,930,000		48.005		869,300.00		TL
	87,090,000		47.999		870,900.00		TL
R2	54,410,000	53,810,000	48.000	47.984	16,323.00	16,143.00	TL
	53,360,000		47.961		16,008.00		TL
	53,660,000		47.991		16,098.00		TL
R3	210,000	230,000	47.150	47.133	63.00	69.00	TL
	240,000		46.544		72.00		TL
	240,000		47.705		72.00		TL

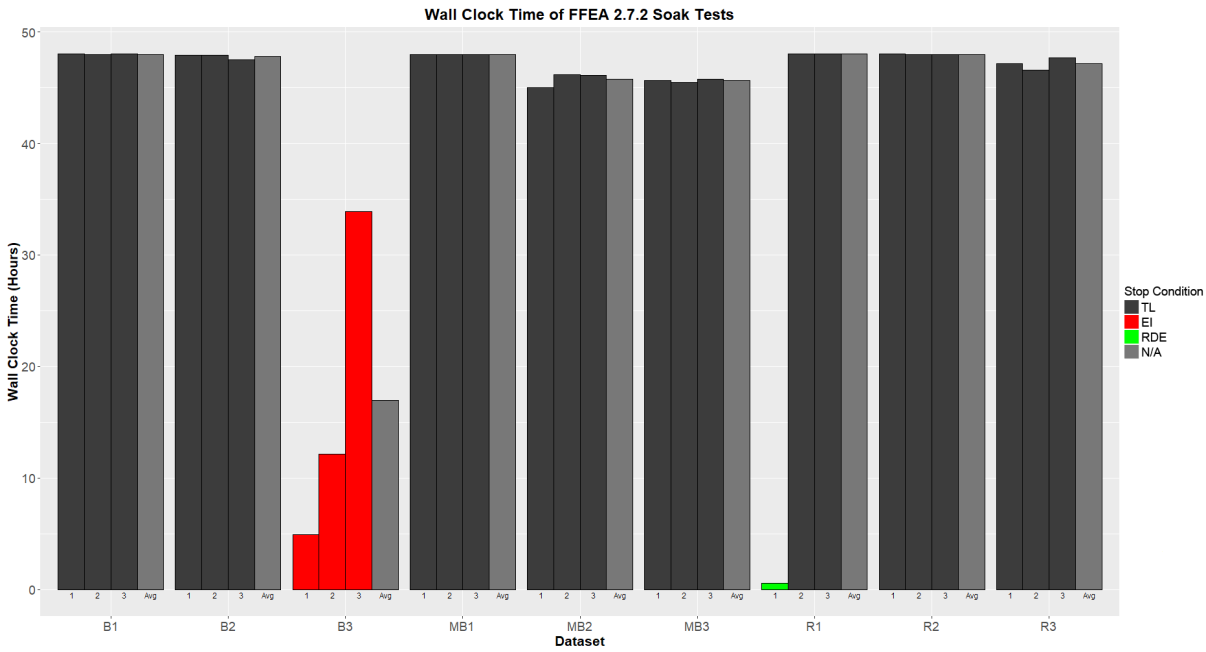

Figure 3.19: Bar chart showing the soak test wall clock times in FFEA 2.7.2. Bars are grouped by dataset showing the results of each individual three runs followed by an average result. The colour of the bar indicates the stop condition that terminated the FFEA simulation (with the exception of average results).

Most of the datasets managed to pass the soak test and run until the time limit was reached. Some datasets that reached that limit did not quite reach a 48-hour wall clock time or close to it. This can be seen as due to these computationally larger datasets taking much longer to calculate each individual step so that every 10,000 steps sometimes took hours. It can clearly be seen with `MB3` for example, having only reached 40,000 steps within 45.621 hours, compared to the smaller dataset `MB1` that managed to reach just under 7,000,000 steps in 47.952 hours.

Some runs did not pass the soak test. `R1` experienced an RDE error in the first run, but this did not reoccur, hence why this run was considered an outlier. Subsequent runs of `R1` reached the time limit, so it seems as if the occurrence of this error does not necessarily indicate a problem with a dataset.

`B3`, however, experienced an EI error in each run. The wall clock time the error occurred at varied vastly at approximately the 5, 12, and 34-hour marks. This shows a seemingly random chance of when EI errors can occur. When looking at the log files of the standard output from each run of the soak tests of `B3`, there were two elements that inverted. Respectively, the IDs of the elements that inverted each run were 1602, 979, and then 1602 again, so it can be seen that it is not necessarily the same element that first reaches a high enough energy value to invert

when running the same FFEA simulation. Of course, with these errors only happening in one of the datasets in this experiment, this cannot provide a statistically valid analysis of the EI error, so further study would be required for understanding this error in great detail.

Both types of failing stop conditions recorded were due to software errors specific to FFEA. There were no hardware-related crashes. What this suggests is that despite any problems that were found during the profiling of FFEA 2.7.2, such as the significant memory leaks, none of them are enough of a problem to be causing the software to crash. In order to test this claim, the soak tests were re-run but with the total memory available limited to 32GB, the same amount of memory as the PC used for profiling. Table 3.13 and Figure 3.20 present the results of the soak test re-runs.

Table 3.13: Results of soak tests on FFEA 2.7.2. with 32GB RAM limit.

Dataset	**Steps Reached**	**Average Steps**	**WCT (Hrs)**	**Average WCT (Hrs)**	**Simulation Time (ns)**	**Average ST (ns)**	**Stop Condition**
B1	125,510,000	123,646,667	48.007	48.002	12,551.00	12,364.67	TL
	124,110,000		48.002		12,411.00		TL
	121,320,000		47.997		12,132.00		TL
B2	990,000	990,000	47.789	47.780	99.00	99.00	TL
	990,000		47.781		99.00		TL
	990,000		47.769		99.00		TL
B3	20,000	53,333	4.947	13.039	20.00	53.33	EI
	0		0.000		0.00		EI
	140,000		34.169		140.00		EI
MB1	6,680,000	6,793,333	47.982	47.980	6,680,000.00	6,793,333.33	TL
	6,690,000		47.988		6,690,000.00		TL
	7,010,000		47.971		7,010,000.00		TL
MB2	130,000	130,000	45.276	45.423	130,000.00	130,000.00	TL
	130,000		45.549		130,000.00		TL
	130,000		45.443		130,000.00		TL
MB3	40,000	40,000	45.674	45.988	40,000.00	40,000.00	TL
	40,000		46.663		40,000.00		TL
	40,000		45.628		40,000.00		TL
R1	87,760,000	87,426,667	48.000	47.994	877,600.00	874,266.67	TL
	86,880,000		47.987		868,800.00		TL
	87,640,000		47.994		876,400.00		TL
R2	54,130,000	54,203,333	47.998	48.002	16,239.00	16,261.00	TL
	54,290,000		48.003		16,287.00		TL
	54,190,000		48.005		16,257.00		TL
R3	240,000	226,667	46.909	46.647	72.00	68.00	TL
	230,000		46.205		69.00		TL
	210,000		46.827		63.00		TL

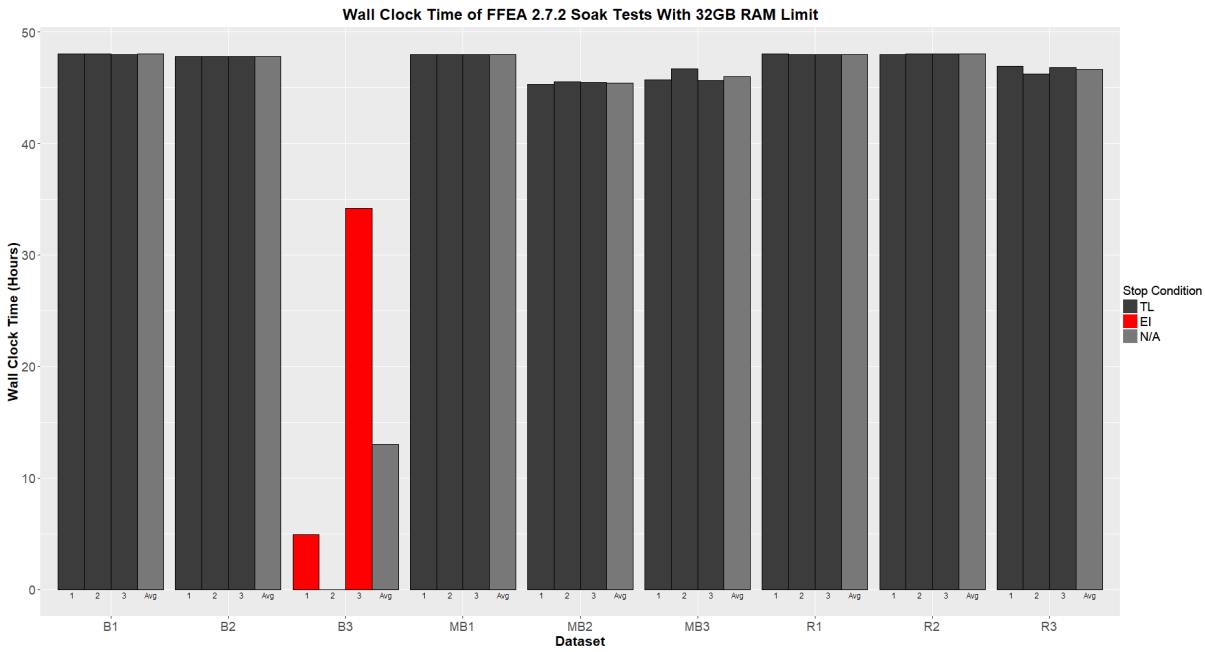

Figure 3.20: Bar chart showing the soak test wall clock times in FFEA 2.7.2. with 32GB RAM limit. Bars are grouped by dataset showing the results of each individual three runs followed by an average result. The colour of the bar indicates the stop condition that terminated the FFEA simulation (with the exception of average results).

To compare the two sets of soak tests, the average wall clock times were plotted against each other in a bar chart which is presented in Figure 3.21.

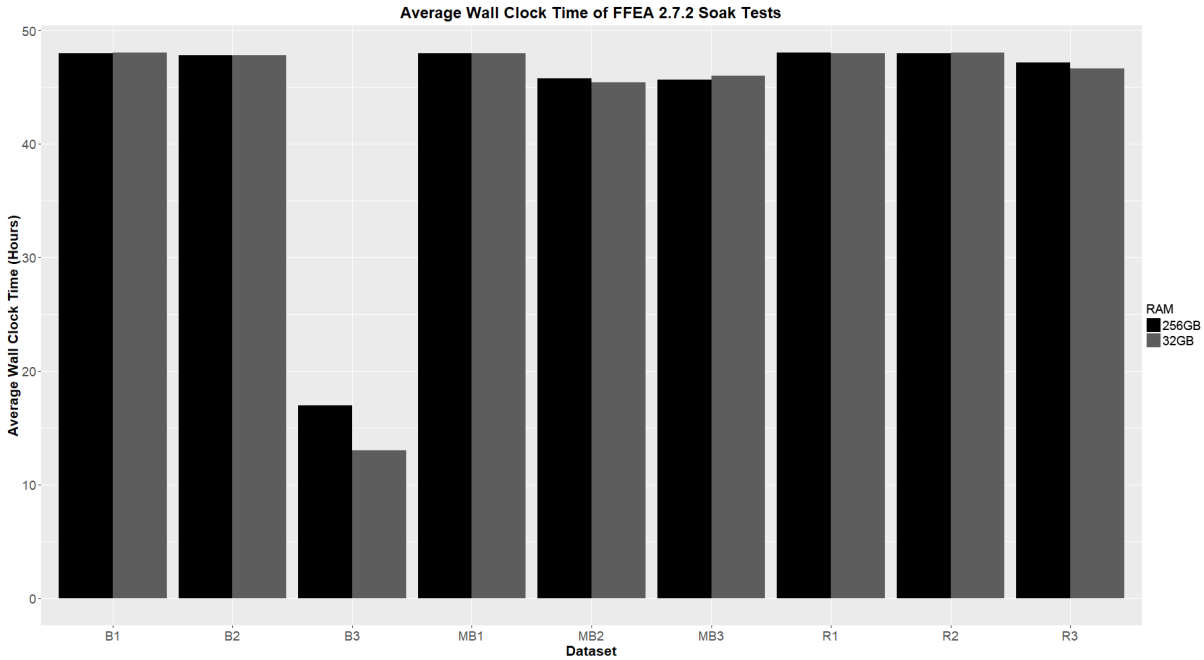

Figure 3.21: Bar chart showing the comparison of average wall clock times between the two sets of soak tests with RAM limits of 256GB and 32GB.

The datasets appear to behave the same for both soak tests for the majority of cases. `R1` this time did not have any runs produce an RDE error, which is expected as it appeared to be an outlier in the original soak testing. The average wall clock times for each dataset are very close to the same for each set of soak tests, with the slight exception of `B3` which has a difference of 3.951hrs between the sets of tests. This is also expected, however, as the random nature of the EI error as discussed previously could easily cause such a difference, so this would still suggest the dataset behaves the same in both cases. Due to this similar behaviour, and with no introduction of memory-related stop conditions, this backs up the earlier suggestion that the memory leaks are not causing software crashes. Therefore, while there are performance problems within FFEA 2.7.2, it seems they are not fatal problems.

3.4 Quantitative analysis conclusions

In this chapter, the evaluation of FFEA has begun in order to attempt to answer the first question in the research scope (§1.1), to see how a holistic understanding of a piece of research software can be gained in order to determine its sustainable qualities.

The results of this quantitative analysis have highlighted a few factors of FFEA 2.7.2's source code that could be impacting its sustainable qualities. The static code analysis has shown that the source code does not adhere to the conventional standards for programming language styles, which may have an impact on development by reducing the readability of the code (an important factor of software sustainability as noted by the survey of RSEs conducted by Souza et al. (2019) discussed in §2.1.2). The Python source code for FFEAtools had especially poor linting scores, which could not simply be standardised due to the heavy use of outdated Python 2 code, and requires a full re-implementation into a new and more modern project structure. If these issues go unaddressed, they could impact further development of the FFEA source code and source code maintenance (important factors of the first pillar of RSE as shown in Figure 2.2) by making the source code more difficult to work with.

The dynamic code analysis has shown that FFEA 2.7.2 has minimal resource-based issues, but that there exists a memory leak that can cause potentially megabytes of lost memory depending on which part of the source code is ran in order to simulate input data. While this is significant enough to be addressed and is certainly the highest-priority issue found here for software optimization, it remains to be seen how much this is impacting the sustainability of

FFEA 2.7.2 beyond a slight inefficiency of the code. The results of the soak tests performed suggest that it is not causing any major bottlenecks in the running of the software as no hardware-related crashes were found. There were however some run failures due to a physics bug with the simulation called an "element inversion", which implies a potential bottleneck-causing issue in FFEA 2.7.2's functionality. While this only occurred in one of the test datasets, it should be investigated how often this error seems to occur for users when performing the qualitative analysis in the next chapter.

The selection of tests performed in this chapter have given a good overview of the state of FFEA 2.7.2's source code, and the factors that may be impacting its sustainable qualities. However, there are some further tests that could have been made to investigate this deeper. In regards to the memory leak found during dynamic testing, the tests performed were not able to locate the exact cause of this, only the area of the source code it potentially resides within. Theoretically this could have been found with either deeper analysis of Valgrind's results or by using a more in-depth dynamic code analysis tool such as Score-P (Knüpfer et al. 2011). Beyond soak testing, further types of stress testing through different loads on resources may also have assisted in determining more information about this memory leak and how likely it is to hinder or halt simulation progress.

One method not performed during the static code analysis was to measure FFEA 2.7.2's source code against established software quality metrics, as outlined by such standards as ISO/IEC 5055:2021 (ISO 2021). This would have provided a better set of measurable qualities to analyse FFEA 2.7.2 and compare to either other research software, or to later versions of FFEA to see how the software quality changes with each new version. This was a mistake in the design of the static code analysis methodology due to putting more consideration into the specific case of FFEA than into designing a generic method for general research software. If this research were to be re-performed or iterated upon, this would be added to the general evaluation method.

Conclusions on FFEA 2.7.2's sustainability as a whole cannot yet be drawn as qualitative analysis is first needed, which is conducted in the next chapter. Once done, those results can be compared to the results from the quantitative analysis in this chapter, and from that it can be determined what are the major sustainability-impacting factors that are hindering the user experience and causing bottlenecks in the workflow, which will be discussed in §4.4.

Chapter 4

Evaluation of modern novel research software – A case study of fluctuating finite element analysis. Part II: Qualitative analysis and evaluation conclusions.

This chapter continues discussing the evaluation of FFEA 2.7.2 by detailing the qualitative findings from Rogers (2021a) relevant to this thesis, i.e. the "Qualitative Analysis" substage of the general method as outlined by Figure 1.1. This analysis was performed by surveying users and developers to identify issues with the software through analysing their experiences. This is then used to determine issues that may impact FFEA's sustainable software qualities, and to gain opinions from FFEA users and developers on what may improve such qualities.

After conclusions are discussed on this qualitative analysis, this chapter then finishes with a holistic discussion of the evaluation of FFEA 2.7.2 performed in this chapter and the previous one to identify the key bottleneck(s) in FFEA's workflow which negatively impact its sustainable qualities, as the last step in the "Evaluation of research software" stage of this research (Figure 1.1).

4.1 Survey design

To qualitatively evaluate FFEA 2.7.2, a series of questions were devised to ask users and developers about their experiences with it, either via a written questionnaire or an interview. The questions for users ask participants to describe their workflows and give feedback based on what aspects have been easy or satisfying to them, and what aspects cause difficulty, specifying questions for each major component of the FFEA software suite. Developers are asked similar questions but about their development workflows. A copy of the questionnaire (which contains the same questions as the interview) can be seen in Appendix A.

The design of the questions took influence from the list of evaluation scenarios presented by Isenberg et al. (2013). Most questions were centred around understanding environments and work practices, and user experience, as one of the bigger priorities of this evaluation was to understand the workflows of participants, to build up an understanding of what areas of the FFEA software suite cause the most problems for them, but also which areas they deemed praiseworthy so that it could be seen what aspects may or may not require change. Questions were also asked that briefly discuss algorithm performance and user performance, but not as in-depth as that scenario usually calls for, as this analysis was not for measuring objectively the participant's performance. The algorithm's performance was measured as a part of the quantitative analysis in Chapter 3, but it was desirable to understand participants' opinions on their ability to perform with FFEA 2.7.2 and get a general idea of the length of wait times. These are the generic aspects of research software that are needed to be studied to perform relevant qualitative analysis and to understand its current sustainable qualities. However, specifically to the case study of FFEA, questions were also included on communication through visualization and collaborative data analysis to gleam some information on the FFEA software suite's communicative abilities, as this is a key aspect of this particular software.

The questions asked were designed to be very open, with most closed questions being followed by asking why the participant chose that answer. This is because it was expected that this research would not get many participants, due to FFEA being a fairly niche tool, so it was considered best to try and get as much detail as possible from each participant, rather than focus on the statistical analysis of closed questions with a tiny sample size. Charts will still be presented of the results to the closed questions, however, but this small sample size should be kept in mind.

Those invited to participate in this survey were selected as known users of the FFEA software suite, either as part of research work or educational studies. Five people agreed to participate in this study, and of those five, two participants also had experience in developing the FFEA source code within the past 12 months prior to the survey. All participants had experience with scientific research, but participants had mixed levels of experience with software engineering, some having none at all.

Answers to survey questions were collected either by collecting participants' written answers, or by recording and transcribing an interview with them. Three participants provided written answers and two participants took part in an interview. The answers to closed questions were collated into a spreadsheet for analysis. For binary yes/no questions, it was simply counted how many participants answered each way. Ranked questions were on a scale where a participant could extremely disagree, highly disagree, disagree, agree, highly agree, or extremely agree with a positively-worded statement (no neutral option was provided as participants were allowed to not answer a question they had no opinion of). Each ranking was assigned a numerical value from 1 to 6 respectively, so that ranked questions could be plot onto bar charts, and so that a mean average answer for each question could be calculated. For each open question, the key points made by each participant were extracted from their written answers or transcript. These key points were then compared to see what commonalities arose, and which participants were providing unique answers. These were then surmised in the results, where the key points are highlighted in **bold**.

4.2 Survey results

This section is dedicated to presenting the results from the qualitative survey from the five total participants (two of which were developers) that provided answers. These interviews and questionnaires were conducted/collected between late February and early May of 2021. Questions specified the part(s) of the FFEA software suite they were in reference to in order to get a clear distinction between issues/praises for different components of the suite.

4.2.1 User responses

FFEA runner and tools

All participants were asked questions about their experiences with FFEA as a user, beginning with questions about the FFEA runner and FFEAtools. First, participants were asked about what functionality in the FFEA runner and FFEAtools they use, and their workflow for using them. **A common workflow was the use of FFEAtools to create tetrahedral and rod models for FFEA**, often from obtaining electron microscopy maps from atomistic structures (usually PDBs) and then extracting surface profiles and coarsening them. Afterwards, they would gain a volumetric mesh either with FFEAtools or externally with TetGen (Weierstrass Institute for Applied Analysis and Stochastics 2013). These would then be used as input files for running simulations with the FFEA runner. **All participants mentioned this workflow in some capacity**. One participant mentioned the use of the rod parameter extraction file to determine flexibility parameters of the rods that represented sections of the protein they were analysing (another participant also mentioned a minute use of `FFEA_rod`). Another mentioned their workflow involved analysis by accessing information on node positions in their simulation and writing scripts to track the distance between two surfaces.

Participants were next asked to provide an estimation of the time it takes to set up their workflow and how long it takes the FFEA runner to generate results from that. **For the setup, the answers range from a timeframe of hours to months**. Less complex meshes and simulations appear to take a range of 30 minutes to a few hours according to participants, but for the more complex (i.e. often the simulations their main project revolves around) this can take weeks or months. Some participants specified a difference between the setup and assembly of data. **It is the assembly that can take long periods of time**; the creation of meshes, setting up interactivity between objects, the positioning of objects (which one participant described as a very trial-and-error process). For obtaining results from the FFEA runner, for smaller simulations (such as of equilibrium structures) this reportedly takes less than an hour. For their larger simulations that they expect important results from, this reportedly ranged from two hours to a few weeks, with the most common answer from two participants saying it is in a timeframe of multiple days. One participant gave a detailed example. They had a simulation of three rod objects connecting to a cargo object and binding to another. The time step was set to 3e-13s, the maximum time was set to 60ns, and the number of threads was set to eight.

Running this simulation reportedly took around 24-36 hours.

The next set of questions asked participants what parts of their workflows they found easy, satisfying, difficult, and unsatisfying respectively. Two participants mentioned that **running a simulation in FFEA is easy**, with one of them specifically mentioning that analysis of the results is quite easy (such as analysis of trajectories through the FFEAtools trajectory module), and a different participant mentioning that analysis scripts that mainly plot graphs are easy to write as "a lot of it is boilerplate code used from old projects". Two participants said they found **mesh generation is easy** with one attributing that to good documentation. Both also said rod generation is easy, with one saying it was because all that a user needs to do is run a single Python script to get a rod. One participant also said they found **writing HPC submission scripts for FFEA is easy**, as the queue system would instantly reject invalid FFEA scripts, and that old scripts can be ported to other HPC systems fairly easily, with the main challenge for HPC being sending files to the correct folders. One participant also mentioned that **adjusting existing parameters in an FFEA file is easy** as changes rarely resulted in difficult errors.

For satisfaction, **two participants gave praise to `FFEA_rod` for being easy-to-use Python code with intuitive and clear functionality**, with one specifically mentioning they liked that they could preview rods in Python as an easy check, and the other saying they found their workflow for it easier due to being able to use a tool like Spyder (Spyder Doc Contributors 2023) to auto-complete certain commands. **The participant that mentioned the ease of trajectory analysis also stated they found that made it satisfying to them**. Two participants also mentioned that **simply getting a working simulation that finished correctly to be satisfying**. One participant stated **satisfaction in making particularly complicated figures during analysis** or ones with visual appeal such as heatmaps. One participant when asked this question said that they did not know how to characterise the workflow of any scientific software to be satisfying or dissatisfying and that they all require a degree of trial-and-error. They said that the workflow for FFEA is difficult because the problem they are solving when creating FFEA systems is difficult, but **it is that problem solving that is satisfying**; the end result.

For things that participants find difficult in their workflow, two participants stated **difficulty in the creation of a mesh or model** (this may not be a disagreement with the participants

that stated mesh *generation* is easy). One said that **making a mesh that works the first time is a nontrivial process and that initialisation of an FFEA system is the hardest thing for them**. The other said that **creating a regular model took many steps, it requires manual adjustment of meshes and that setting up surface-surface interactions is "cumbersome"**, the latter being a difficulty mentioned by another participant. **Four participants specifically mentioned positioning causing difficulties for them**. Complaints include that positioning rods in the simulation box is buggy, that aligning FFEA structures to an atomistic/cryogenic one is difficult, that it is hard to gauge scaling and set up an appropriately sized box in the FFEA file, and that positioning is a very trial-and-error process exacerbated by it being done though text files rather than via a graphical program, making multi-molecule scripts very difficult to set up properly. Another difficulty one participant stated was that **while the interface for creating rods is easier, they behave differently to blobs as they sometimes ignore parameters in the FFEA script file**. Finally, one participant discussed how it was **difficult for them to keep track of all the files generated throughout their workflow**, and the function of each of these files for the blobs.

For parts of the workflow participants found unsatisfactory, **two participants had complaints relating to simulations stopping part way through due to a bug in FFEA that causes elements to invert** (which is mentioned in more detail when participants are asked about bugs later in this subsection). One of these participants said that FFEA is an exception to other software where "once it starts running, you expect it to finish running", and you have to keep an eye on your FFEA simulations to ensure it *stays* running. The other also said that this **manual requirement of having to check on their simulations is a nuisance**, especially for multiple long simulations as fields need to be altered in the input scripts and files need to be renamed. They also noted that the restart method is inconsistent between blob and rod simulations and expressed a desire to see this process automated if possible. Building off that, the same participant said that **the trial-and-error nature of editing FFEA input scripts is unsatisfying** as many small things can go wrong at different stages, and the error messages produced by the runner are not always helpful. One participant stated frustration that **there is no interactive method for positioning objects** how you want and being able to save that system to have a ready-to-go setup. The ability to use a mouse for drag-and-drop commands would be much easier to them than dealing with some of the problems they have to manually overcome, such as figuring out positioning issues due to rods using SI units and blobs using

FFEA's custom units and that rods and blobs have different origins where they first initialise in a system, as well as having to calculate rotation matrices to position something. Finally, one participant built on their difficulty with the difference between scaling the rods and blobs:

> *The scale of the rod was 1 because it had already been scaled in python, but the scale of the blob was much smaller in the FFEA file. This was not a major issue but would be easier to cross-check the sizes if they were in one place.*

Participants were then asked if they had developed any workarounds to assist with difficult and unsatisfying tasks in their workflow. Three participants mentioned **the use of UCSF Chimera** (Pettersen et al. 2021), but each for different reasons. One participant said that they found it easier to use that or VMD (Humphrey et al. 1996) to extract surface profiles over using FFEAtools. Another said they used it to figure out positioning based on the PDB files they generated meshes from and then used those positions in FFEA effectively. Finally, one participant considered their best tip to users is to use Chimera to smooth tetrahedral meshes before coarse-graining them. This same participant also mentioned **trying to automate the process of making these tetrahedral meshes, but found mixed success**, saying that the limiting factor was the algorithms rather than the software. One participant also mentioned **a workaround for rotations due to not being able to do so interactively with a mouse**. They used constant forces to apply a linear rotational force, simulated it, and then used the results from that to move the system into the position they wanted, which they considered to be a complex workaround for something that should be simple. One participant said **they had written a Python script to generate FFEA input scripts in a modular fashion**, such as using it to add entries for molecules that consist of a rod with two blobs connected to each end and the two interfaces. They also said **they used BASH scripts to restart simulations for them** instead of manually having to change fields in the FFEA input files. This script would rename the relevant files and resubmit the FFEA simulation to the HPC system they were running it on. Finally, one user mentioned that **they used external modelling software** such as Blender (Blender Online Community 2023) **to check for problems with their meshes**, such as non-manifold surfaces, though they mentioned this as a general part of their workflow in a previous question.

Participants were then asked to note any bugs they had encountered during their commonly performed tasks. First of all, a participant said that **rods do not respect most global FFEA**

settings, such as how they will resume from checkpoints even when told not to. One participant said that while they did not encounter any bugs, they raised concern over the **FFEA units being difficult to understand** and that in a fair few cases such as utilising van der Waals interactions this would mean the user would have to understand in detail how the software works (at the conceptual level not the source code level). This is unlike other simulation techniques such as molecular dynamics (MD) where users do not need to know this level of depth to use common functionality. Two users pointed out some **difficulty during installation**, such as clashes with certain versions of GCC for compilation and getting import errors after installing FFEAtools due to its use of the now end-of-life Python 2. According to one participant, the most common bug in the FFEA runner is **the inversion of elements that causes simulations to crash**. These result when elements have too high energy, which can often be due to them being too small or thin, but other unclear factors appear to be a cause. They said this can often be solved by increasing the time step of the simulation or increasing the size of particularly small objects. Otherwise, they will adjust the mesh in an external program such as Blender to remove these small or spiky elements. Finally, one participant said a less common bug for them was that **when rod-blob simulations are finished and visualized, they will appear in the corner of the simulation box with large amounts of empty space, and when played the rods would greatly deform** and sometimes "ping" into the corner of the box. They said they had no idea of the cause and recall getting around it by just removing whatever changes they made in the input files that started causing this problem.

The next question asked if the users had experience with collaborative work using FFEA, and how well they think FFEA accommodates collaborative work. In general, participants leaned towards a favourable answer. Two participants said that **it was very doable for multiple people to work independently on different parts of the workflow** such as parametrising, experimental imaging, meshing, simulations, and analysis. However, one of them mentioned that **difficulties in meshing complicated systems and its slightly messy documentation and codebase can make it difficult to introduce new people to FFEA**. One of them also said that **models and systems are small enough to conceivably put them under version control**, but they had not tried it. Another participant said that "it's easy to do that as it is with any other software package", as long as collaborators have the same files and version. One participant said that while they had not done collaborative work, **they found their rods work easy to discuss and detail what is happening at each step due to the**

clear labels in the Python scripting. Although, one participant did mention that FFEA has a lot more files than other MD simulations, so there is more to send across to another user rather than a single file to run. They also stated that **the main difficulty for collaboration for them was doing it across different operating systems** (specifically macOS to Linux).

The final question for this section is if there are any features participants can think of that if added to FFEA would reduce the difficulty and dissatisfaction they find doing certain tasks. Three participants mentioned the introduction of **graphical tools for interaction parts of the process** such as model building, setting up systems, setting surface interactions, and creating and connecting rods. One participant explained their frustration by saying that instead of being able to click on a node to assign a parameter such as pinning, they have to find the node ID they want to interact with and then manually write the parameter in the appropriate text file. Three participants stated that **they wanted certain parts of their workflow to be automated**. One said that they wished the entire initialisation process to be automated so that a user could feed in a PDB or EMDB file into FFEA and then have it return a ready-to-go FFEA surface having given some parameter information such as the level of detail of the surface. This participant described the current process as a hassle that requires far more knowledge of the functions of the software than other simulation software and also often requires the use of third-party software to fix certain things...

> *[A]t no stage should I, the user, have to know what the code is, at all[...] I shouldn't need to know how it's making a surface, it should just be able to do it.*

Another user expressed that generating blobs may be easier if during the process of generating files, they are named based on the input file by default with the option to overwrite the name. If they could be output into a new folder automatically, that may help with keeping track of what stage the user was at and make for easier management of FFEA files they reasoned, however, they described that as a minor inconvenience. Finally, the participant that earlier expressed a complaint about having to restart simulations manually briefly mentioned here a desire for automated restarting. **Another feature suggested was the ability to set equilibria**, i.e. allow a user to give FFEA a topology file and set it to be the equilibrium state, so that the user can interact with the system during simulation, such as applying a push force on a protein to see what happens. They added that **if FFEA could run in virtual reality (as other MD techniques have tried) it would be a further boon to interactivity** during a running

simulation. The participant that earlier stated they had issues with scaling said **they wish that the scaling was the same for rods and blobs** for the sake of comparison, or failing that if FFEA had a command that would output the scale of the blob in the same Python format as the rods. They note that this is not an issue when using a PDB structure to generate rods as in that case the scaling of the rods and blobs are equal. Finally, one participant briefly mentioned a few features they would like: **a robust installation wizard, consistently detailed error messages, and consistent documentation and code lookup on the FFEA website**. For the latter, they specifically suggested concise installation and requirement instructions on a single page, if possible, with simpler menus.

FFEA visualizer

Users were next asked about their experiences with the FFEA visualizer in PyMOL (Schrödinger 2023). As before, participants were asked to describe their workflow. A few participants have mentioned **using the viewer for testing purposes during setup**, however, some mentioned it in other questions rather than this one. Most participants mentioned the **use of the visualizer to view their results**, with one adding then using it to produce snapshots trivially and also movies which is a harder process that requires a third-party plugin. One participant specifically mentioned that **they do not use the visualizer for analysis and that they do not think many people do**. Another said they do not use it for analysis but that it *directs* the analysis, such as visualizing a trajectory to give insight into where analysis needs to be performed, such as in locations with maximal or minimal fluctuations. One participant said their main use for the visualizer was to construct molecules, and that they had a lot of experience with positioning and constructing rod-blob models, though sometimes they have had to use UCSF Chimera to calculate positions and then put them into FFEA.

The next question asked how much time it generally takes for the FFEA visualizer to load the data of the participant's workflow(s). In general, **answers ranged from 30 seconds to 15 minutes**, with some pointing out this depends on the length of the trajectory and the number of checkpoints, and that less high-end machines can take longer. One participant, who answered "5-10 minutes", said that it could be made a lot faster with more efficient Python code but that it is not the main bottleneck in the process "as long as you have systems that don't have 100-1000 blobs". Two users gave some more details to give ideas of timings, with one saying that for something "average" that has it set to something like 77 checkpoints that it would open

in about 30s "if that". Another said that their system of three interacting blobs with about 100 steps takes under one minute, and the same system with about 1e5 steps takes under 15 minutes to render on a PC with four cores and 16GB RAM.

Participants were next asked how well-suited they find the FFEA visualizer for their workflows and how well it presents their results. This question provoked some mixed responses. Three participants had an attitude of it being "fine". One of these participants said that they found **it presents the results satisfactorily and gives good options for presentation-quality images**, and that **the workflow for them is simple** but "could be better". Another said that they found the visualizer displays trajectories the same as any other trajectory type in PyMOL, but **they are never entirely sure what the visualizer has done to show the produced results** because of FFEA's custom units. They explained that if you check the trajectory data against the input file data, they are off by some rotation or translation factors. One participant had a more positive answer that considered the visualizer to be good, as **they found moving the structure around easy and they also praised the feature that lets users remove layers of the model**, which they used to look closer at their blobs and rods separately. They did however express **difficulty in identifying the element and node indices**, and they would like an easier way to do this. Finally, one participant said they did not consider the visualizer to be well-suited. They said **they only use it for checking things they have had to manually manage**, rather than the visualizer itself being used to set up a system. They said it is suitable for watching a simulation, but it is not useful for the process of creating and setting up a system, clarifying that "it's fine for the latter end of the job but not for the start". They also expressed annoyance that they cannot change the colours of the rods when they can do so for blobs and that you cannot use the visualizer to compare two trajectories as PyMOL loads things as scenes. **They would like to be able to open two trajectories at the same time** and control them independently for comparison. They did still note appreciation for the presentation, as it allowed for seeing wireframes, node labels, and the simulation box.

Building off the last question, participants were asked how effective they found the outputs of the visualizer for communicating structural biology to the public. All participants gave a positive answer, even if they had some criticisms. **Many praised FFEA's visual style**:

People love the stylised visuals the FFEA viewer produces. I think maybe you could

> *lean into it and put colourful geometric backgrounds on it or something.*

Participants said that they found that the use of a 3D shape instead of representations such as ribbons made the FFEA visualizer **effective at explaining molecules**, and if a user still wanted to use the more common representations used by structural biologists, they could overlay a translucent 3D mesh to show the shape.

> *As someone who didn't study biology at university, 3D structures are far more intuitive to me than a sequence schematic.*

One participant also praised the images and movies the visualizer can produce for being intuitive communication, the movies especially as that allows for observation of structures from multiple angles and zooms in a continuous motion without filling a presentation slideshow with large numbers of static images. They also said it allows for easy demonstration of molecule flexibility and interactions. **Criticisms were made however towards the rods in the visualizer**. The participant that discussed making molecule shapes translucent over more complicated representations expressed annoyance that that cannot be done for rods. They also said they felt the rods were a "less convincing" representation of a molecule as it "just looks like a stick". Another participant suggested making the material axes of rods more translucent and to add an arrowhead to the end to make it clear they are for reference and not part of the structure.

The next questions asked once again what part of the participants' workflows they found to be easy, satisfying, difficult, and dissatisfying respectively. Most participants when asked about ease gave a very short positive answer saying "[m]ost things are easy to do", "simple tasks [are] easy enough", and "[l]oading everything in is relatively easy". One participant gave specific mention to the helpfulness of being able to turn off certain layers of the model, as it lets them do things like check node indices of one component more easily.

In terms of satisfaction, two participants once again pointed out the ability to hide and unhide objects, as it makes it easy to make videos and identify node indices. Another participant said they found getting the first visualization of a simulation that has been running a long time to be very satisfying (if it works). They said **producing movies of simulations was very satisfying** too as they felt it allowed them to effectively demonstrate their work and productivity. One participant, however, simply said "[t]here isn't really any part of the FFEA viewer is [*sic*] satisfying".

For difficulties, one participant said that **setting up a system in FFEA is the hardest thing as you cannot use the visualizer to do it**.

> *The visualiser is fine, it just lacks features. Setting and getting things based on node/tetrahedron indices is by far the hardest thing.*

They went on to explain that PyMOL cannot manipulate tetrahedral meshes programmatically as they can for atomistic models, so that means a user has to load atoms onto nodes or elements, but indexing them is difficult and there is a hard limit on the amount you can have, leading to problems parametrising large systems. Another participant also said **they found identifying element and node indices hard**, as if both numbers are on the screen at once they are hard to distinguish, so finding node indices and a *corresponding* element index (or vice versa) requires workarounds such as reloading structures with various settings to identify these correspondences. Another participant said that **users also must take care to ensure what appears on screen is representative of what the runner actually calculated**. When discussing difficulties in the workload for FFEAtools and the FFEA runner in an earlier question, one participant mentioned that **trying to understand how the centroids, rotations, and scales of objects are affected by making edits is "tediously slow"** because PyMOL has to reload trajectories after every adjustment. One participant, however, said they found nothing particularly difficult in the visualizer.

For factors that cause dissatisfaction, one participant said that **they considered PyMOL's UI to be unergonomic**. Users are unable to open FFEA files directly as they need to open them in the plugin, and you cannot supply them as command-line arguments. They said **the inbuilt video encoder is very outdated** so rendering high-quality videos requires exporting frames as PNGs and using FFmpeg (Tomar 2006). They also said a lot of controls are small and the buttons poorly labelled. They also found the syntax for the console quite bad, strangely not Python, and **they criticised that official documentation for PyMOL is hidden behind a paywall**. They also said they found the floating window system outdated. They went on to explain that these are problems with PyMOL and not FFEA, and they are unsure how to realistically change things:

> *A brief survey*[1] *of MD visualisation software finds more egregious UX problems in paraview [*sic*], plus primitive tools for loading all-atom structures. VMD apparently*

[1]For clarity, the participant meant survey as a verb rather than in reference to a research survey.

> *doesn't have support for the kinds of things the FFEA viewer needs to do in its plugin system. Other tools, like UCSF Chimera, aren't really made for viewing trajectories, which makes their usefulness limited.*

Another participant expressed **dissatisfaction with the visualizer editor**, and how they usually must edit things such as the structure or van der Waals properties manually. Another participant said they were annoyed by being unable to change every colour associated with FFEA objects inside the viewing window, and that they would have to edit the plugin itself if that is needed. They also said they think each option in the FFEA viewer window should have tooltips saying what they do instead of making it so users have to look it up in the documentation, making things more difficult for new users. They briefly mentioned also finding PyMOL's look to be outdated.

When participants were asked if they had created any workarounds for the tasks they find difficult or unsatisfying, only one said they did, and that was the aforementioned manual editing of input files for things that cannot be done in the visualizer editor. They expressed that it would not be an easy thing to do if the user was not involved in FFEA development at any stage.

Participants were then asked to recount any bugs they had come across in the FFEA visualizer while performing their workflows. One participant said that **random crashes are common in PyMOL**, and they think it is probably a threading issue. They are also unsure if there is a solution as it "might just be a PyMOL problem". Another participant said that **when installing it, it can get the incorrect version of Python** and cause clashes when Python looks for imports. Another participant said there were three bugs they encountered. The first is the import errors just mentioned, and that they solved this by using appropriate environments. They also said these errors may have been solved by another user, but these edits have not been tested yet. They also said that **if the checkbox for springs is enabled in the FFEA viewer, but no springs are present, PyMOL will crash**. This occurs despite there being a check for the number of springs in the source code for the plugin. They solved this by setting springs to off by default so users will only enable it if they know their simulation contains springs. Finally, **the FFEA springs**, which PyMOL represents as "strings", **will sometimes not display at all** even if assured by the simulation all are present. This is an unresolved bug.

Participants were then asked if they have come across any accessibility issues with the FFEA

visualizer. Two participants said they did not have any. One participant said that they only had some in the beginning because they had the wrong version of PyMOL, so the FFEA plugin did not work. Another said they saw **issues with the colours**. The colour of rods is not adjustable and is currently set to green for the rod and blue for tangents. They also said that when viewing van der Waals elements, the colours are also not adjustable. While it did not cause them accessibility problems, they said it **may affect colour blind users**. They also said the FFEA team is aware of this issue. Finally, one participant said the following:

> *I don't think the FFEA viewer is very accessible. Tiny buttons, lack of labelling, lack of visual hierarchy, no contrasting colours, limited keyboard controls, etc. Again, not the FFEA team's fault, it's a PyMOL problem.*

This participant said that these problems do not personally impact their ability to use the FFEA visualizer, and that the team is aware of the problems, but that they do not have any way to take these issues upstream to PyMOL.

Participants were then asked about their experience using the visualizer with multiple people, and how well the visualizer accommodates collaborative work. One participant simply said they found it works as well as any other viewer does. Another said they found it works quite well as with it just being a PyMOL plugin; it has all of PyMOL's movie features, so it is **easy to send movies and snapshots instead of full files**. "[I]t makes making pretty pictures for grant proposals a lot easier". When asked about collaborative work using FFEAtools and the FFEA runner, one user mentioned that with the visualizer **you can share PyMOL scenes and environments with other people as long as it is contained**.

The final question in this part asked if there are any features participants can think of that if added to the FFEA visualizer would reduce the difficulty or dissatisfaction in performing their workflows. Three participants mentioned **adding interactivity to the interface**, all of which included a desire for **a graphical way of building complex structures of multi-object systems**, with such features as the ability to drag-and-drop objects, create surface-surface bonds, being able to select a node/element/face and view its properties (such as position, indices, neighbouring nodes and elements etc.), and connect rods and blobs. One participant mentioned regarding nodes that currently users have to view a node ID in PyMOL and then index it within a Python script using FFEAtools to gain information. One participant also **wanted the interface to be useful for meshing and mesh analysis**, such as being able

to rotate around the centre of mass and editing the mesh tetrahedra. They said that if this interface could have pointers to every file it read in, users could edit them on the fly and update the files as they work on them.

Ranked questions

At the end of the user section of the survey, each for FFEAtools, the FFEA runner, and the FFEA visualizer, participants were asked to agree or disagree with a series of statements that said these components of FFEA were easy to use, easy to learn, satisfying to use, and that they run at an appropriate speed. Participants were allowed to say if they extremely disagree, highly disagree, disagree, agree, highly agree, or extremely agree. They were then asked if they would explain their answer. A neutral option was avoided as participants were allowed to simply not answer the question if they had no opinion on the statement. Each statement was worded positively so that for each an agreement was consistently a positive result, so that average answers could be calculated. This was done by giving each answer from extremely disagree to extremely agree a value from 1 to 6 respectively (which will be referred to as "user score"), then calculating mean averages as usual. As a reminder, due to the small sample size from this survey, these scores should not be considered indicative of how the average *potential* user would rate FFEA.

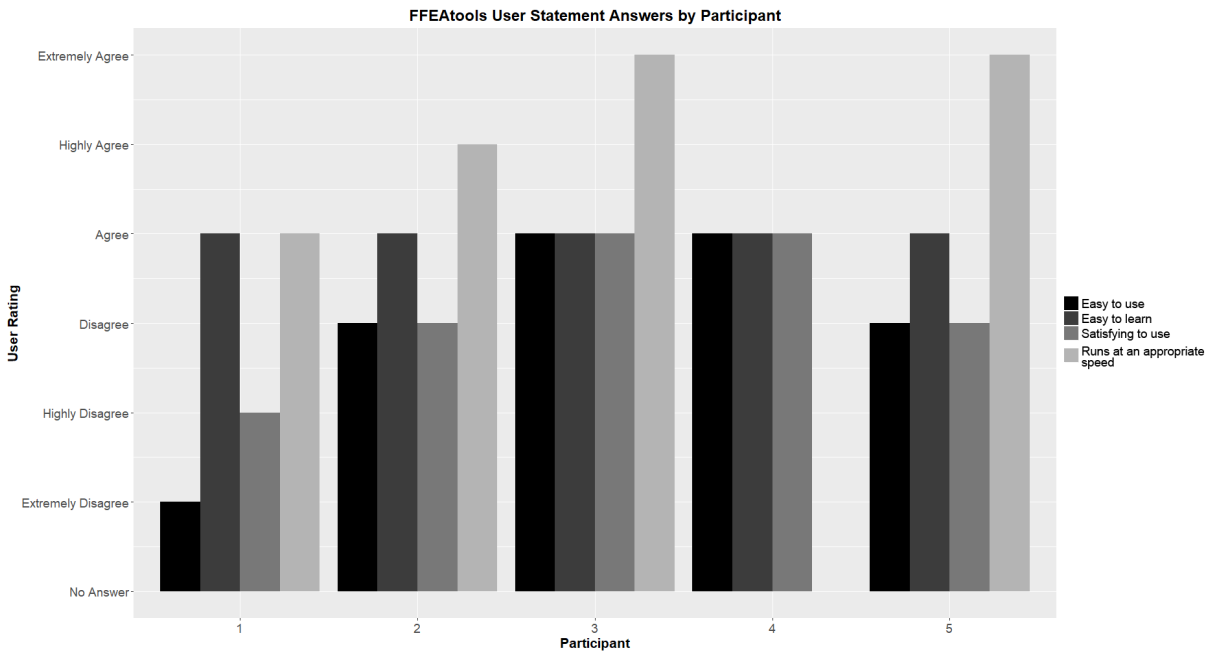

Figure 4.1: User statement answers for FFEAtools.

Figure 4.1 shows all the given closed answers to the statements towards FFEAtools. When asked if FFEAtools is easy to use, answers ranged from extremely disagree to agree, with an average user score of 3/6 which gives an average answer of "disagree". A participant that disagreed said they found there to be **too many modules for the API**, and all of these modules are too independent which made them tricky to use, but said **the actual list of available functionalities is fine**. Another participant, who agreed, again said some parts of it are not very easy to use, and that **compilation can have issues** due to the problems with the version of Python used, and that is what held back a more positive answer. Finally, another participant that agreed said that **the process for FFEAtools is well-documented but the number of files it produces makes it difficult to keep track of what a user has completed**.

When asked if FFEAtools is easy to learn, every participant answered "agree" (a user score of 4/6). One participant said that **they found the documentation to be good**, and the example files provided really helped. Another said they found that it is clear what each tool does, and even if the API is dense, it is clear what each object is. The latter participant described FFEAtools as easy to learn but difficult to use.

When participants were asked if FFEAtools is satisfying to use, answers ranged from highly disagree to agree, with an average user score of 3.2/6 which is a "disagree". One participant, who disagreed, described FFEAtools as feeling "hacky", saying that **you cannot load an entire system at once and must do it all individually and manually**. Another user, who agreed, said that **they found the rods to be very intuitive** and that **they can make a mesh from a PDB very quickly**, but other problems cause dissatisfaction, mainly because of compilation problems.

Finally, when asked if FFEAtools runs at an appropriate speed, answers ranged from agree to extremely agree, with an average user score of 5.25/6, a "highly agree". Further comments were very sparse, with two participants basically saying they have no problems with FFEAtools's speed.

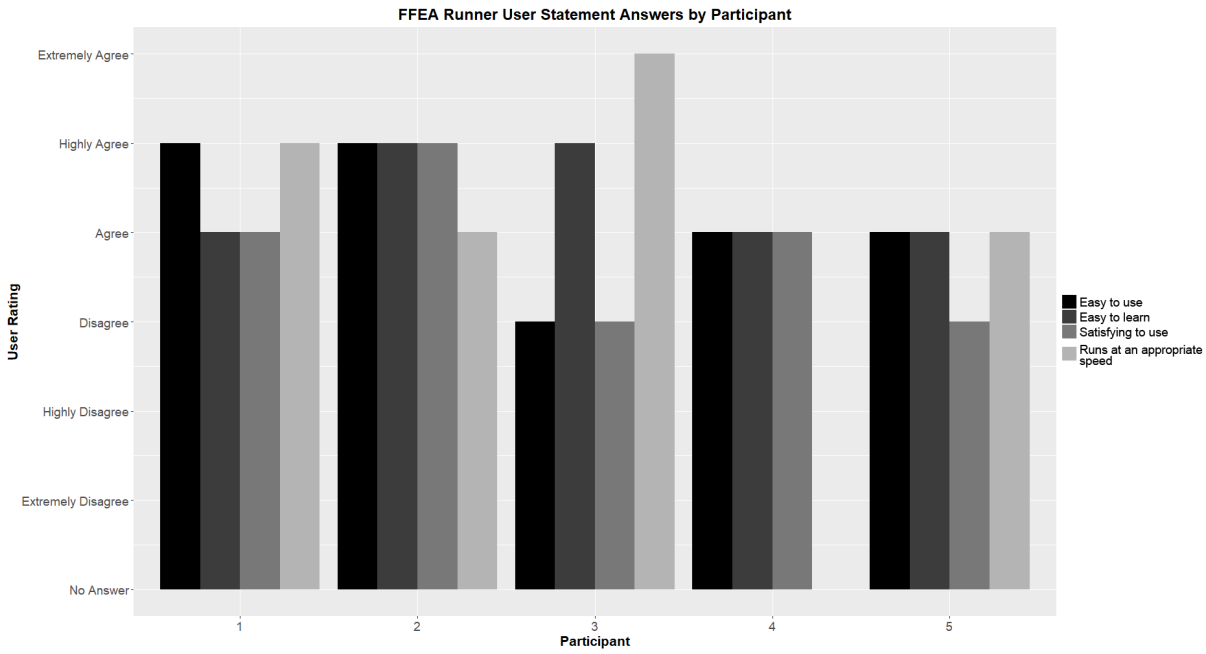

Figure 4.2: User statement answers for the FFEA runner.

The next set of statements were for the FFEA runner, and the closed answers from each participant can be seen in Figure 4.2. When asked if the FFEA runner is easy to use, the answers range from disagree to highly agree, with an average user score of 4.2/6 which is an "agree". A participant who highly agreed said that **once the setup is completed, then it just runs immediately**, and the main thing to worry about is the time step, which they said users should care about anyway. One participant, who agreed, said that **the documentation for it is good**, but that they did not initially know what lengths and time steps were appropriate for the simulations. One user who disagreed, however, said that **the lack of a GUI was a problem** and that they thought that if a user is outside the main FFEA group who are in constant discussion over the software, then it is not always very clear, **not very well-documented**, so it is hard to know what every variable does, what the associated text files are, and what the contents of these files should look like. They said **the runner is a complex structure for a new user**, so they do not consider it very easy to use.

Next, when participants were asked if the FFEA runner is easy to learn, responses ranged from agree to highly agree, with an average user score of 4.4/6, which is an "agree". One participant highly agreed because they also highly agreed that it is easy to use. They said the **files are human-readable** with column headings that are easy to understand. Another highly-agreeing participant said **they found the tutorial good and well-documented**. They said you

can learn the functionality easy, even if they do not find that functionality easy to use (they disagreed on the FFEA runner being easy to use). They said that they did not think, however, that the steps are very "streamlined and[...] succinct", that **there are a lot of different steps to go through but getting through them *is* simple**. They said **some parts are not as well documented** which holds it back a bit. They surmised saying that basic features that appear in the tutorial are easy to learn, but if a user wants to do something more conceptual that is more difficult due to the less documentation for less commonly used features. Finally, one agreeing participant simply said they found the documentation good.

When asked if the FFEA runner is satisfying to use, answers ranged from disagree to highly agree, with an average user score of 3.8/6, an "agree". One participant who highly agreed said **they found it satisfying in its simplicity**, and another participant who agreed said they found it good that **the runner shows it running through the various steps and that any errors are clearly displayed**. One participant who disagreed however said the features they like are very informative and tell you what the runner is doing during a simulation, but that it should print out a log file as default rather than printing to the terminal. They said that the actual information it prints out is useful for troubleshooting, but **they would not call it a satisfying and enjoyable process to manually edit text files**.

Finally, when asked if the FFEA runner runs at an appropriate speed, answers ranged from agree to extremely agree, with an average user score of 4.75/6, a "highly agree". A participant that extremely agreed said that **compared to MD it is very quick and not as computationally intensive**, but they said they are very used to MD so FFEA in comparison may just feel very fast to them. A participant that agreed said that the current pace is good, but it has a much better potential:

> *[I]f it was refactored such that it was built with parallelization in mind, and it could run* massive *systems incredibly quickly... but currently it doesn't do that.*

Another agreeing participant said that **they found the code to scale acceptably** for eight cores, though adding lots of blobs can cause problems. They also said that rods by themselves scale the best.

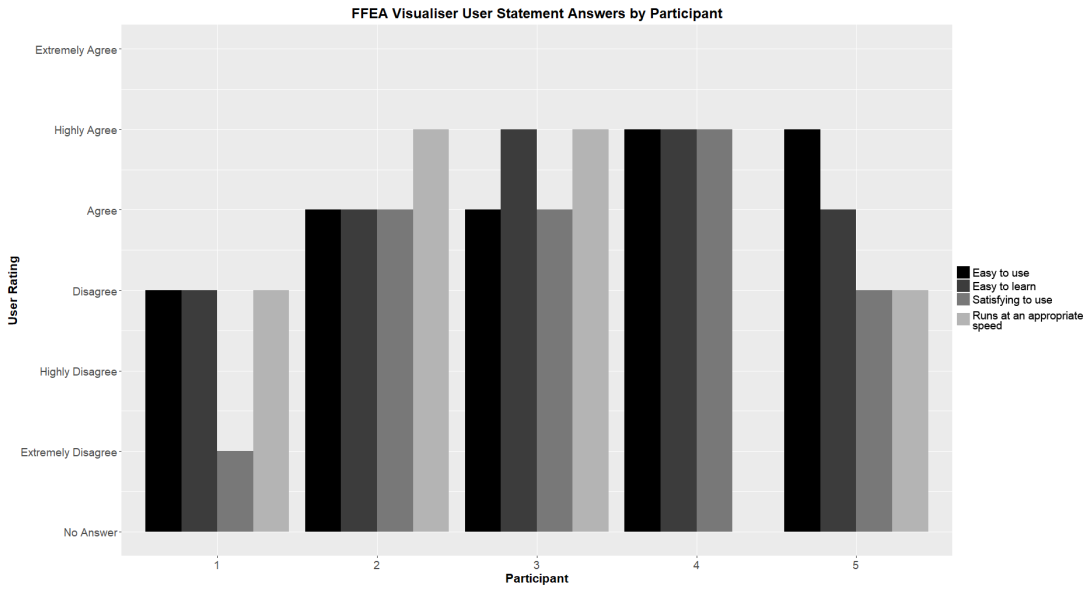

Figure 4.3: User statement answers for the FFEA visualizer.

The final group of statements were for the FFEA visualizer, the answers for which can be seen in Figure 4.3. When asked if the FFEA visualizer is easy to use, answers ranged from disagree to highly agree, with an average user score of 4.2/6, an "agree". One participant that agreed said that **it is difficult to install but is otherwise fine**, and another agreeing participant also said similarly they had problems with compilation and getting it to run but opening a file and looking at a simulation is easy and understandable. One participant who highly agreed said that **it was clear to them to see the model and simulation box** and that it is easy to move the view around.

Next participants were asked if the FFEA visualizer is easy to learn, prompting responses that ranged from disagree to highly agree, with an average user score of 4.2/6, an "agree". One participant who highly agreed said that **the documentation is good**, however, they already knew PyMOL well before using the FFEA visualizer. Another highly agreeing participant said that they had to perform a fair amount of trial-and-error but found it generally ok to pick up. One participant that agreed said that **the front screen is obvious and easy to see what you need to do, but the editor is not as easy**, and this holds it back.

When participants were asked if the FFEA visualizer is satisfying to use, answers ranged from extremely disagree to highly agree, with an average user score of 3.4/6, a "disagree". An agreeing participant said that **when the visualizer loads a system that PyMOL makes it look good** and "pretty", and a highly agreeing participant said that it was satisfying to

them because of **the ease with which a user can move the view and remove layers of the model**. Another agreeing participant said that they liked seeing their simulations but **there is lots of functionality the visualizer does not have that would make it more satisfying**, especially interactivity.

Finally, participants were asked if the FFEA visualizer runs at an appropriate speed, which got responses ranging from disagree to highly agree, with an average user score of 4/6, an "agree". A highly agreeing participant said that **the visualizer runs slow for massive systems**, but that is typical of PyMOL and not FFEA's fault. Similarly, another highly agreeing participant said that it is slow on certain machines and for big simulations, but they said that tools like VMD are similar in that case. A participant that disagreed also said that the visualizer is slow for large systems and that they found this to be annoying, but not too long as to be a major issue to them.

Figure 4.4 shows the average answers for these statements by each participant and the average for all participants. User ratings were fairly consistent with average answers hovering around "agree", with participant 1 being the notable exception with a bigger range of user scores of each part of the FFEA software suite. For the overall averages, all parts of the software suite had an average answer of "agree" to the positively worded statements. FFEAtools had the lowest average user score of 3.8/6, the FFEA visualizer's average user score was 4/6, and finally, the FFEA runner had the highest average user score of 4.25/6. This gave the whole software suite an average user score of 4.02/6.

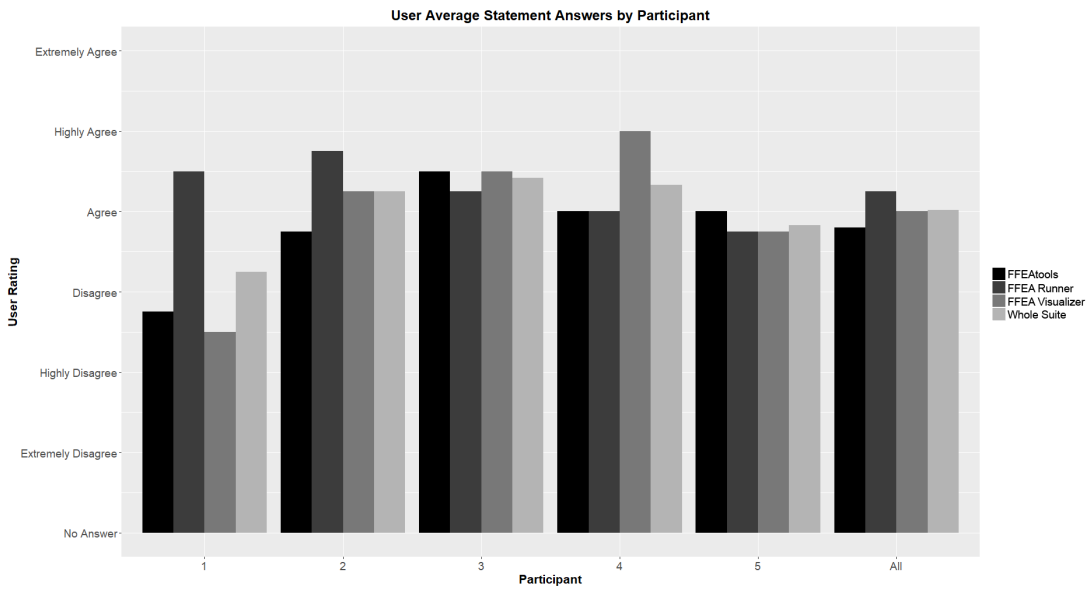

Figure 4.4: Average user statement answers.

4.2.2 Developer responses

The two participants that declared themselves to have experience with full-time development of FFEA in the 12 months prior to the survey were then asked about their experiences with this work. The first question asked which of the three components of the FFEA software suite they had experience with developing. Both participants said they had experience with all three.

Next, participants were asked if they could give a brief description of the general tasks they had performed when developing FFEA. The first participant said they had developed a new algorithm, performed refactoring, developed new tests for the source code, written documentation, and had performed debugging. The other had said that they had primarily edited documentation, changed some of the imports in FFEAtools, and adjusted the indentations in the source code for the FFEA visualizer plugin for PyMOL. They also said they had attempted to debug small parts of the rod source code in the FFEA runner but unsuccessfully.

Participants were then asked how much time or effort it took to develop the areas of FFEA they worked on. The first participant said it took them four years to do everything they had done. The second said that documentation and website changes were done gradually over a few months with fairly low effort. They also said that adjustments to the FFEAtools and the plugin were low effort.

Next, participants were asked what parts of developing FFEA they found easy to do. One participant said that while **writing numerical simulations is not easy**, **the rod code is easy to use**, but they did note a potential bias having worked on it. They also said **they believe FFEA's finite element code and solvers to be the nicest part of the codebase**, but they had not made any changes to either. The other participant said **they found the documentation easy to edit** as it is "fairly sensibly laid out".

The next question asked what parts of development participants generally found satisfying to do. One participant said that they found **development of the rod source code to be satisfying**. The other said that **they found it satisfying to resolve the import and indentation errors** as they were "extremely annoying bugs that popped up quite often" and only required little effort to remove.

The next question then asked what parts of developing FFEA they generally found difficult to do. One participant said they found it **difficult to develop a holistic understanding of**

how the source code works:

> *I think there's lots of detritus strewn around world.cpp and other files where it doesn't belong. I think the documentation only tells you what lines do and not what entire files and objects do. Some of the code, like collision handling, is simultaneously very abstract but also has a huge amount of unnecessary duplication.*

They also mentioned a few additional specifics, saying that **they found adding I/O functions to be time-consuming** as they are all hardcoded, and that **initialisation is difficult to maintain** as it is "a long-winded process spread through many functions in many files". They also said **they thought the collision detection code "looks incredibly scary"**, though they had not used it themselves. The other participant said there was nothing they found particularly difficult during development but **attempting to debug the rod code was difficult** for them due to its size and complexity, though they noted this may be due to their inexperience with debugging.

Participants were then asked what parts of developing FFEA they found unsatisfying. One participant had a large and detailed response to this question:

> *Data is often nested deeply inside complicated data structures with little reason. Some objects have member variables and functions which are badly named, misleading, and\or don't do the thing you assume they do. Many objects have very few public methods and variables, too few to actually manipulate the data inside them. Many functions that could be pure are instead member functions that operate on data from specific data structures, which hurts reusability within the codebase. The parser desperately needs a rewrite due to all the hardcoded XML tags. The python is generally poorer than the C++, in part because it's written in a C++-like style, and the structure is very odd, with all of the python modules dumped on the python PATH, and tools dumped in a separate folder. The way the python modules are dumped means they tend towards being extremely long files, as developers don't have an easy way to split them into separate files. The way that the tools are split up means they can't re-use code from one another, which means they're full of duplication. I attempted to fix this back when I started[...] but didn't finish what I started.*

The other participant said that they do not like that **there is many defunct branches in the**

FFEA repository, and that they hope the development team can soon change the repository to an alpha-beta-gamma structure[2]. **They also found the layout of the C++ source code to be confusing**, and that the level of detail and the usefulness of comments is inconsistent between files, but they noted that they found the rod source code to be good in their view. Finally, they said that the use of namespaces in the C++ code can make finding the origin of certain functions, routines, and objects confusing to people less experienced with the language, even if they understand why namespaces are used.

Participants were next asked what tools they had used during developing FFEA and how they helped. The first participant said that GDB (The GDB Developers 2023) and Jenkins (jenkinsci-docs 2023) were probably the most useful and that Jenkins was particularly of use to them for finding "elusive" physics bugs. For rod parametrisation, they used MDAnalysis (Gowers et al. 2016), SCWRL4 (Krivov et al. 2009), and QUARK (Xu and Zhang 2012), and they also briefly used gprof (Fenlason and Stallman 1998) and Valgrind (Valgrind Developers 2022a) to look at performance. For writing source code, they used a text editor and the Linux console. The second participant said they mainly used the Visual Studio Code (Microsoft 2023) text editor and a terminal, but they are considering upgrading to a complete IDE with a debugger as they believe it would be useful to do so.

The next question asked participants what they saw as FFEA's current biggest issue that requires attention. The first participant stated three issues; **the lack of usability for new users**, that **it is difficult to create complex systems through the writing of FFEA scripts**, and that **they considered several parts of the source code to be "almost unmaintainable"** and that they should be rewritten, pointing in particular to collision detection and input file parsing. The other participant said that **the complicated structure of FFEAtools is something that needs resolving** as it is currently built using Python 2.7.X and that if the FFEA team wants the code to be "relevant" then an upgrade is needed as soon as possible.

The final question in this part asked if there are any changes to FFEA or its development

[2]While the participant did not explain this in their survey answers, this refers to a project branch structure discussed in FFEA source code meetings. Developers when working on a new feature or bug-fixing would create a branch off of the latest version on alpha to make changes in, and when finished and tested this branch merges back into alpha. When the software suite is ready for a release, the alpha is merged into beta, and beta is given to people to test. If needed, a gamma branch is used for last-minute patching. Once satisfied with this testing, the developers then merge this into master and tagged for public release when developers are certain it is ready. This keeps the master branch a clean list of tagged stable merges. To surmise, alpha is for developer work, beta is for user testing, gamma is for patching, and master is for stable releases.

cycle the participant thinks would improve the ease of development. One participant gave four suggestions. The first would be a project to **restructure the Python modules** and to incorporate its functionality into a cleaner Python API instead of multiple standalone tools. Next, they suggested a project to **identify which areas of the FFEA runner source code are the least maintainable and cleanly rewrite them**. They mentioned in particular the source code for I/O, collisions, initialisation, and the simulation box as parts to rewrite. They also suggested **fixing the data structures**, especially the removal of second-order elements. Finally, they suggested **creating a custom viewer that can load FFEA and PDB files with a graphical way of creating FFEA systems**. The second participant also gave four suggestions, starting with the aforementioned desire for adopting an alpha-beta-gamma branch structure for the repository. They also suggested **creating a set of coding standards** for developers to avoid inconsistencies such as indentation errors in the Python code. Next, they suggested **a test-driven approach to development**, where tests are devised first before any source code is written. Finally, they suggested **hiring a dedicated research software engineer**, believing it would vastly improve FFEA "as the majority of its code has been written by stressed PhD students".

Ranked questions

At the end of the developer section of the survey, each for FFEAtools, the FFEA runner, and the FFEA visualizer, participants were asked to agree or disagree with a series of statements that said that they were easy to develop for, that their source code was easy to understand, and that they enjoy developing for it. As before, participants were allowed to choose from a range of responses from extremely agree to extremely disagree, were asked afterwards to explain their answer, and that the statements were worded positively so that agreeing was consistently a positive result for calculating mean average responses.

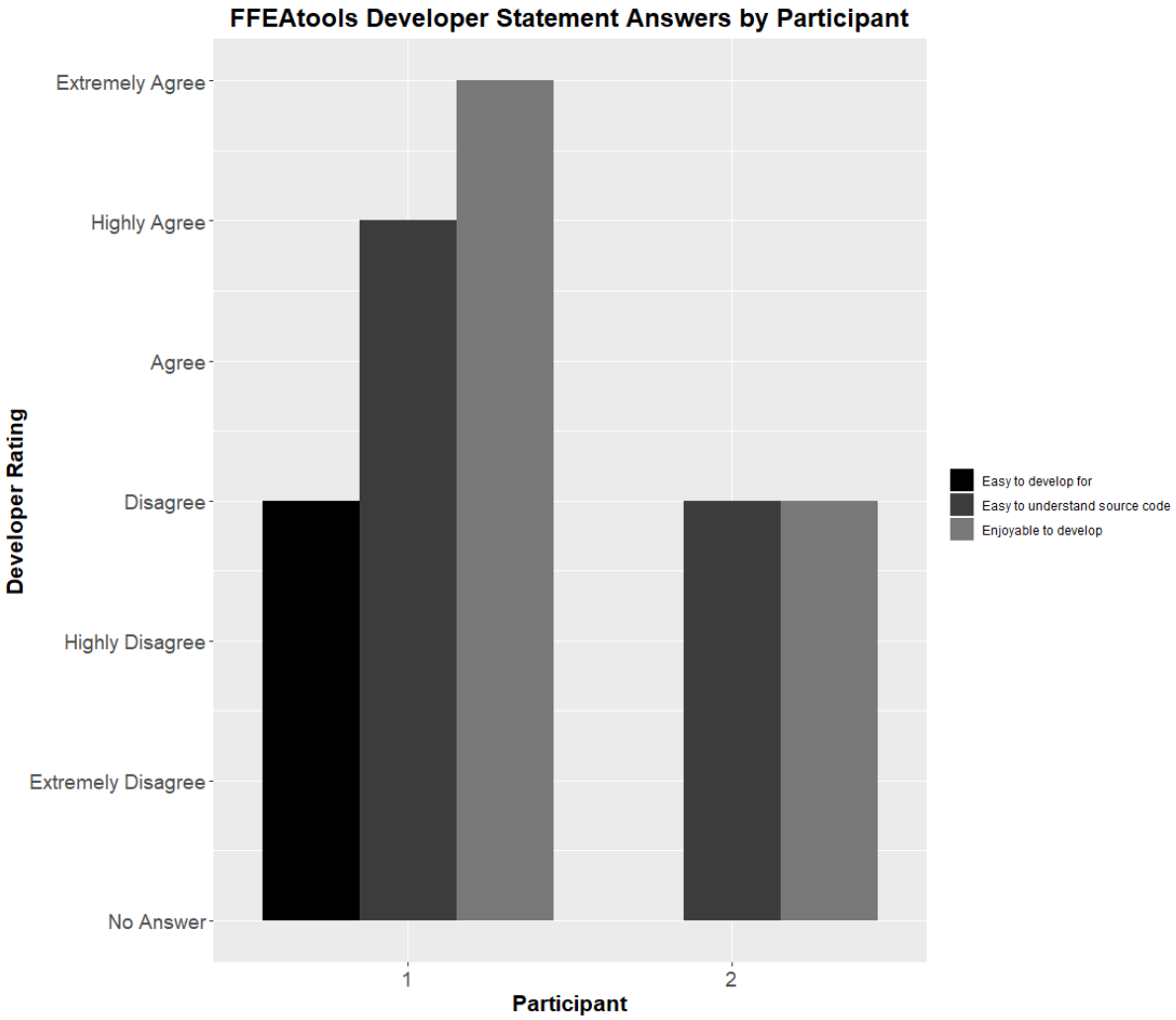

Figure 4.5: Developer statement answers for FFEAtools.

Figure 4.5 shows the closed answers given to the statements on FFEAtools. When asked if FFEAtools is easy to develop for, only one participant answered with a "disagree", giving no further comment. When asked if FFEAtools's source code is easy to understand, the first participant answered highly agree. The second disagreed, saying that the source code itself is fine but the **complicated structure and interdependencies** due to files importing each other make it difficult to understand how a change may affect the rest of the code. Finally, when asked if FFEAtools is enjoyable to develop, one participant extremely agreed, while the other disagreed, each with no further comment.

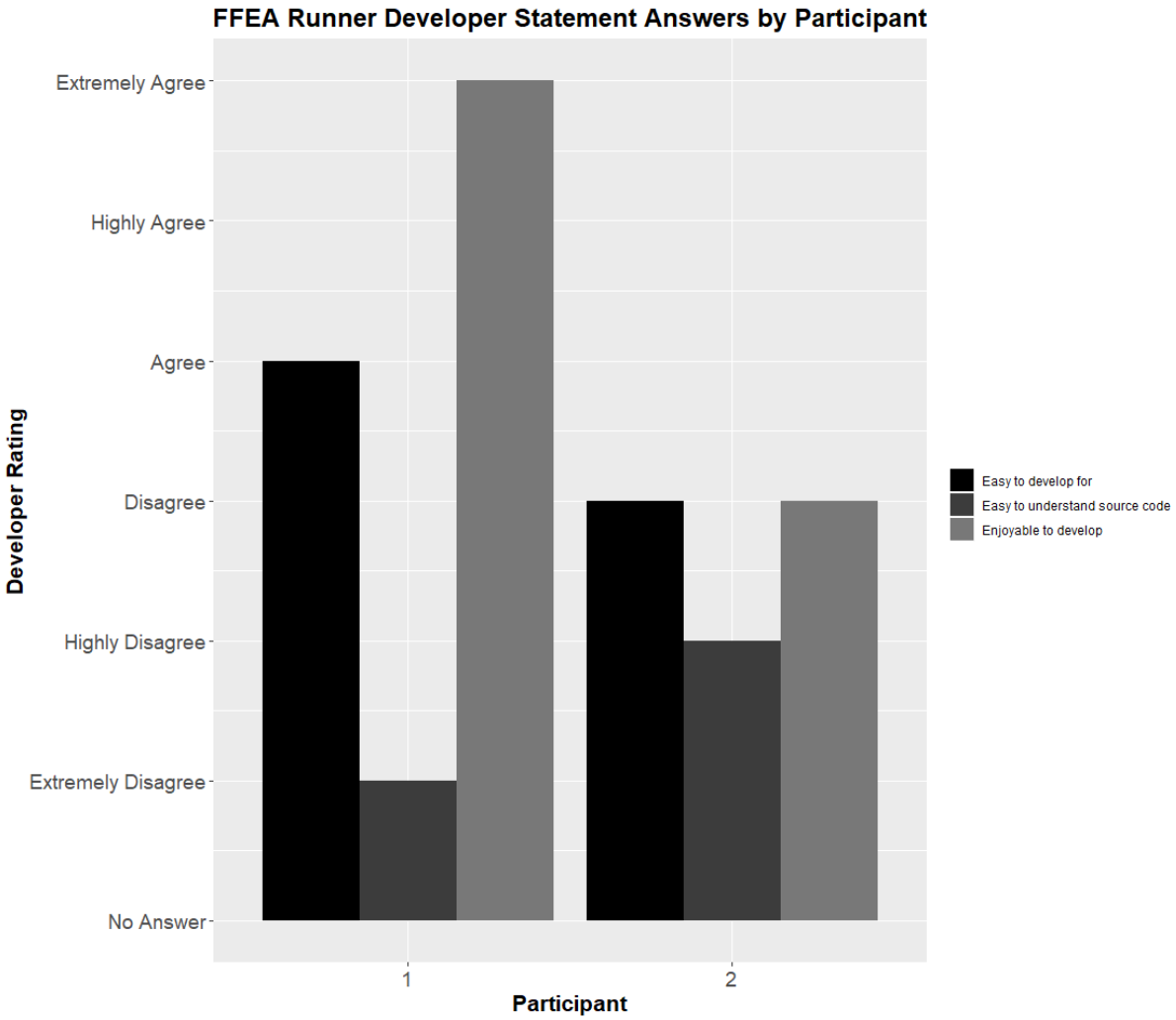

Figure 4.6: Developer statement answers for the FFEA runner.

The next set of statements were on the FFEA runner, the results of which can be seen in Figure 4.6. When asked if the runner is easy to develop for, one participant agreed and the other disagreed, with no further comment. When asked if the FFEA runner source code is easy to understand, one participant extremely disagreed. The other highly disagreed, saying that **files are often large and complex**, with **comments being inconsistent** between different parts of the code. They also said that there is **no extensive developer documentation** on the website with the exception of the source code for rods. Finally, when asked if they enjoyed developing the FFEA runner, one participant extremely agreed, while the other disagreed, each with no further comment.

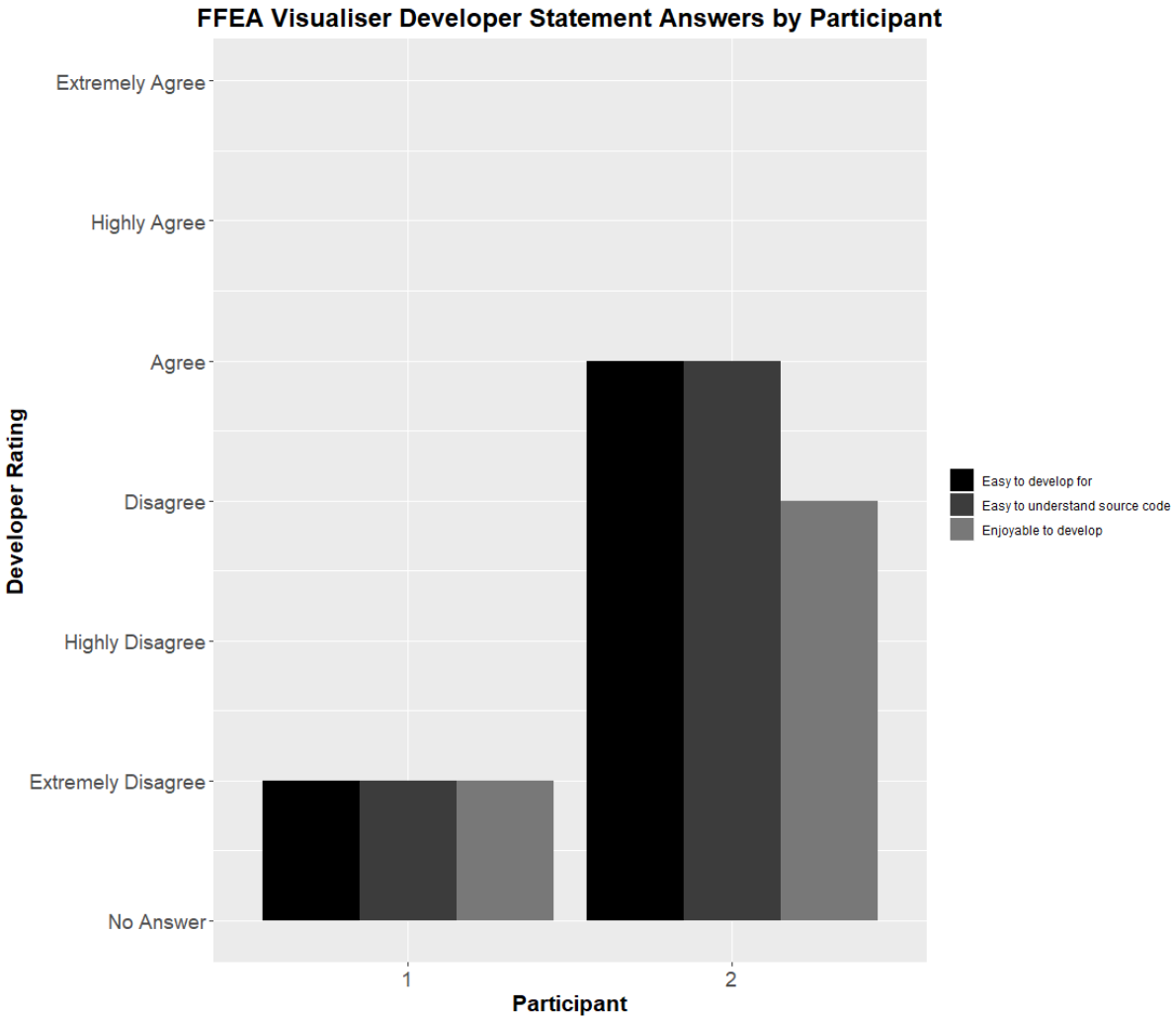

Figure 4.7: Developer statement answers for the FFEA visualizer.

The final set of statements were asking about the FFEA visualizer, and the results can be seen in Figure 4.7. When asked if the visualizer is easy to develop for, one participant extremely disagreed, while the other agreed, each with no further comment. When asked if the source code is easy to understand, one participant extremely disagreed, while the other agreed, the latter saying that there are far **fewer files involved**. Finally, when asked if they enjoyed developing the FFEA visualizer, one participant extremely disagreed, and the other disagreed, with no further comments.

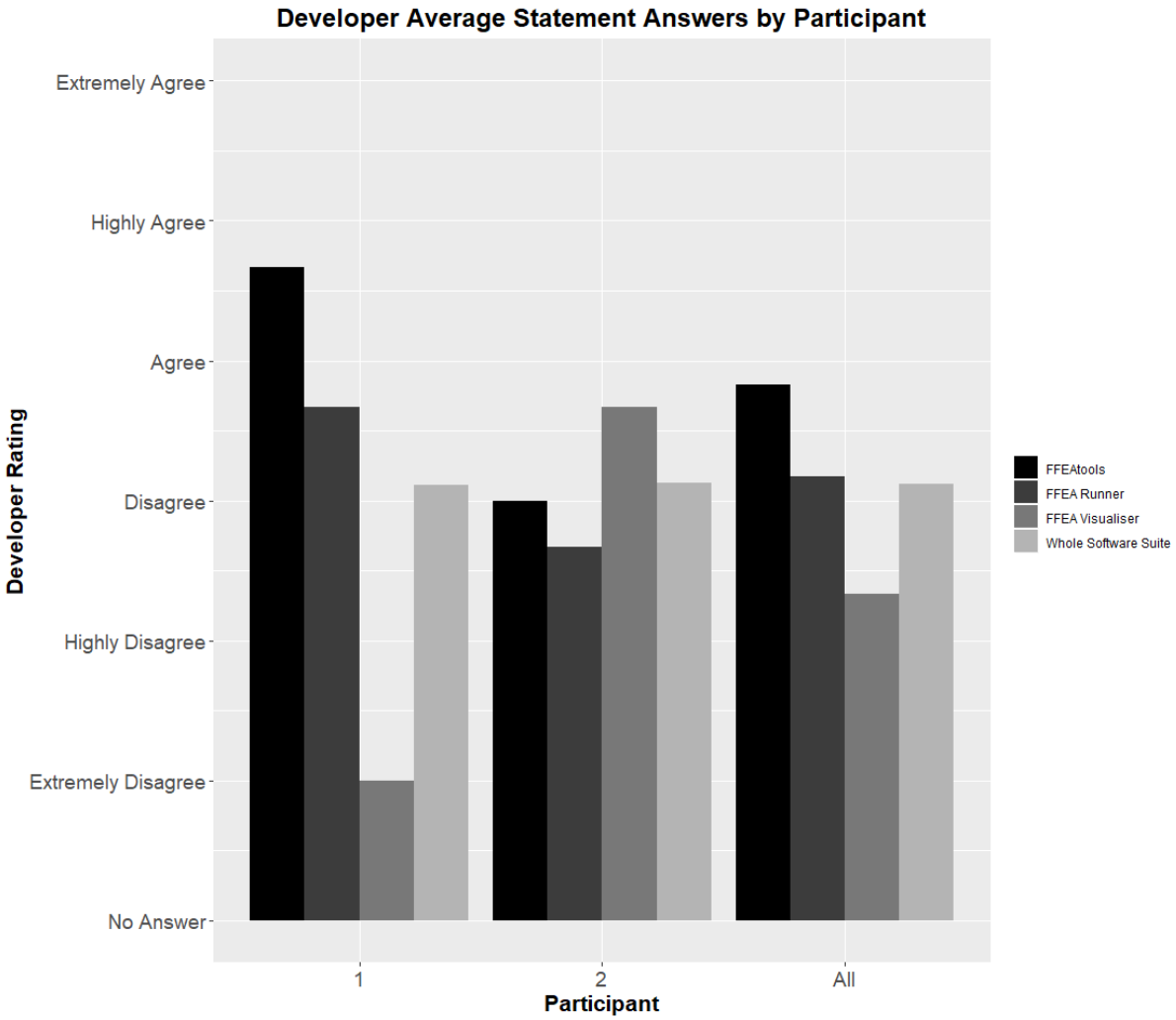

Figure 4.8: Average developer statement answers.

Figure 4.8 shows the average answers for these statements by each developer and the average for both developers. The FFEA visualizer had the lowest average developer score of 2.33/6 which is a highly disagree, then next was the FFEA runner with an average developer score of 3.17/6 which is a disagree, and finally FFEAtools had the highest average developer score of 3.83/6 which is an agree. This gave the whole software suite an average developer score of 3.12/6, which amounts to a disagree towards the positively worded statements.

4.2.3 Participant final remarks

The last question of the survey asks the participant if there is anything about their experience using or developing FFEA that had not been discussed in any other question, but they feel should be stated. Three participants gave a response to this.

One participant provided the following statement:

> *My comments here have tended towards the negative because I'm listing things that I think should be improved, but my overall experience is extremely positive. Even though it has issues it's still pretty good compared to most scientific software, and a lot of fun to work on once you learn how it works. The problem is that the complexity of FFEA systems and of all the parametrisations makes setting up and analysing systems an inherently complex task. This, in turn, requires a lot of software engineering.*

This participant also said that **they think FFEA should not use Matplotlib** (Hunter 2007). They said that for simple plots it is fine but that it can be very buggy for plotting complex things, giving particular mention to logarithmic plots that have many glitches including overlapping text, incorrect aspect ratios, missing labels and tick marks etc. They also said they found the API to be confusing.

Another participant said the following:

> *It's got massive, massive potential and what's currently holding it back is that it's not being developed with[...] anything other than the next PhD project in mind[...] it's become a lot of patches on top of patches[...] it's necessary for like a global warming planet, right? If you want to stop using big computers, you need something in the middle like FFEA to do molecular biology properly, but it just needs* someone *to sit down and develop it with[...] general functionality in mind rather than just whatever project is on the go at the time.*

Finally, one participant said that **the main things they found difficult were the lack of interactivity with systems** and that they had **issues with compiling**. Otherwise, they said they considered FFEA to be "quite good".

4.3 Analysis

This section analyses the results from the survey that have been shown in the previous section. This is split into further subsections on the topic of perspective for analysis.

4.3.1 User work practices

While the participants of this survey all had different projects that they used the FFEA software suite for, they also all roughly shared a common main workflow. This is the process of using FFEAtools to create models for FFEA (either through the creation of tetrahedral mesh, rod models, or a combination of both), then using these models as input files for running simulations in the FFEA runner. Most participants then also mentioned the use of the FFEA visualizer to then view the results of these simulations.

This workflow, being as common as it is, should therefore likely be the aspects and functions of the FFEA software suite that receive the largest amount of attention by the development team, and it indeed appears that is already the major focus, with the main tutorial in FFEA's documentation being a detailed step-by-step example of this workflow. This tutorial has even commonly received praise from participants for being well-written and making FFEA easier to learn, which suggests that effort put into addressing this workflow pays off. Participants have found ease and satisfaction in such parts as running and viewing simulations, and generating meshes and rods, with some attributing that to these functions being well-documented.

Participants, however, have not found the common workflow to be without problems. While the generation of meshes has received praise for ease, the creation of meshes and FFEA systems has been commonly considered to be less easy. This has been described as a lengthy process of many trial-and-error steps, with participants finding such things as parametrisation and positioning to be difficult, especially due to the methodology of making changes through editing text fields in multiple input scripts. The FFEA runner has also received criticism for the common element inversion bug that causes simulation crashes that require users to restart them, and that this restart process can itself be annoying as input scripts will need some of their fields editing and some files will require renaming, which currently needs to be done so manually (or through a user creating their own scripts to do so as one participant did as a workaround). As for the visualizer, while most found that it presents results of simulations satisfactorily, some still pointed out a few issues. One is that it can be difficult to understand the viewed results with certainty due to FFEA's custom units. Another is that the visualizer is not well-suited for helping users with the majority of the common workflow, in that it can only be used to check parts of the creation and setup processes that are managed manually instead of through the visualizer itself, and that it is only the very end of the workflow with viewing results where

the visualizer has suitable functionality. The visualizer was also criticised for the difficulty in identifying element and node indices and how they correspond.

Many of the suggested features for FFEA address issues with the common workflow. The most common suggested feature was the addition of a graphical interactive interface for FFEA to make it easier to build and parametrise complex FFEA systems, to set up interactions in multi-object systems, and to select nodes/elements/faces to view their properties and alter their parameters. The next most-reoccurring suggested feature was the automation of certain parts of the common workflow, such as the entire initialisation process from atomistic or cryogenic data to a parametrised FFEA surface mesh, the restarting of simulations, and automatic management of FFEA files and their directories.

With how much of the discussion, critique, and suggestions around work practices are centred around this common workflow it seems justified that there is a focus of effort on the development of functionality used in this workflow, and that this focus should continue. While the small sample size of this survey hampers the validity of statistical analysis and suggestions of the average FFEA user workflows and their experiences with them, the fact that every participant discussed this workflow to some extent indicates importance.

However, there appears to have been a downside in that efforts may be *too* focused on the common workflow. Some participants have criticised that less-common functionality they may need is harder to learn and understand due to a lack of attention in the documentation. The FFEA runner reportedly does not have every variable and associated text file documented, and not every FFEA file has documentation on what the contents of that file should look like. This problem could be exacerbated considering some participants expressed that users have to know how the FFEA software works on a conceptual level in detail to perform some of these tasks, such as utilising van der Waals interactions. This could essentially mean that compared to other simulation software, FFEA demands more understanding from a general user but does not sufficiently provide ways for users to gain that understanding.

While it is likely prudent to keep investing significant effort into improving the common workflow, the less-common work practices require much-needed attention, lest they become unused due to new users being unable to figure them out. Potentially, this could lower user retention as they stop using the FFEA software suite altogether due to being unable to learn the functionality they require in a reasonable time frame. Time should be made especially for documenting

the undocumented functionality, and once more users can pick up the less-common functionality more easily they can then provide better feedback on these functions to improve them as needed.

4.3.2 User experience

While there were minor variations in user score between the components of the FFEA software suite, on average participants answered an "agree" to the positively worded statements on the ease of learning, ease of using, satisfaction of using, and the appropriateness of the running speed of the FFEA software suite. Along with some further comments by participants, it suggests out of the participants of the survey people tend towards having a positive experience with FFEA, but one that has much room for improvement. As an example, to restate what one participant said in their additional comments in Section C:

> *My comments here have tended towards the negative because I'm listing things that I think should be improved, but my overall experience is extremely positive.*

The previous section on work practices covers a lot of the current state of the user experience with the positive and negative sides of the common workflow, but there are other areas to discuss. However, one thing to note regarding the common workflow is that despite difficulties, many participants noted general satisfaction upon completion of their work and the problem solving it required and that the results produced by the FFEA runner and visualizer especially have an impact on that satisfaction, such as the visualizer's ability to produce high-quality images and movies with good communicative ability (the latter of which is discussed in the last subsection on the use of FFEA).

One thing of note to the user experience of the FFEA runner that has not been discussed is that one participant pointed out that its input files are all in a human-readable format with column headings that are easy to understand. While this was a minor point made it is an important one, as with the already difficult nature of the common workflow that requires a lot of manual editing of text files, this would be made much worse if these files were less easy to understand. Care should be taken to ensure this is a feature of the runner that continues as changes are made.

Another aspect to address is the analysis of systems and simulations in the FFEA software suite. FFEAtools received praise for its easy-to-use analysis functionality, such as trajectory

analysis and graph creation. The visualizer on the other hand was not considered as good for analysis according to some participants. One of those participants, however, noted that while they do not do analysis in the visualizer, it directs the analysis by giving insight into what areas of the system or simulation to study. This is another problem that could be solved by the inclusion of a GUI for the FFEA software suite. If developers could incorporate FFEAtools's analysis functions into a visualizer, then that alone would give users a tool that both directs *and* performs analysis.

One of the questions on the visualizer asked participants about any accessibility issues they may have come across. Participants did not state having any accessibility issues that may clash with software like the FFEA visualizer, therefore if an accessibility evaluation of FFEA is desired it would have to be done in a separate piece of research. However, there were some points gleaned from participant observations. The main one for the FFEA visualizer itself was how certain features cannot change colour, such as rods and van der Waals elements, which could potentially cause issues with colour blindness. The other issues pointed out were accessibility problems with PyMOL itself that the FFEA development team cannot change, such as the tiny controls, poorly labelled buttons, limited keyboard controls, and its own issues with colour such as the lack of contrasting colours. These downsides to PyMOL should be considered when debating if the FFEA visualizer should move away from PyMOL. It could be useful to the FFEA development team to see if there have been any reports written on the accessibility of PyMOL to help with this decision or to see if there are ways the existing FFEA visualizer could be improved to counteract problems[3]. That, and if the FFEA development team does decide to port the visualizer or create a new one for another platform, then they should also see if the new platform has any reports written on its accessibility.

Building off the last point, PyMOL had a fair few other user experience problems addressed by participants; lack of access to documentation without passing a paywall, having to reload trajectories after every adjustment, the inability to compare trajectories, and poor console syntax as examples. Participants also made many of the same complaints about PyMOL's UI outside of the question on accessibility. However, not all comments about the use of PyMOL were negative. As PyMOL is being used as a framework for the FFEA visualizer, users already familiar with it for other MD uses would be able to pick it up fairly quickly, one participant

[3]A search for such a report was conducted at the time of writing this evaluation, but unfortunately to no avail.

saying this was the case for them. PyMOL's object hiding function was a useful tool to some for getting a clearer look at specific parts of the system, although this is not a functionality unique to PyMOL and can be seen in many graphics programs, so it could feasibly be incorporated into a new visualizer.

Overall, during the surveying there were more complaints made against the use of PyMOL than praises, and they should be considered when discussing how to move forward with the visualizer. However, this cannot be considered a clear indication to move away from PyMOL, as there are other factors to consider, such as whether a different framework can realistically produce a better visualizer. One participant pointed out that they did not know how to realistically do this, as they reported that other MD visualization software has issues, such as ParaView having a poor user experience in their opinion and having less-suitable tools for loading atomistic structures, VMD not supporting some of the functionality the FFEA visualizer would need, and UCSF Chimera not being designed for viewing trajectories. While clearly there is a desire for the FFEA visualizer to be improved in some way, the development team must carefully ensure if it is moved to a new system that it allows for significant enough improvement for the effort to be worth it.

4.3.3 User and algorithm performance

When participants were asked to describe the speed of their usage of the FFEA software suite, there appeared to be a large difference between the time taken to set up a simulation and the time it takes for the software to run them. Answers for timescales ranged from hours to months for the setup, hours to weeks for running a simulation in the FFEA runner, and 30 seconds to 15 minutes for loading simulation data in the FFEA visualizer.

Participants generally lent towards a positive outlook on the speed of the FFEA software suite components. When asked if they run at an appropriate speed, on average participants highly agreed for FFEAtools and the FFEA runner, and agreed for the FFEA visualizer. FFEAtools's speed was mostly uncommented on with two participants saying they had no problem. The FFEA runner's speed had positive comments associated, with praise given to its reported comparably faster speed to MD and its decent scalability, though one user while still agreeing that it runs at a good pace said that it has the potential to be even faster if refactored to work towards parallelization. As for the visualizer's speed, while some criticised its slow performance

for large systems, this was not described as a major issue, with some justifying the speed by saying that it is an issue of PyMOL rather than the FFEA visualizer plugin itself, or that other simulation visualization software like VMD was similarly slow for large systems.

More disagreeable notions were made, however, to the length of the setup process. While less complex systems can take up to only hours to set up, more complex systems that may be a user's main project take weeks to months to set up. A user's main project is expected to take a lengthier amount of time as they perform the problem solving needed to create a viable and useful simulation, but this process is made slower by the difficulties encountered in the main workflow, with a common complaint being the trial-and-error nature of editing files for processes such as positioning. More of these difficulties were discussed earlier in the analysis of work practices.

It seems that for the FFEA software suite the real bottleneck in the speed of using it is in the setup of systems rather than the running of simulations, even if the latter still has room for improvement. This could suggest that the parallelization of the FFEA runner source code be considered a lower priority compared to improving and adding functionality to making the setup of the common workflow an easier and therefore quicker process. If a user has to spend six months to set up a system, even a week wait for results is only approximately $\frac{1}{27}^{\text{th}}$ of the total time taken.

4.3.4 Collaboration and communication

The ability to collaborate using the FFEA software suite was met with generally favourable answers. Participants mentioned that different people could easily work independently on separate parts of the workflow, however, existing difficulties with the common workflow (such as the difficulty in setting up complex systems and areas of poor documentation) can inhibit this by making it hard to introduce new people to FFEA to work with. Another criticism was that FFEA has more input files that users would have to share compared to other MD simulations.

For its communicative abilities, all participants had praise for the FFEA visualizer. Its presentation of molecular objects as 3D shapes was considered a great communicative tool to a more general audience for being a more intuitive representation compared to ribbons and sequence schematics, and that even if a user wished to include the more-common representations structural biologists are familiar with, the 3D shape could still be overlaid translucently over them

to still have this clearer representation. The visualizer's communicative abilities were not considered perfect, for example, the rods were criticised for being a less clear and convincing representation, but they were thought highly of by participants.

While the collaborative and communicative abilities of the FFEA software suite was not the major focus of this evaluation there are still two main things to glean from the discussion of it. First is that the collaborative ability of FFEA while generally considered favourable is still somewhat victim to the difficulties presented when discussing the setup of the common workflow. Secondly, the communicative abilities of FFEA appear well-regarded, and while not many things hamper it, it may still be worth investing effort into fixing these issues and looking into how the communicative abilities of FFEA may be improved, as it has potential to be a highlight of the software and the visuals it produces.

4.3.5 Developer work practices

The two developers that had answered the survey had some similarities in their roles, with both having worked on documentation and debugging, but one of the developers had additionally developed a new algorithm within the source code, performed refactoring, and developed source code tests (though for the latter the other participant did highly regard testing as very important).

Remarking on the quality of the codebase, both participants expressed that it is quite inconsistent, and some parts of the source code are better written than others. The source code considered to be of higher quality was `FFEA_rod`, the finite element code, and the solvers. However, the collision handling, I/O, input file parsing, simulation box, and general Python code (especially for FFEAtools) were considered of lower quality and in need of maintenance or rewriting. Issues with the source code appear to be made worse by there being no extensive developer documentation outside of `FFEA_rod`, with only explanations of what lines of code do rather than entire files and objects. These issues build up to a lot of difficult to understand source code, especially for the FFEA runner as suggested by the developer statement results (§4.2.2). A lot of suggested features by developers address these problems, such as rewriting unmaintainable FFEA runner source code, porting FFEAtools to Python 3 and restructuring its code, and creating a set of coding standards for developers to avoid these inconsistencies and problems to reoccur.

Some participants discussed, however, an issue with how development of FFEA is practised; that problems arise due to the majority of the FFEA software suite's source code being written for postgraduate projects, making the code become "a lot of patches on top of patches". This was stated by one of the developers, as well as one of the users in their answer to Section C. The former suggested that a dedicated research software engineer be hired to work on FFEA. If this is done, then there are several suggested features and projects that may not make for a good student project that they could be directed towards that would make new development (especially of brand-new features that *would* make for a good student project) an easier process, and ones that in a cleaner codebase would be less likely to add to problems with unmaintainability.

4.3.6 Developer experience

From the two developers that took part in the survey, the FFEA software suite gained an overall average answer of "disagree" to the positively worded statements on ease of development, ease of understanding the source code, and enjoyment of development. The different components of FFEA varied on their average answers, with a "highly disagree" to statements about the FFEA visualizer, "disagree" on the FFEA runner, and "agree" on FFEAtools. While developers express that they gained satisfaction from their work on FFEA, there appears to be many issues in the developer experience. Some of these were discussed previously when discussing work practices such as the poor developer documentation and inconsistencies in the source code, but there are a few other points to discuss.

Criticism was made towards the branch structure of the FFEA source code repository on Bitbucket, in that there are many defunct branches. Looking at the repository, as of August 2021, there are a total of 30 branches, 19 of which are active. The number of branches that have been updated since August 2020 is 10, seven of which are active. The participant that brought this up suggested adopting an alpha-beta-gamma branch model that had been discussed in meetings with the FFEA development team. These branches have been created but have yet to see much use, with only the alpha branch containing commits, however, there have not been any large enough changes to the codebase since their introduction to make use of the beta and gamma branches. There are plenty of existing branches that could be dismissed and making use of the new alpha-beta-gamma structure could be an easy way of preventing this problem in the future.

There are some other issues in the codebase itself that makes development difficult. Reportedly,

data is unnecessarily nested deep inside complex data structures, there are poor naming conventions for variables and functions, the lack of public methods and variables means a lot of code must be redundantly rewritten instead of reused, the Python code is written in a C++ style with structural issues, and the C++ has a confusing layout that has comments of inconsistent levels of detail and usefulness. All of this appears to be contributing to a source code that is of inconsistent quality and style, full of unnecessary redundancy, and that is hard to understand. These are further things that should be considered for writing a set of coding standards for the development of FFEA to try and solve these issues and prevent them from reoccurring as new developers work on the software.

4.3.7 Qualitative analysis conclusions

This chapter has detailed the answers given to a qualitative survey on the current state of the FFEA software suite, and then applied analysis to these answers to glean important information and provide suggestions for future development and practices to address this information.

A reoccurring issue that has come up throughout this study is difficulties in the common workflow that cause dissatisfaction in many areas of the setup of FFEA systems, make it difficult for new users to pick up and learn, and increase the length of time it takes to work on a system. These problems may cause a reduction in user retention if new users are unable to figure out how to work effectively with the FFEA software suite. One developer saw the lack of usability for new users and the difficulty of creating complex systems through writing FFEA scripts to be some of the biggest issues in FFEA that requires attention. However, care needs to be taken to ensure that the common workflow is not the *only* part of FFEA that receives developer attention, as less-common functionality is much harder to use due to problems such as lack of detailed documentation.

A complicated issue with the FFEA visualizer is the use of PyMOL as a framework. There are many complaints made against it, but it is not a simple problem to address and make changes due to the difficulty in finding a new framework that would not introduce just as many issues or the effort it would take to create a visualizer from scratch. However, it is clear that the participants desire more interactivity in the FFEA software suite to make a lot of its functionality easier and address many of the problems in the common workflow. If this is something that PyMOL is not capable of then a new visualization tool may be a requirement

for moving forward.

While there are many things to improve in FFEA, there are still many positives of the software pointed out by participants; its effective communicative abilities through its visuals, the ease of running and managing the software on HPC, the human-readability of FFEA files, the ease of executing the FFEA runner etc. Much praise was also given in general to FFEA's latest major feature of `FFEA_rod`, which could provide guidance through example to new development and lay down foundations for coding and development practices and conventions for the FFEA team to adopt. Through understanding the benefits and merits of the FFEA software suite and the issues that hinder these merits, as well as the relevance and user adoption of FFEA, this will hopefully provide current and future developers of the software with as much relevant information as possible to assist them with improving the software and the practices used to develop it.

The design of the survey was effective for a holistic qualitative analysis of the sustainable qualities of FFEA 2.7.2, and with the exception of questions used to analyse FFEA's ability to collaborate on research work and communicate its results, these questions could be easily applied to any general research software by simply changing the topic software of the questions. The key limiting factor for this qualitative analysis however was the small sample size of five participants. This was difficult to avoid, as FFEA is a very niche piece of research software, however this does mean that any statistical analysis that could be performed would be weak, and that there were less opportunities to test this method of qualitatively evaluating research software to see if there are any gaps in the survey questions. It would be beneficial to re-test this methodology with a more well-known piece of research software in order to get a much better sample size and to test the effectiveness of this analysis method more rigorously. One set of questions that might be of benefit to add to the survey are more questions about the participants themselves, such as information about their career experience to compare the user experience of novice researchers against more experienced researchers. As it was expected that the survey would only get a small number of participants, and so be of little statistical use, questions about personal information were avoided for this specific case.

4.4 Identifying the bottleneck in the workflow

Over the past two chapters FFEA 2.7.2 has been quantitatively analysed as a piece of software, and qualitatively analysed as a tool. These two chapters provide us with a holistic understanding of FFEA 2.7.2 in order to answer the the first question in the research scope (§1.1), to understand the qualities of the software that are impacting its sustainability. From the results of these two chapters it can be determined where the major bottlenecks are in the workflow of FFEA, and what causes it. Does the software have performance issues that causes major disruptions to a user's research, or are there flaws in the features of the software as currently designed that slow the process?

In the previous chapter, the static and dynamic code analysis of FFEA 2.7.2 highlighted a few potentially-serious issues. From the results of `FFEA_linter` it can be seen that the C and C++ code for the FFEA runner contains a large number of errors and warnings of potential errors. A lack of standardisation for the code can cause difficulty in the software's development, which can contribute to software decay, as it is easy for a group of developers to be confused by each other's work if each member formats and structures their code differently. Some of the responses to the survey in this chapter confirm some of these issues for FFEA 2.7.2, with participants stating their difficulty in understanding and developing existing code for the FFEA runner due to bloated files, unnecessarily hardcoded features, inconsistent code comments, and incomplete documentation.

A similar picture is painted for the Python code for FFEAtools. As previously mentioned, FFEAtools in FFEA 2.7.2 is written in the now-unsupported Python 2, and in a very outdated project structure that is incompatible with modern Python 3 project structures. This is reflected in the poor results from `FFEA_linter`, where the Python 2 code was given a linting score of -11.09/10. The poor state of the code has caused some reported difficulty in development from FFEA developers due to the "complicated" file structure and interdependencies from files importing each other making it difficult to tell how a change to the code may impact other parts of it. One participant said however that adjustments to the FFEAtools code itself is fairly easy. Some users also reported difficulty in installing FFEAtools due to import errors related to Python 2's end-of-life.

The Python 2 code is so outdated that a migration process to convert it to Python 3 would

be very complicated and prone to error, as the code structure *and* contents would need changing simultaneously. It is suggested that FFEAtools is updated to Python 3 through a re-implementation of the toolset. Until this is done, this will bottleneck some of the development process, as the FFEAtools code will need this re-implementation done before any new tools can be added to it, and before any major maintenance to existing tools can be performed. Tools can be written separately for now while this re-implementation is in progress, but that will split up the auxiliary FFEA software, meaning users will have more installations to perform, which could be off-putting to new users.

The results from `FFEA_profiler` showcase that the FFEA 2.7.2 runner contains memory leaks, with the worst offender, the line `ffea.cpp:165`, recorded as causing a minimum of 254KB memory loss, and a potential maximum of 186MB memory loss. If developers were to focus on code optimisation, then it is suggested that fixing this memory leak is the highest priority, and that the software should be re-profiled afterwards to see if all the potential indirect leaks were also fixed in the process.

While the memory leaks are significant and should be addressed at some point, they do not seem to be fatal problems. When soak testing the FFEA 2.7.2 runner, not a single run was terminated due to a hardware-related crash, only due to software-related physics errors. This suggests that the memory leak, as well as all other problems found by `FFEA_profiler`, do not cause enough of a problem on the running of the software. Participants of the survey also never mentioned any memory-related problems when using the FFEA runner. While some participants did note long wait times for some simulations to run, which may possibly be improved by fixing the memory leak (although to a currently-unknown degree), this wait time was meagre in comparison to how long it takes to set up a simulation in the first place.

According to participants of the survey, while running a simulation can take between under an hour to (if a particularly large and/or unstable simulation) a few weeks, setting up a simulation can range between a timeframe of hours to *months*. Participants mainly pointed towards the difficulties in mesh creation and system parametrisation as to why setting up an experiment can take so long. These difficulties result from mesh and system setup requiring a large number of manual trial-and-error steps.

The main issue users had with FFEA system parametrisation was that it is all done through text fields, including the positioning and connecting of FFEA objects. There is no graphical or

interactive form of manipulating a system, such as by drag-and-drop positioning. Users must change these parameters in text fields and then re-run the simulation to see if the changes implemented correctly, which makes the editing of parameters a longer process than it could be if instead parametrisation was done interactively with instantly-visible results.

Producing an FFEA mesh from input data requires multiple functions from FFEAtools, any of which may fail due to a previous step producing an unworkable geometry (for example a surface profile may fail to calculate a coarse-grained geometry). Even a completed mesh is not without risk, as it is common for FFEAtools to produce meshes that contain unstable elements in their geometry, usually by having edges that are under the minimum length limit for continuum mechanics. These unstable meshes cause "element inversion" errors (see §5.2.1 for more details), where the FFEA simulation terminates due to non-physical behaviour.

While the FFEA 2.7.2 code contains a number of concerning issues that should be addressed, the results from the qualitative analysis recorded a huge cost-investment for users who wish to set up experiments with FFEA due to the difficult and time-consuming process of setting up such an experiment, requiring potentially months of labour. This makes simulation setup a clear major bottleneck to the workflow of FFEA users. If the FFEA development team wishes to remove the bottleneck, then the design of the workflow must be changed to something more reliable and user-friendly, such as in how meshes are created or how FFEA systems are parameterised and edited.

The subsequent chapters to this thesis will be dedicated to work designed to remove as much of this bottleneck as possible within the confines of this research. Chapters 5, 6, and 7 detail the design and development of a new meshing algorithm for FFEA. The goal of a new meshing algorithm is to attempt to reliably create meshes of a stable geometry by using defined shapes for mesh elements. It will also be attempted to design such an algorithm in a more user-friendly way to the FFEAtools method of mesh generation by having fewer steps for a user to perform. If such a meshing algorithm is successful enough to be worthy of replacing the older meshing algorithm, then this would also mean less work for developers to do when re-implementing FFEAtools, as instead of re-writing the FFEAtools meshing algorithm, it can be supplanted with the new one. Whether or not the FFEAtools meshing algorithm *should* be replaced is dependant on the response to this thesis.

Finally, Chapter 8 details an attempt to improve part of the workflow of FFEA system para-

metrisation through the identification of elongated biomolecules to automate the process of generating KOBRA rod meshes. The aim is to design an algorithm that can detect if an input biomolecule contains one or more rod structures, identify their location, and then automatically generate KOBRA meshes and connect them to any other existing FFEA objects. If successful, this will remove a number of steps in the setup of a simulation that uses KOBRA meshes which before had to be done manually, and should therefore improve the ease of the user-experience.

Chapter 5

Fluctuating finite element analysis tetrahedral meshing

The evaluation of FFEA 2.7.2 in Chapters 3 and 4 allowed us to identify a bottleneck in its workflow due to the difficulty in creating stable meshes for simulations. This issue could significantly impact the user retention of FFEA, as a difficult and time-costly workflow will make it harder to encourage new users to try the software, and train them with it. An overly-complex software workflow also means there are more potential failure points in the source code that require developer attention and maintenance, and thus more opportunities for code decay. Therefore, in order to extend FFEA's lifespan, the sustainable qualities of the software must be improved by making changes to the functionality to simplify and stabilise the workflow for the user and developer.

The following chapters are dedicated to improving the sustainable qualities of FFEA based on the findings of the software evaluation. Chapter 5 discusses the current problematic functionality of FFEA to understand why it causes workflow issues, so that new features can be developed to address these issues without recreating them. Chapter 6 details the development of a new meshing algorithm to create a simpler and more stable meshing process, with a comparative analysis then performed in Chapter 7 to compare against the old meshing process discussed in Chapter 5. Chapter 8 presents a proof of concept of another new function aimed at improving the workflow through using classification algorithms to automate part of the meshing process. These chapters make up the "Improve sustainable qualities of research software" stage of the

general method as outlined by Figure 1.1

This chapter details how FFEAtools creates tetrahedral meshes and the limitations of this method that cause disruptions to users' research. Then, it is discussed how members of the FFEA team began work on an alternative meshing method in response to this bottleneck. This makes up the "Understand current functionality to be changed or replaced" step of the general methodology (Figure 1.1).

5.1 File formats for FFEA tetrahedral meshing

Meshing for FFEA requires the use of various file formats at different steps of the meshing process which will be outlined and explained here for ease of reference later. Table 5.1 lists all of the file formats used for FFEA tetrahedral meshing.

Table 5.1: List of file formats used in FFEA tetrahedral meshing

File Format	File Extension	Contents
Protein Data Bank	`.pdb`	3D atomic structures of biomolecules
CCP4 Map	`.map`	3D volumes of objects imaged with electron microscopy
MRC2014 Map	`.mrc`	3D volumes of objects imaged with electron microscopy
Wavefront Object	`.obj`	3D geometric structures
STL	`.stl`	Triangulated surface geometry of 3D structures
NETGEN Volume	`.vol`	Volumetric mesh structures
TetGen Elements	`.ele`	Tetrahedra vertex nodes
TetGen Faces	`.face`	Triangular faces
TetGen Nodes	`.node`	Coordinates in 3D space
Visualization Toolkit	`.vtk`	3D geometric datasets

The Protein Data Bank (PDB) format is a text-based archival file format that describes the 3D structure of biomolecules. These files list records that detail the context of the experiments performed to obtain these 3D structures, as well as lists the atomic data of the structure, such as coordinates, chemical element, occupancy, temperature factor, and the residue it belongs to (Worldwide Protein Data Bank 2012). As well as their archival purpose, PDB files can be read by atomic simulation tools such as VMD to experiment *in silico* with their molecular dynamics (Humphrey et al. 1996). PDB files are usually obtained from relevant databases for biomolecules such as the RCSB Protein Data Bank (Berman et al. 2000).

The CCP4 (Collaborative Computational Project) and MRC2014 (Medical Research Council) formats are equivalent binary file formats that describe 3D image and volume data obtained from electron microscopy. These files represent image and volume data as point cloud data by plotting

electron density values onto a gridspace (hence the general term for these files - electron density maps), as well as containing a header of relevant metadata (Cheng et al. 2015). Electron density map files can be read by molecular visualization tools such as UCSF ChimeraX (Pettersen et al. 2021) for structural analysis and various methods of visual representation. ChimeraX is also capable of converting MRC2014 and CCP4 files between each other via the "save as" function. Electron density map files are usually obtained from relevant databases for electron microscopy data such as the Electron Microscopy Data Bank (European Molecular Biology Laboratory – European Bioinformatics Institute 2023).

The Wavefront Object (OBJ) format is a text-based file format that describes the geometry of a digital 3D object. These files contain the positions of each vertex, the UV position of each texture vertex, the normals of each vertex, and the vertices of faces for each polygon (*Advanced Visualizer 3.0 Manual* n.d.)[1]. The OBJ format is not specialised to a specific type of object representation, and sees general use in computer 3D modelling.

The STL (stereolithography) format is a text-based or binary file format that describes the triangulated surface geometry (i.e. its 2D surface planes are subdivided into triangles) of 3D structures. These files contain the vertex coordinates that make up each facet of the surface geometry (Burns 1993). The STL format is commonly used in general computer-aided design and manufacturing (CAD and CAM), having been originally designed for 3D Systems's CAD software.

TetGen is a tool for generating tetrahedral meshes that uses a variety of text-based file formats of their own design to describe them (Weierstrass Institute for Applied Analysis and Stochastics 2013). There are three TetGen file formats that are used for FFEA meshing; nodes, elements, and faces. TetGen node files contain a list of coordinates to points in 3D space. These nodes are listed numerically so that the other file formats can refer to locations in 3D space by the index of the node, instead of having to specify their coordinates each and every time. Elements files contain a list of all the tetrahedra that make up the mesh. Each tetrahedron is denoted by the indices of the nodes where their four vertices lie. Similarly, face files contain a list of each triangular face in the mesh, with each face denoted by the indices of the nodes where their three vertices lie. TetGen meshes are designed to be simple and easy to use so that they

[1] An archived copy of the full original manual where the OBJ file format was first documented is not available, only the appendix that describes the file format. See more information from the USA Library of Congress at `https://www.loc.gov/preservation/digital/formats/fdd/fdd000507.shtml`

are suitable for different applications. They can be generated by inputting other 3D geometry files such as STLs. Alternative to TetGen geometry files, a user can instead use a NETGEN (Netgen/NGSolve Team 2023) Volume file as it contains all the same geometric information needed for FFEA, however this file type is unused in this research.

The Visualization Toolkit (VTK) format is a text-based or binary file format that contains 3D geometric datasets. These files contain geometric and topological data in a 3D gridspace, as well as attributes for this dataset at specific points or cells (and also a header of metadata for these contents) (Avila et al. 2010). The attributes described in a VTK file are decided by the user, as long as it is a supported datatype, making it suitable for multiple applications. This allows a user to store, analyse, and visualise 3D data, such as the physical properties of a solid object or a fluid in 3D space.

5.2 FFEA tetrahedral meshing process

Originally, meshing for FFEA required the use of an external tool (Hanson 2018) such as NETGEN (Netgen/NGSolve Team 2023). This is still an option to this day, however as FFEA was further developed, the team created the accompanying toolsuite FFEAtools, which included functionality for meshing biological objects (Solernou et al. 2018). Figure 5.1 shows an example of a mesh of a dynein motor protein created with this functionality.

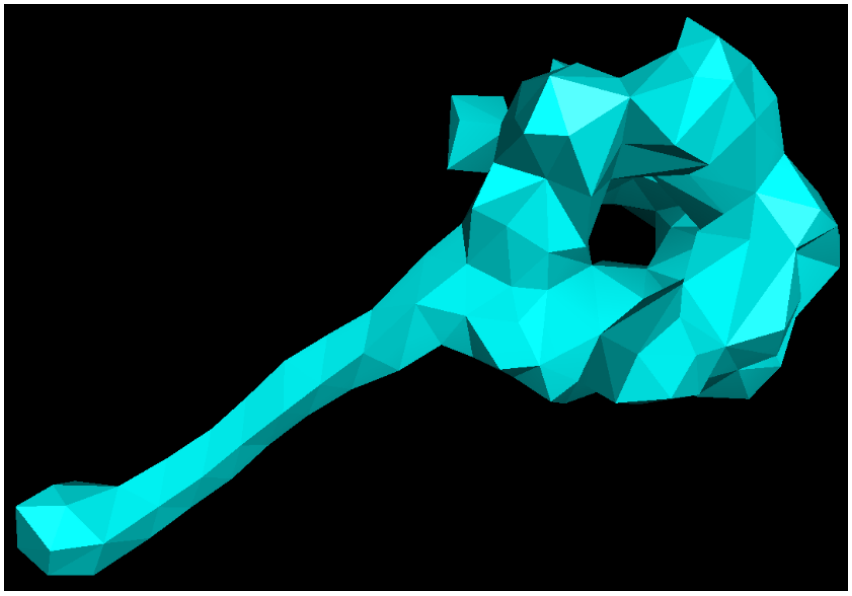

Figure 5.1: Example of a dynein motor protein mesh created with FFEAtools, viewed in PyMOL. Mesh data from Solernou et al. (2018).

A step-by-step process for meshing with FFEAtools was then showcased in a tutorial in the FFEA documentation (Cocking et al. 2021b). The process for meshing from atomic data is as follows:

1. With atomic data as input (such as in PDB format), convert input data to a electron density map (`.map`) with a user-defined number of voxels (in Cartesian format) and effective atomic radius in angstroms using the command `pdbtoemmap`. Alternatively, this step can be skipped by using an existing electron density map as input.

2. With an electron density map as input, optionally use the command `emmaptosurf` to process the map, such as by removing "floaters" (small pieces of mass separated from the main object) and filling internal cavities. This outputs another `.map` file.

3. With an electron density map as input, use the command `emmaptosurf` to convert to and output an `.obj` file of the object at a user-specified contour level (isolevel).

4. With an `.obj` 3D geometry file as input, use the command `surftocgsurf` to coarse-grain the object to a user-defined value. This approximates the shape by attempting to make the smallest edges the length of the coarse-grain value given (in angstroms) to reduce the computational load compared to an atomic model. This is especially necessary for FFEA which attempts to work with relatively large biological objects at the mesoscale. This outputs an `.stl` surface mesh file.

5. With an `.stl` surface geometry file as input, use an external tool such as TetGen (Weierstrass Institute for Applied Analysis and Stochastics 2013) or NETGEN (TetGen is recommended for stability in the FFEA documentation) to convert the input to a volumetric mesh. This output can either be in the format of a NETGEN `.vol` file, or in the format of TetGen `.ele`, `.face`, and `.node` files. TetGen files can also optionally be converted to the `.vol` format using the command `tettonet`.

6. With a volumetric mesh as input, use the command `voltoffea` to input the mesh and apply user-defined material parameters (such as the density, shear viscosity and modulus, bulk viscosity and modulus, effective node hydrodynamic radius, and dielectric constant). This will output all the files needed for running an FFEA simulation of this mesh to the defined material parameters, including an `.ffea` script file that links to all these files and is used to run FFEA simulations.

5.2.1 Limitations of the FFEAtools meshing algorithm

As explored from a user perspective in the previous chapter (§4.2.1), the FFEAtools method of meshing has a number of flaws. Most recurring issues centred around flaws in the process of steps that need to be done. Creating a mesh was described as difficult as there are a lot of steps to follow, many of which require manual adjustments to the mesh and its parameters in order to successfully generate one that would work well in users' simulations. These manual adjustments need to be done by altering text files or command-line text fields, lacking the ease that a GUI might provide. When a mesh *is* successfully generated, there still may be existing problems with it (such as self-intersecting faces or mesh element edges under the stable length limit) that compounds the time-cost of experiments, because if a mesh is unstable then a simulation may prematurely terminate frequently and require restarts, which is another manual process. Users may be able to overcome this by re-meshing their object, but this means re-doing previous and still potentially time-costly steps that will not guarantee a stable mesh. One user in §4.2.1 mentioned that these problems can also make it difficult to introduce new users to FFEA.

A common result of mesh instability is an error known as "element inversion". This is the result of non-physical behaviour occurring in simulation when a mesh element has too high an energy value, which will prematurely terminate the simulation. Usually this behaviour results from a mesh element's node moving too far during a single time-step in such a way that it passes through one of its own faces and "inverts" the element (see Figure 5.2). This is often caused by mesh elements that have one or more sides that are too small (sometimes called "sliver elements"). There are some workarounds to this, such as decreasing the time step of simulations, increasing the size of small mesh objects, or attempting to fix an existing unstable mesh in external meshing software, but these workarounds do not guarantee a fix to the problem, especially anything involving re-structuring the mesh, as that could introduce new unstable mesh elements. These fixes can still take significant time even when they do work, for the same reasons it can take significant time to create meshes in the first place. Meshes are found to be unstable in FFEA simulations, and prone to the causes of element inversions, when elements contain edges shorter than approximately 5Å. This is because below this length-scale the mesh material will be intrinsically atomic and will not behave in simulation like a continuum, and therefore cannot represent a biomolecule (while also being functionally useless to an analyst as there already exists good simulation methods for atomic models) (Oliver 2013).

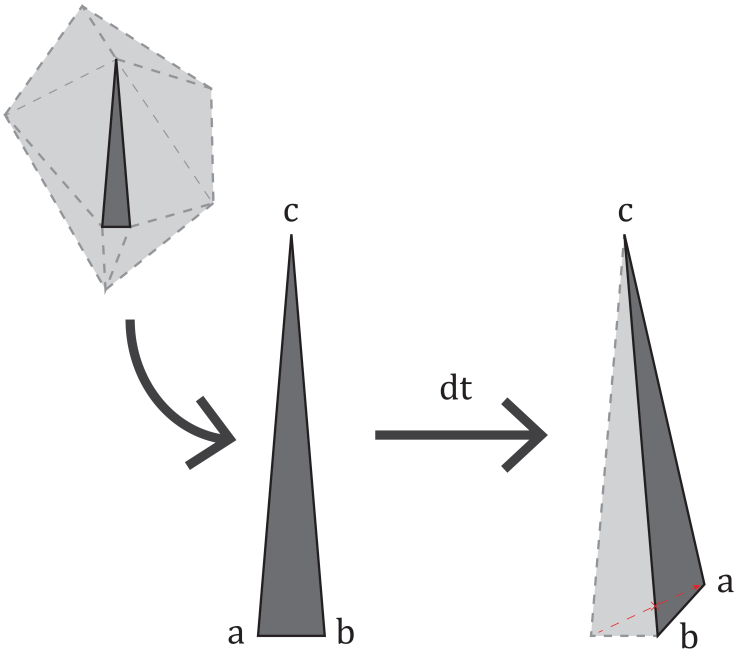

Figure 5.2: Example of mesh sliver-element inversion during a simulation timestep. In this example, due to the small length of edge ab, node a is able to move far enough during a single simulation timestep (dt) that it passes through edge bc, which is non-physical and causes premature termination of the simulation.

Multiple users during the survey expressed a wish for the workflow (or parts of it) to be automated. One participant stated how users need too much knowledge of how the meshing functions work in order to use them, and that users should just be able to input 3D image data along with some input parameters, and then get an output of a tetrahedral mesh ready to use in FFEA in a single step. One participant mentioned an attempt to automate the current meshing process, but found that the algorithms used by this process limited their ability to do so successfully (unfortunately their answer did not elaborate details), rather than limited by the software implementation of those algorithms.

Due to the problems with the meshing workflow, users can find themselves requiring external software, either to fix problems caused by the current meshing process or to circumvent problems a step is likely to introduce. Ideally the meshing workflow would require only the tools it provides so that users are not reliant on tools out of the control of the FFEA development team that could be changed at any time, and so that users would not need to learn how to use additional tools.

5.3 An alternative method of meshing

In order to overcome the limitations of the FFEAtools meshing method for their own research, Gravett (2022), working alongside Joanna Leng (both members of the FFEA team), began creating an algorithm to process 3D image data of molecules in voxel format from MRC files into tetrahedral meshes. This version of the meshing algorithm can be found at Gravett, Rogers et al. (2023) under the name `tet_from_pix`.

In this tool, the image data is re-sampled to make the data coarser as the image resolution is too high and is a significant factor is making the numerical method unstable. The algorithm uses the marching diamond crystal lattice (MDCL) algorithm (an alternative version of the marching tetrahedra algorithm (Doi and Koide 1991)) to slice each voxel over a mandatory user-defined threshold density value (a.k.a. the isolevel) into five tetrahedra. This slicing cuts the voxels in half diagonally across each three pairs of opposing faces (Figure 5.3) to create four trirectangular tetrahedra (a tetrahedron where all three face angles at one vertex are right angles), and one interior regular tetrahedron with alternating direction between neighbouring voxels (Figure 5.4) (B. 2020). This produces a more isotropic mesh with more uniform material properties than the standard marching tetrahedra algorithm. The tetrahedra coordinates are also transformed to the size of the initial molecule, so that the coarsening does not alter the length scale when visualised. These tetrahedral meshes are then output in the TetGen format (Weierstrass Institute for Applied Analysis and Stochastics 2013) which can now be used with FFEA, and also in the VTK format (Schroeder et al. 2006) for analysis in ParaView (Ayachit 2015). The authors had successfully implemented the algorithm to process MRC data and output the tetrahedral mesh files. This was the point at which collaborative development of this tool as a part of this thesis began.

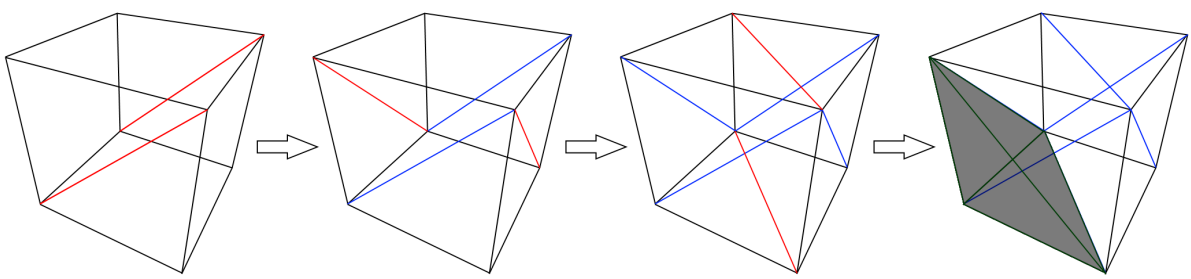

Figure 5.3: Voxel slicing via marching diamond crystal lattice. The voxel is sliced in half diagonally three times, once across each pair of opposing faces, to create five tetrahedra (red lines are new slices, blue lines are slices from a previous step). One of the trirectangular tetrahedra created is highlighted in grey and outlined in green.

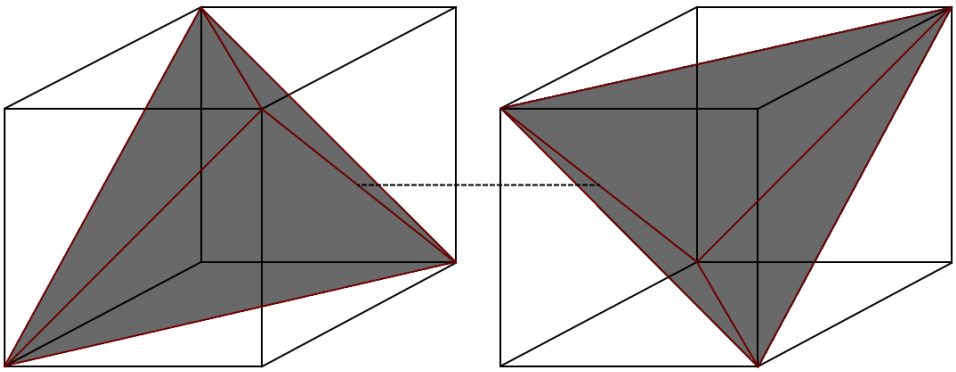

Figure 5.4: Alternating interior regular tetrahedra of adjacent voxels sliced by MDCL. Parallel non-touching faces have slices in the opposite direction, causing the inner-tetrahedron to be in symmetry with its neighbouring voxel's inner-tetrahedron.

5.3.1 Algorithm process

The following list explains the steps in the alternative meshing algorithm at this period in its development.

1. Input 3D image data in `.mrc` format.
2. Image data is placed on the cryo-EM map gridspace, where each cell of a grid is a voxel.
3. The algorithm iterates through the gridspace to choose which voxels are added to the output array, based on the user-defined density value required to be filled.
4. Output 3D mesh in voxels.
5. The voxels are sliced into five tetrahedra using MDCL (see Figure 5.3)
6. The 3D mesh data is output as `.ele`, `.face`, and `.node` TetGen files for use in FFEA, as well as in `.vtk` format for viewing in visualization tools such as ParaView.

A flowchart in Figure 5.5 describes how the algorithm functioned at the time as code as designed by Gravett (2022). This begins by inputting 3D image data from an MRC file, which the user has pre-processed including coarse-graining the data as desired. The algorithm then reads and extracts the image data and the dimensional data from the header. The user must also input the density threshold as a floating point number for use in later steps, which currently has to be done in-script.

The algorithm then iterates over each voxel in the MRC data and compares the voxel's density value to the user-input threshold. If the density is greater than the threshold, a cube is created

and its coordinate data populates an array which will contain the coordinates of all accepted cubes. These cubes are also transformed to have the same size and origin as the original MRC data to account for size transformations during any coarse-graining. From this coordinate array, NumPy (Harris et al. 2020) is used to create a unique array of all cube coordinates without duplicates. This unique array is written to a TetGen `.node` file to be the final mesh's coordinate data.

Once the algorithm knows which voxels are accepted into the mesh, it creates a cube connectivity array which is populated with lists of indices of coordinates that make up a cube. The algorithm then iterates over this connectivity array and slices each cube with MDCL (Figures 5.3 and 5.4) to create tetrahedra from the cubes. The algorithm checks if the index of the current cube is an odd or even number, and uses that to determine the directionality of the slices.

When all the tetrahedra are created, the algorithm then outputs the final mesh data. It first creates a VTK tetrahedra array from the tetrahedra data and writes it to a VTK file, so that the mesh can be analysed in visualization tools such as ParaView. Next, the algorithm creates a tetrahedra connectivity array, which is populated with lists of indices of coordinates that make up a tetrahedron. This array is written to a TetGen `.ele` file, which now lists each set of vertices that make up a tetrahedron in the final mesh. Finally, the algorithm iterates over each tetrahedron and extracts a list of indices that make up each face of that element. This array is then written to a `.face` file, which now lists each set of vertices that make up a face in the final mesh. These TetGen mesh files can then be used in FFEA simulations.

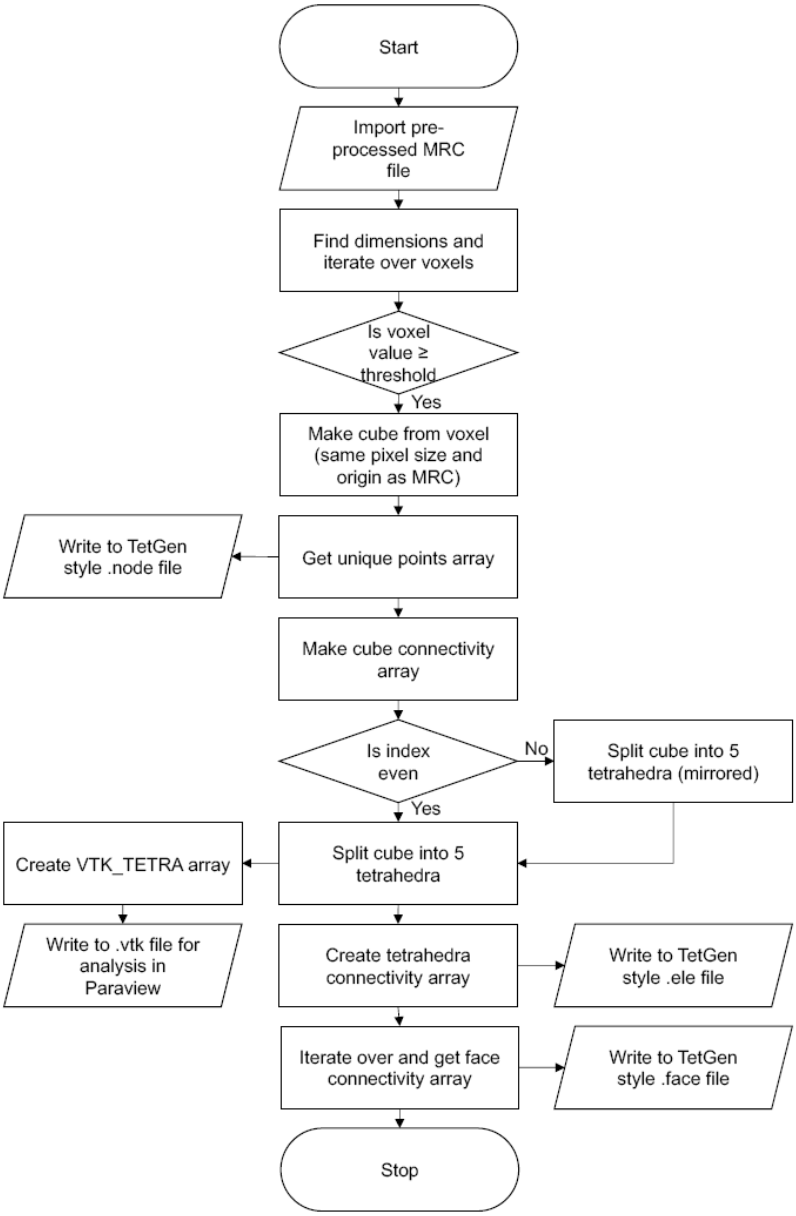

Figure 5.5: Flowchart of marching tetrahedra FFEA meshing algorithm process. Diagram by Gravett (2022)

5.3.2 Limitations of the alternative method of meshing and conclusions

At the time this research joined in with the development of the new meshing algorithm, coarse-graining the model had to be done by the user externally before inputting the data into the algorithm, with the suggested tool being UCSF ChimeraX (Pettersen et al. 2021). While the algorithm could create tetrahedra from sliced voxels, it could not yet do anything else with them, such as select specific tetrahedra to be included in the mesh. This meant that meshes were restricted to cubic voxel shapes. These limitations are addressed in the next chapter where the development of the alternative meshing method continues in order to bring it to a

feature-complete and user-friendly state.

Gravett (2022) also noted that the algorithm is currently unable to generate rod structures for FFEA (which is a different method of biological object representation in FFEA (Welch 2020)), commenting that it would be a useful feature, especially if it was able to automate combined blob-rod structures based on the shape of input cryo-EM data. They opined that a machine learning algorithm may be required to automate this process by predicting the type of structure based on the shape of the electron density map. This idea is explored in Chapter 8 where a prototype of such a machine learning algorithm is developed in order to test if this potential method of automation is possible and if it could further improve the user experience.

Chapter 6

Marching diamond crystal lattice tetrahedral meshing algorithm

In the previous chapter it was discussed how FFEA currently generates tetrahedral meshes and the limitations of that method which cause issues in the software's workflow. It was then discussed how Gravett (2022) with other members of the FFEA team began work on developing a new meshing algorithm for the software suite to produce meshes that perform with more stability in FFEA simulations. With an understanding of the issues of this current functionality that impact FFEA's sustainable software qualities, and with Gravett's work as a starting point, new functionality can be developed to improve the workflow and thus improve FFEA's longevity through more user-friendly and maintainable features.

This chapter details the development work completed as part of this thesis (i.e. the new work in this chapter was contributed by the author) on the new meshing algorithm up to the completion of a Python prototype implementation of the feature-complete method. In the general methodology (Figure 1.1) this is the "Develop new functionality to improve sustainable qualities of research software" step. This work includes adding a coarse-graining function to the algorithm, adding an interpolation function to smooth the mesh surface, providing a warning message to users who have generated a mesh with parameters under the known stable limits, as well as general code improvements for ease of use and development, and bug fixes. This method of mesh generation and its Python implementation is tentatively referred to from now on as the Marching Diamond Crystal Lattice Tetrahedral Meshing Algorithm, or MDCL-TMA.

The source code of the Python prototype of MDCL-TMA is stored on Bitbucket (Gravett and Rogers 2023)[1], which was forked from the original repository for the new meshing algorithm (Gravett, Rogers et al. 2023) (from commit `5d7645d`).

6.1 Coarsening the mesh

To coarsen the mesh internally a function was created that coarse-grains the input MRC data if the user supplies a resolution (R) (as a floating point number) to use for scaling. This function was written up in the script `mrc_zoom` which is called by the rest of the algorithm in `tet_from_pix`, but can be executed separately.

The coarsening function works by parsing the original MRC data and changing the scale variables of the voxel dimensions and data by the scale factor as calculated by

$$SF_D = \frac{\text{mrc_input.voxel_size}}{R}. \tag{6.1}$$

This data scale factor (SF_D) is in the format of a tuple of size three, one each for the x, y, and z dimensions. The input data uses a cubic gridspace, and the MDCL-TMA algorithm does nothing to attempt to change that, so each dimension of the scale factor should be approximately equal, with potential minor differences due inherited differences from the input data and from floating point errors.

The data scale factor is applied to the input MRC data using SciPy's `ndimage.zoom` command for image processing. SciPy's `ndimage` is a package for multidimensional image processing, and its `zoom` command uses spline interpolation to zoom an array across each axis by a defined factor (The SciPy Community 2023). This transformation changes the size of the array and transforms the array data so that it is equivalent to the old data at this new size (see Figure 6.1 for an example). In this case, the input MRC data is zoomed using `ndimage.zoom` by a factor of the data scale factor calculated by Equation 6.1.

[1]This repository is currently set to private due to potential licensing issues. While we wish to commit to open access to source code, this must first be addressed.

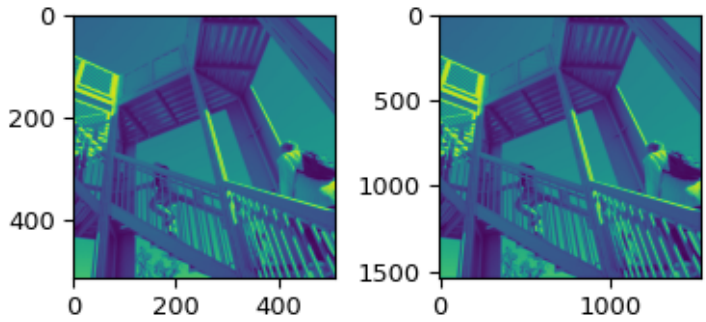

Figure 6.1: Example of an input image (left) zoomed using `ndimage.zoom` (right) by a scale factor of 3. The image remains looking the same despite being scaled up as seen in the axes. Diagram by The SciPy Community (2023)

A new MRC file is then created for this transformed data to be written into. However, while the MRC data has been transformed at this point, the header data is what determines the size of the voxels, and this needs to be set to the appropriate size. This can be calculated by comparing header data in the old and new MRC files. MRC files have header data variables called `nx`, `ny`, and `nz`, which are the number of columns, rows, and sections in the data array. I_{nx} and C_{nx} denote the `nx` values for the input MRC file and the coarse-grained MRC file respectively, and likewise for `ny` and `nz`. It can be determined how to scale the dimensions of the voxels by calculating scale factors with

$$SF_V(x) = \frac{C_{nx}}{I_{nx}}, \tag{6.2}$$

and then the voxel dimensions can be transformed by the voxel scale factors (SF_V) with

$$\text{voxel size (x)} = \frac{1}{SF_V(x)} \times I_{nx} \times \frac{C_{nx}}{C_{nx} - 1}. \tag{6.3}$$

The same equations are used for each axis.

The new MRC file is then output by `mrc_zoom` for use in the rest of the algorithm process. This new file automatically supplants the original input MRC data if the coarsening function is called from within `tet_from_pix`.

6.2 Interpolation for a smoothed mesh

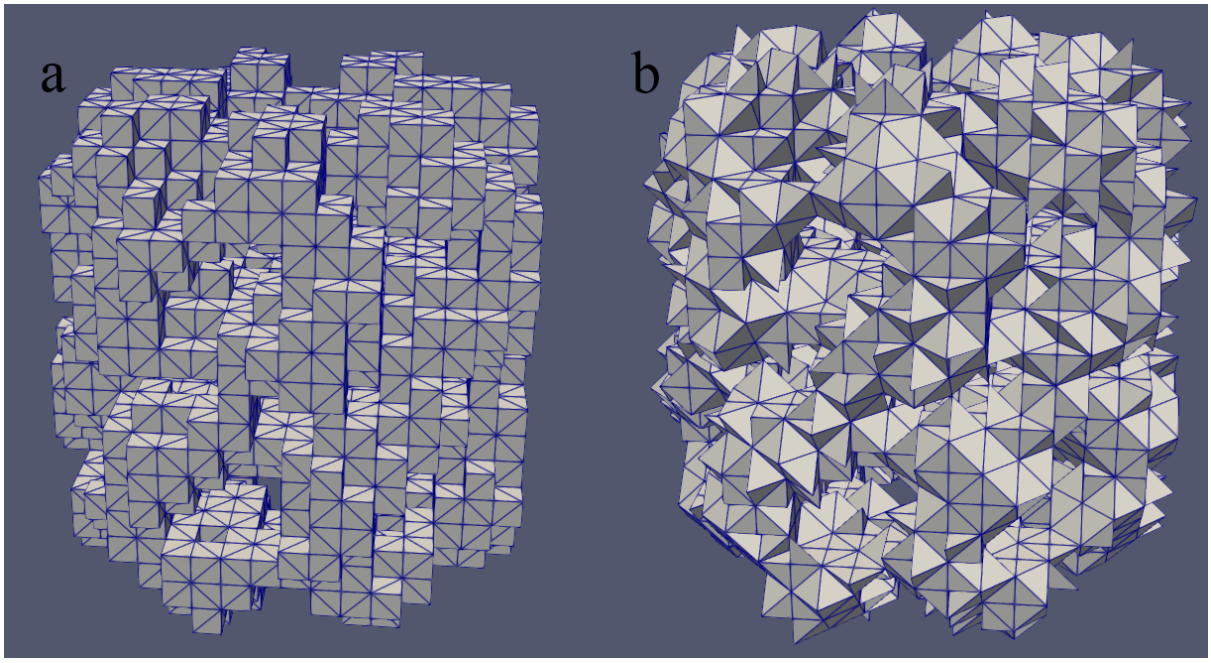

Figure 6.2: Comparison of groEL MDCL-TMA mesh without (a) and with (b) interpolation to smooth the surface.

A feature was added to smooth the mesh through interpolation by selecting specific tetrahedra for inclusion in the mesh, rather than selecting by voxel. This allows the algorithm to create a mesh with a surface that better represents the shape of the original biomolecule, instead of a purely cubic structure (see Figure 6.2 for a comparison). A more accurate shape means a more accurate distribution of the biomolecule's density, and thus more accurate physics applied to the object. Subjectively, it also means a better visual representation of a biomolecule for scientific communication in figures and recordings of simulations that use MDCL-TMA meshes.

This interpolation scheme was implemented by adapting the step in the algorithm that utilises the marching diamond crystal lattice algorithm so that each tetrahedron is assigned a density data value of the average of the density values at their four vertices (essentially a linear interpolation of their values). This is expressed by

$$\text{tetrahedron}_d = \frac{A_d + B_d + C_d + D_d}{4}, \tag{6.4}$$

where A, B, C, and D represent the four vertices of the tetrahedron and d represents a density value.

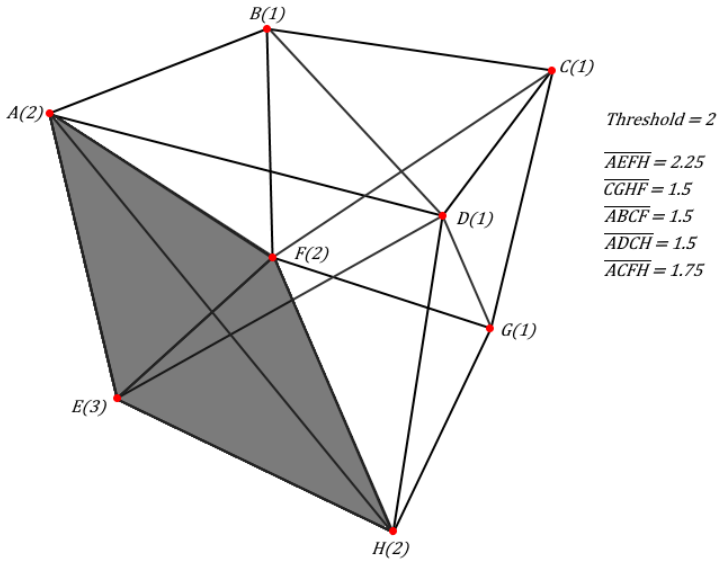

Figure 6.3: Example of interpolation of one voxel in MDCL-TMA. Numbers in brackets represent the data value of that vertex. The data value of a tetrahedron is equal to the mean of its vertices' data values. If the tetrahedron's data value is greater than the user-defined threshold value, it is accepted as part of the mesh structure, otherwise it is discarded. In this example, the tetrahedron highlighted in grey ($A\bar{E}FH$) is accepted as its calculated data value of 2.25 is greater than the threshold of 2.

Then, for each of these tetrahedra, the user-defined threshold is applied by accepting tetrahedra with density values greater than the threshold (see Figure 6.3 for an example). The algorithm counts the number of elements that will be in each array which stores coordinate and data values. Zero-arrays for both are created of a length of the number of elements. The algorithm then re-iterates over each voxel, applies the threshold to each tetrahedron again, and now puts the coordinate and density data values of each accepted tetrahedron into their respective arrays. This outputs a smoother mesh of tetrahedra that is not confined to producing only fully packed voxels.

6.3 Stability check

As FFEA developers have found there to be stability issues with meshes that contain edge lengths under 5Å (§5.2.1), it felt appropriate to warn users about this. MRC data can return voxel dimensions using the Python library `mrcfile`, which allows Python read and write data and headers in the MRC file format as NumPy arrays (Science and Technology Facilities Council 2017). In the current Python prototype of MDCL-TMA, this is used to check if a user's input data (coarsened or not) produces voxels with edges under 5Å, and if so a command-line warning

will appear (Figure 6.4). The user is not blocked from continuing however, as mesh instability is not a guarantee under this edge length, and a user of MDCL-TMA may have reasons other than their use in simulations to mesh an object. Users should not be prevented from being able to use a working mesh, even if it is not recommended to use in simulation due to materials not acting as a continuum at this scale.

```
(base) [sccdrADM@uol-lap-d2t6063 mdcl_tetrahedral_meshing]$ python tet_from_pix.py -i /home/sccdrAD
M/Documents/FFEA/Meshing/analysis/testdata/12_her2tp/mdcl/emd_20055.mrc -o /home/sccdrADM/Documents
/FFEA/Meshing/analysis/testdata/12_her2tp/mdcl/emd_20055_mdcl_2ang -t 4.2 -r 2
WARNING: Elements of the generated mesh contain edges shorter than 5Å, which can cause instability.
 If you find this to be the case, we suggest coarsening the input further using the -r command.
```

Figure 6.4: MDCL-TMA Python prototype command-line user warning of potential instability for meshes coarsened under 5Å.

6.4 Bug fixing and code tidying

In order to make the MDCL-TMA code easier to operate, simple command-line options were added for parametrising the meshes, which before were in-script variables to be manually changed. An issue was also solved whereby the output meshes were slightly off-set from the map data (Figure 6.5), so they would not align properly, which is important for comparing the atomistic PDB data, the electron microscopic map data, and the mesh. This was done by translating the tetrahedra coordinates by half the length of the voxels in the x, y, and z axes so that the new mesh would align with the original molecule when viewed in visualisation tools such as PyMOL.

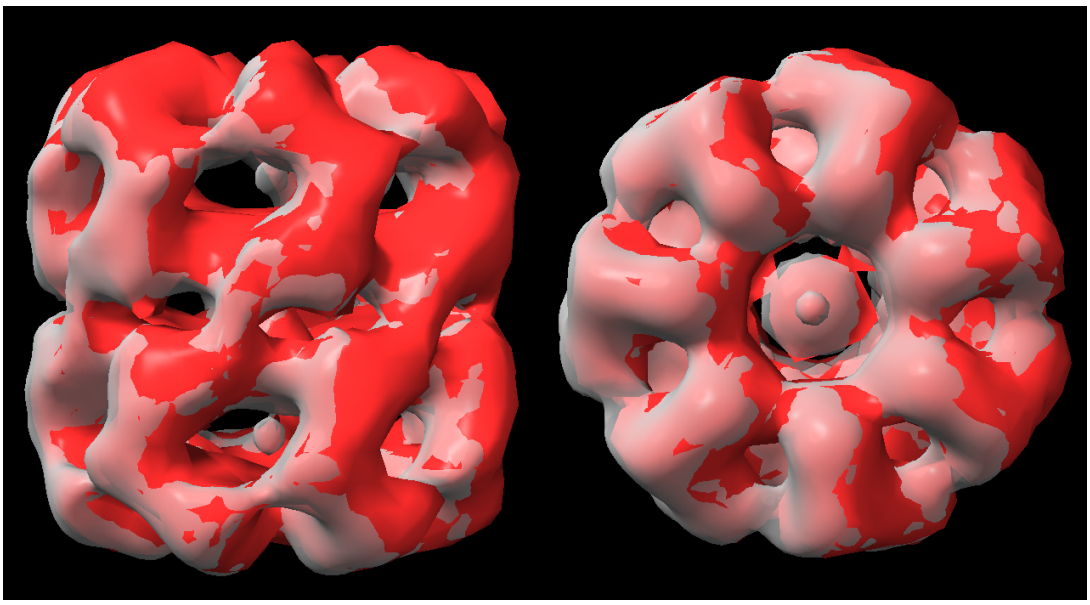

Figure 6.5: Comparison of original MRC data (transparent white) with overlaid bugged output MRC data (red). The two should overlap, but the output data is slightly off-centre.

6.5 Feature-complete algorithm process

The following list explains the steps in the feature-complete MDCL-TMA algorithm. A 2D example of this process can be viewed in Figure 6.6.

1. Input 3D image data in `.mrc` format.
2. Image data is placed on the cryo-EM map gridspace, where each cell of the grid is a voxel.
3. The input 3D image data is optionally coarse-grained to change the size of output elements. This step is optional as a user may have already coarse-grained their input data from either a previous run of MDCL-TMA or with an external tool.
4. The voxels are sliced into five tetrahedra using MDCL (see Figure 5.3)
5. The algorithm iterates through the gridspace to choose which tetrahedra are added to the output array, based on the user-defined density value (isolevel) required to be filled (called the "threshold" by the algorithm). In the example, green elements are output and red elements are not.
6. The number of tetrahedra accepted into the mesh is counted, and data arrays for the tetrahedra are created with a length equal to this counted number.
7. The algorithm iterates through the tetrahedra again and reapplies the density threshold. The algorithm then populates the newly created data arrays with the coordinates and density data values of the accepted tetrahedra.
8. The algorithm outputs the tetrahedra data as `.ele`, `.face`, and `.node` TetGen files which make up a mesh for use in FFEA, as well as in `.vtk` format for analysing in visualization tools such as ParaView.

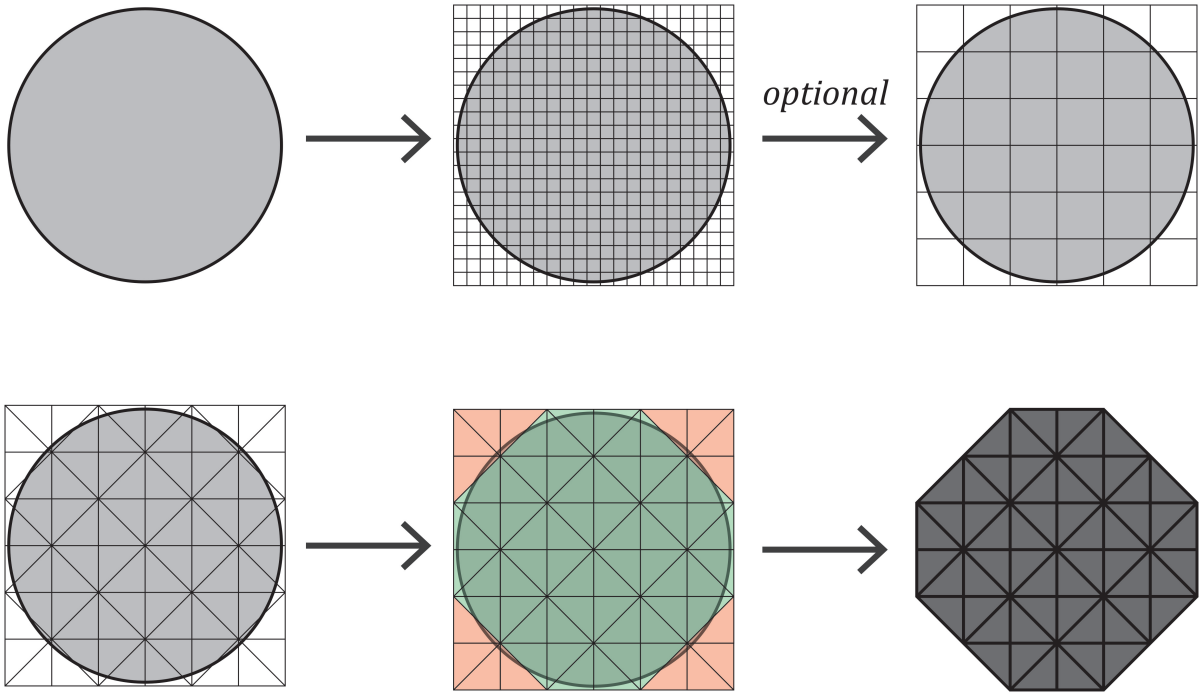

Figure 6.6: Example of the MDCL-TMA process rendered in 2D

A more detailed explanation of the MDCL-TMA algorithm process has been outlined by the flowchart in Figures 6.7 and 6.8. The flowchart describes how the algorithm functions as code. This begins by inputting 3D image data from an MRC file, which the algorithm reads and then extracts the image data and the header values of `nx`, `ny`, and `nz` (the number of columns, rows, and sections in the data array). The user must also input the threshold (T) as a floating point number for use in later steps.

If the user supplies a value for the resolution (R), the data is coarse-grained using the `mrc_zoom` function. The scale factor to transform the data by is calculated with Equation 6.1. The data array is transformed by this factor using spline interpolation via SciPy's `ndimage.zoom` command, which changes the size of the data array and transforms its contents to be equivalent to the old data in this new size. This new data array is written to a new MRC file. The header information of the new MRC file is updated to change the size of the voxels, which is done by calculating a voxel scale factor by comparing the header data values in the old and new MRC files using Equation 6.2, and then applying this voxel scale factor with Equation 6.3. These transformations are skipped if the user did not supply a value for R.

The algorithm then iterates over each voxel in the image data for the first time. In each iteration, a counter (i) is started. An if statement checks whether i is an even or odd number, which is used

to determine the directionality in which the voxel is sliced using MDCL (Figures 5.3 and 5.4) to create tetrahedra. Each tetrahedron is then assigned a density data value (d) calculated with Equation 6.4.

For this first iteration over the voxels, after each one has been sliced using MDCL, a new counter (E_n) is started at 0. The algorithm then iterates over each of the created tetrahedra, and for each if $d > T$ then E_n is increased by 1. Once the algorithm has iterated over each tetrahedron, E_n will be equal to the number of tetrahedra in the final mesh.

Two new zero arrays are then created, one for the coordinates for each tetrahedron (E_C), and one for the density values of each tetrahedron (E_d). Both of these arrays have a length equal to the value of E_n.

The algorithm then iterates over the voxels for a second time, repeating the slices of voxels via MDCL, and a new counter (k) is started at 0. Then, iterating over each created tetrahedron, if $d > T$, the value of E_C at index k is set to equal that tetrahedron's coordinates, and the value of E_d at index k is set to equal d. This populates the arrays with the coordinate and density data for each tetrahedron that will be in the final mesh.

During the second voxels iteration, while the arrays E_C and E_d are being populated, corrections are made to the geometry of the mesh. Without correction, the final mesh will be translated by half the length of a voxel across each axis. This is because the density data values are actually stored in the middle of each edge on a voxel, not their vertices, so when obtaining densities all the voxel coordinate indices are translated by 0.5. To reverse this once the algorithm has obtained the data it needs, the origin is extracted from the header of the MRC file, and then is added to each tetrahedron's coordinates to return each vertex to its original location.

If the user decided to coarse-grain the mesh, then the output mesh without correction would be of a different size to the original molecule data due to the zoom function, so the algorithm also needs to transform the mesh back to this original size. This can be accounted for during the second voxels iteration by multiplying each coordinate by a resolution scale factor (SF_R). This resolution scale factor is calculated by dividing one of the MRC's cell dimensions by the size of the relative linear array (i.e. the values of `nx`, `ny`, or `nz`). As the cells are cubic voxels, one only needs to obtain this resolution from one axis of the cell dimensions, so the x axis can

be used via

$$SF_R = \frac{\text{mrc.cell_dimensions.x}}{C_{nx}}. \tag{6.5}$$

At this stage in the algorithm, the full mesh's coordinate and density data have been recorded, and it now needs to populate output TetGen mesh files. Using NumPy, a unique array (E_U) is created from E_C, which contains every coordinate in the mesh without duplicates. E_U is then written to a `.node` file to be the final mesh's coordinate data. Next, the algorithm creates an element connectivity array (E_K), which is populated by lists of indices that make up a tetrahedron. E_K is then written to a `.ele` file, listing each set of vertices that make up a tetrahedron in the final mesh. Finally, the algorithm iterates over each tetrahedron and writes to a new array (E_F) a list of indices that make up each face of that element. E_F is then written to a `.face` file, listing each set of vertices that make up a face in the final mesh. These TetGen mesh files can then be used in FFEA simulations.

The algorithm also outputs two VTK files from the mesh data for analysis in visualization tools such as ParaView. E_K and E_F, used to output TetGen files, are also used to write this data into a VTK of the full final mesh, which is also populated with the density data values from E_d. A second VTK file is created by iterating over each tetrahedron in the final mesh and calculating the average coordinate position of its four vertices. This VTK file is then populated with a list of these average coordinate positions. This was mainly used as a developer aid to ensure tetrahedra densities were being correctly calculated from the average of their vertex densities, but the feature was kept in case it was somehow useful to users (and also for potential future developers).

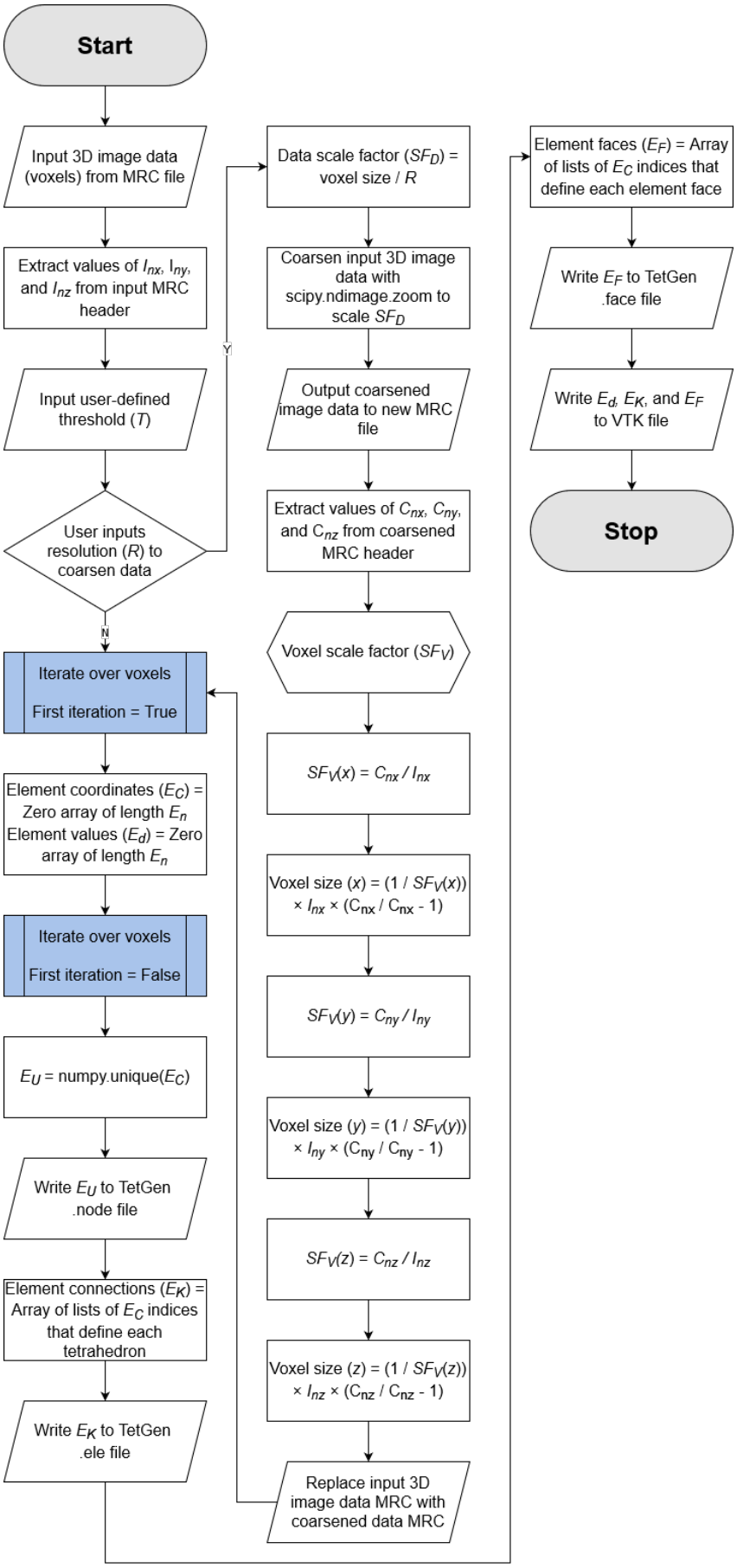

Figure 6.7: Flowchart of MDCL-TMA process. The blue highlighted sub-process is expanded in Figure 6.8.

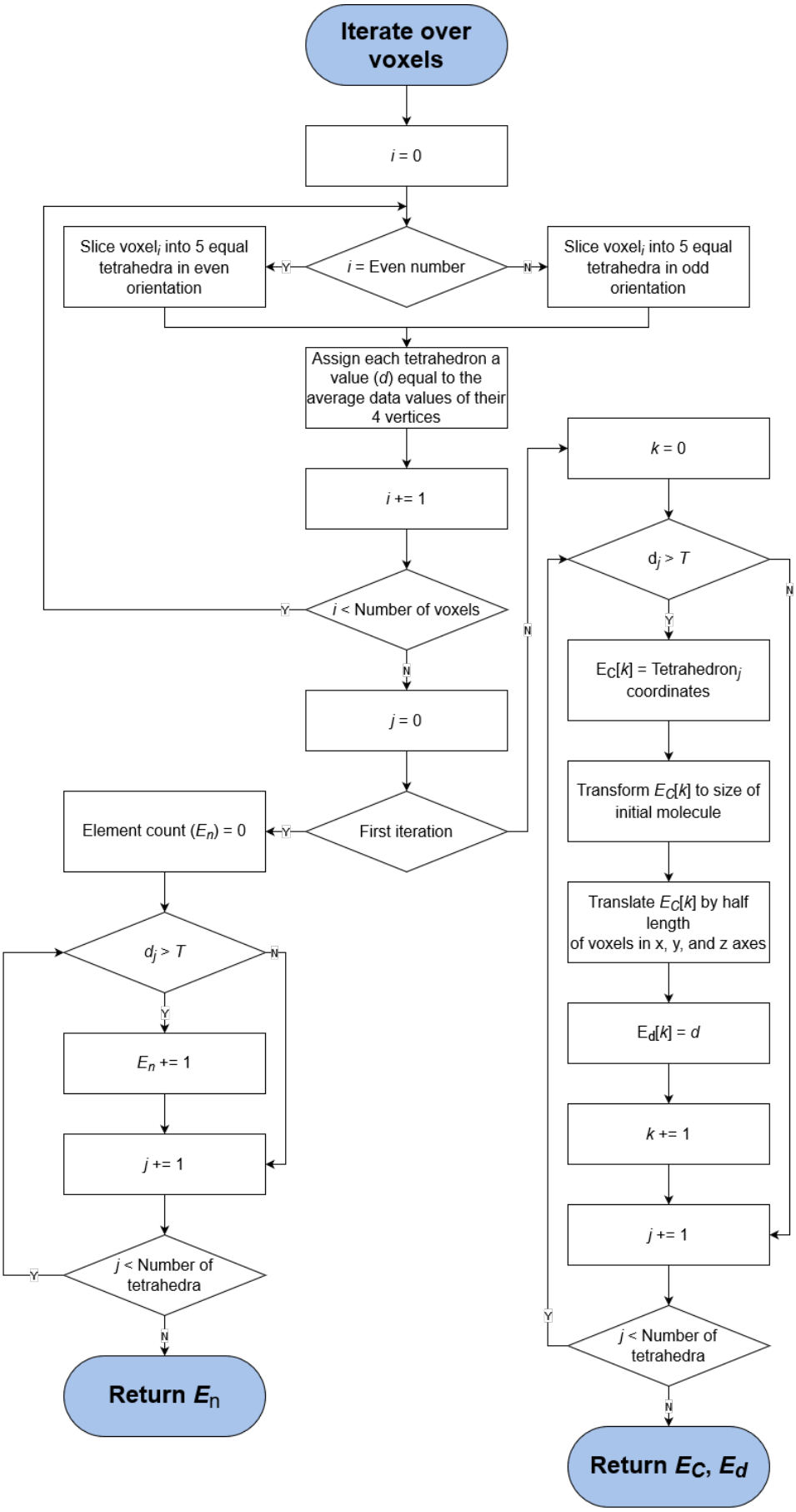

Figure 6.8: Flowchart of MDCL-TMA sub-process "Iterate over voxels". This returns to the process in Figure 6.7.

6.6 Summary of algorithm improvements

Before development as part of this thesis was performed on the new meshing algorithm for FFEA, originally developed by Gravett (2022), the algorithm successfully created tetrahedral meshes from electron density maps by creating voxel geometry from the point cloud density data and using MDCL to slice each voxel into five tetrahedra. However, at this point in development, the algorithm was not tailored for use outside of the experiments performed by Gravett (2022). Any coarse-graining of the object had to be done externally before inputting it into the meshing algorithm, and user-input variables needed to be input in-script. The algorithm also could only produce cubic structural representations of objects, as the density threshold was only applied to each voxel, not to the tetrahedra themselves.

As part of this thesis, several improvements were made to the new meshing algorithm for FFEA, tentatively named the Marching Diamond Crystal Lattice Tetrahedral Meshing Algorithm (MDCL-TMA) here. Usability of the algorithm was improved by adding a coarse-graining function, a basic stability check of the output mesh, and adding command-line options for user-input variables for easier operation.

The coarse-graining function changes the size of the density data array and transforms the array data to be equivalent to the old data at the new size, which allows a user to change the size of the voxels and thus the size of the mesh elements, without needing to do so externally. This means that users do not have to rely on learning additional and external tools as part of the workflow for mesh generation, simplifying the user experience without any change to data processing other than where it is done. A user can still coarse-grain their data externally if they wish, or even choose not to coarse-grain their data at all, as this step of the algorithm is optional, based on whether or not the user input a resolution for re-scaling the data. Having these options for this step means that more users will be accommodated, and therefore should improve user retention for a more sustainable piece of research software with better longevity.

The stability check performs a simple test to see if the output mesh will contain edge lengths under 5Å, as meshes under this limit usually perform poorly in-simulation. This is because mesh material will not behave as a continuum under this length-scale. Users are only warned about this eventuality, rather than stopped completely, as a user may still have reason to mesh objects at this length-scale even if they are not recommended for use in FFEA simulations.

This provides more clear feedback to the user on why some meshes might perform poorly in-simulation, instead of requiring the user to figure it out by themselves, with clear advice on how to ensure that a user will produce a stable mesh. This will reduce some of the trial-and-error nature of the workflow, thus reducing the time-cost of said workflow, and reducing one of the causes of frustrations to FFEA users (as mentioned during the survey of FFEA 2.7.2, see sections §4.3.1 and §4.3.3). Alleviating these frustrations through clear communication from the software to the user will hopefully improve user retention.

The main improvement to MDCL-TMA's capabilities was the result of developing an interpolation scheme to smooth the surface of the mesh by applying the density threshold to the tetrahedra, rather than just the voxels. This was done by assigning each tetrahedron a density data value equal to the mean of the density values at each of its vertices. The algorithm then only accepts tetrahedra into the final mesh if its density data value is greater than the user-input density threshold. This creates a surface geometry more accurate to the shape of the original biomolecule, which means a more accurate distribution of the object's density and thus more accurate physics applied to the object in-simulation. The more accurate shape may also be of benefit to scientific communication through the visualizations that utilise these meshes (i.e. figures and recordings of simulations). This completes the major functionality of MDCL-TMA, making it ready to release for general use (subject to validity testing in the next chapter).

While improvements have been made to MDCL-TMA, it is uncertain yet if it is an improved method over the original meshing method using FFEAtools. Therefore, the next step is to compare the capabilities of both meshing methods to test the claims from this chapter that MDCL-TMA is the more usable and sustainable meshing algorithm. The validity of MDCL-TMA must also be tested to ensure the suitability of its produced meshes for use in FFEA simulations. Such testing will also highlight any limitations with the current implementation of MDCL-TMA, to see what future development work may be required to further improve software sustainability. These tests will be explored in the next chapter.

Chapter 7

Analysis of the marching diamond crystal lattice tetrahedral meshing algorithm

With the completion of the MDCL-TMA prototype, it now requires testing and analysing in comparison to the older FFEAtools method of meshing. This is to see if MDCL-TMA has made any improvements to the capability and stability of meshed objects for FFEA. This is also to see if improvements were made to the workflow for creating the meshes that improve the software's sustainable qualities via a better user-experience, and to ensure that the results they produce are valid. This chapter details this comparative analysis and makes note of any current limitations found within MDCL-TMA. In the general methodology (Figure 1.1) this chapter makes up the "Analyse new functionality" substage. The chapter is then concluded by considering MDCL-TMA against the second and third questions of the research scope (§1.1), to see how the sustainable qualities of research software can be improved, and if improving these qualities will improve the user experience, maintainability, and longevity of the software.

7.1 Comparative analysis

Comparison begins by meshing objects in both meshing algorithms. The initial data is taken from the Electron Microscopy Data Bank (EMDB) (European Molecular Biology Laboratory – European Bioinformatics Institute 2023). EMDB was chosen as it stores data in `.map` files,

which is recommended for meshing in FFEAtools (Cocking et al. 2021b), and is easily converted to the MRC format in ChimeraX for MDCL-TMA. EMDB was also chosen as it is a publicly-accessible database, meaning that all the data used here for analysis can be easily referred to and re-obtained to ensure the process is replicable. The electron cryo-microscopy maps are contoured to the same level (also known as the isolevel) in both meshing algorithms for each object. It is also attempted to coarse-grain the meshes to the same level too, however this is not always possible as errors can cause some meshes to be unable to be coarsened to a specific value (and all meshes will eventually have upper and lower limits of viable coarse-graining values).

The workflow for meshing these objects in FFEAtools is as follows:

1. Obtain 3D map volume data of the object from EMDB (`.map`).
2. With a `.map` file as an input, optionally use the FFEAtools command `emmaptosurf` to output a new processed `.map` file at a specified contour level, with floating objects culled and cavities filled that are smaller than user-defined thresholds.
3. With a `.map` file as an input, use the FFEAtools command `emmaptosurf` to output a `.obj` file at a specified contour level.
4. With an `.obj` as an input, use the FFEAtools command `surftocgsurf` to coarse-grain the mesh to a defined value, and output an `.stl` file.
5. The produced `.stl` file may contain self-intersecting faces which makes further meshing impossible. If this occurs, open the `.stl` file in a mesh-editing tool such as MeshLab (Cignoni et al. 2008), select the self-intersecting faces for deletion, and then export the fixed `.stl` file. If this does not occur, skip to step 6.
6. With an `.stl` file as input, use TetGen to convert the `.stl` file to the TetGen file format (for use in FFEA simulations) without modification of the input, using the `-Y` flag in the command line. This produces an output of `.ele`, `.face`, and `.node` files. This is the complete mesh.

The workflow for meshing these objects in MDCL-TMA is as follows:

1. Obtain 3D map volume data of the object from EMDB (`.map`).
2. Convert the `.map` file to an `.mrc` file. This can be done in ChimeraX by opening the `.map` file and saving as an `.mrc`.

3. Input the `.mrc` file to the Python MDCL-TMA tool, specify the object's contour level, and optionally input a desired coarse-graining value. This then outputs TetGen files (`.ele`, `.face`, and `.node`) of the complete mesh.

The FFEAtools and MDCL-TMA meshes are then compared using three different methods detailed in the following subsections.

7.1.1 Static mesh quality analysis

The first method of comparison is analysis of the quality statistics of the static meshes. The repository for MDCL-TMA contains a Python tool called `mesh_quality_analysis`. This tool inputs TetGen `.ele` and `.node` files (which both meshing methods produce as outputs), parses the coordinate data, and calculates and outputs mesh quality statistics, which are as follows:

1. The minimum and maximum edge lengths found in the tetrahedra (in the same units as the input mesh).

2. The ratio of minimum:maximum edge lengths. This is a value between 0-1, with 1 representing a mesh made entirely of regular tetrahedra.

3. The minimum, maximum, and average values of quality-fitness (QF) of the tetrahedra. This is a value between 0-1, with 1 representing the tetrahedron's shape fitting perfectly to that of a regular tetrahedron. It is calculated with

$$QF = 3\frac{\text{inradius}}{\text{circumradius}}. \tag{7.1}$$

This requires calculating the inradius and circumradius of the tetrahedron with

$$\text{inradius} = \frac{3V}{\widehat{A} + \widehat{B} + \widehat{C} + \widehat{D}} \tag{7.2}$$

and

$$\text{circumradius} = \frac{\sqrt{(aA + bB + cC)(aA + bB - cC)(aA - bB + cC)(-aA + bB + cC)}}{24V} \tag{7.3}$$

respectively, where V is the tetrahedron's volume, $\widehat{A}$, $\widehat{B}$, $\widehat{C}$, and $\widehat{D}$ are the areas of each face of the tetrahedron, a, b, and c are the lengths of three edges meeting at one vertex,

and A, B, and C are the lengths of the respective opposite edges.

4. A frequency distribution histogram of the tetrahedra quality-fitnesses.

Results

The results of the static mesh quality analysis, as well as details of the biological objects used as test data for this and subsequent analysis, are shown in Table 7.1. Objects were chosen either for their geometric shape or for their scientific interest, the RCSB's PDB Molecule of the Month articles being used as inspiration for the latter (RCSB 2023). Not every object was successfully meshed in both meshing algorithms, cases of which are denoted by "N/A" in the results, with the "Notes" column providing any reason for why this was so. Eleven objects total were meshed, and of those eleven, all could be meshed using MDCL-TMA, and two could be meshed using FFEAtools. The meshes produced are shown in Figures 7.1, 7.2, and 7.3.

While meshes generally become unstable in FFEA simulations at a coarse-graining value under 5Å (§5.2.1), MDCL-TMA was still able to mesh one object (hemoglobin) to a coarseness of 4Å and still obtain simulation results. No cases were found for any meshes that worked when coarsened to 3Å or lower, but that does not mean it is not possible (perhaps at very small simulation time-steps). This seems to justify warning a user about meshing objects with a coarseness under 5Å without blocking the user from continuing (§6.3).

The results recorded in Table 7.1 for the ratio of minimum to maximum edge lengths of mesh tetrahedra (plot onto a bar chart in Figure 7.4) show consistent results for MDCL-TMA. All MDCL-TMA meshes produced a ratio of 0.707 (to 3 d.p.). The regularity of the generation of tetrahedra from slicing voxels the same way each time means that regardless of volume, tetrahedra will have a consistent relationship in edge lengths. The constant value of 0.707 can be derived using Pythagoras's theorem with

$$a^2 + b^2 = c^2, \tag{7.4}$$

where c is the hypotenuse of a right-angle triangle, and a and b are the other two sides. For an isosceles right-angle triangle, this can be simplified further using

$$2a^2 = c^2. \tag{7.5}$$

Table 7.1: Static analysis results of meshed objects. "EMD ID" references the ID number of the input data obtained from the Electron Microscopy Data Bank, and "QF" refers to the quality-fitness value of the tetrahedra.

Object	**EMD ID**	**Isolevel**	**Mesh Generation Method**	**Coarseness (Å)**	**Number of Tetrahedra**	**Minimum Edge Length (Å)**	**Maximum Edge Length (Å)**	**Min:Max Edge Length**
Auxilin-bound clathrin coat	EMD-5120	0.14	FFEAtools	N/A	N/A	N/A	N/A	N/A
			MDCL-TMA	20	53686	20.287	28.690	0.707
CS-tubulin kinesin-13 microtubule complex	EMD-5565	0.015	FFEAtools	N/A	N/A	N/A	N/A	N/A
			MDCL-TMA	8	66494	8.169	11.553	0.707
Dengue virus	EMD-5520	0.4	FFEAtools	N/A	N/A	N/A	N/A	N/A
			MDCL-TMA	10	55165	10.191	14.412	0.707
Engineered nano-cage fusion protein	EMD-27812	0.614	FFEAtools	N/A	N/A	N/A	N/A	N/A
			MDCL-TMA	5	38257	5.077	7.180	0.707
GroEl	EMD-5043	1.15	FFEAtools	8	3397	6.090	27.656	0.220
			MDCL-TMA		8597	8.243	11.658	0.707
Hemoglobin	EMD-33189	0.0055	FFEAtools	N/A	N/A	N/A	N/A	N/A
			MDCL-TMA	4	940	4.153	5.873	0.707
HER2-trastuzumab-pertuzumab	EMD-20055	4.2	FFEAtools	N/A	N/A	N/A	N/A	N/A
			MDCL-TMA	5	2696	5.123	7.244	0.707
Homotrimeric poplar cellulose synthase	EMD-21820	0.17	FFEAtools	N/A	N/A	N/A	N/A	N/A
			MDCL-TMA	5	13219	5.100	7.212	0.707
Human cytoplasmic dynein-1	EMD-3705	0.06	FFEAtools	10	8685	0.230	563.961	4.07E+04
			MDCL-TMA		9837	10.154	14.360	0.707
SARS-CoV-2 omicron BA.2 spike protein	EMD-27523	0.0654	FFEAtools	N/A	N/A	N/A	N/A	N/A
			MDCL-TMA	8	2524	8.163	11.545	0.707
Yeast V-ATPase – SidK complex	EMD-8724	0.04	FFEAtools	N/A	N/A	N/A	N/A	N/A
			MDCL-TMA	5	30564	5.085	7.191	0.707

Object	**Minimum Tetrahedron QF**	**Maximum Tetrahedron QF**	**Average Tetrahedron QF**	**Notes**
Auxilin-bound clathrin coat	N/A 0.732	N/A 1.000	N/A 0.785	Could not create STL file due to coarsening at viable limit taking over 48hrs on Bede
CS-tubulin kinesin-13 microtubule complex	N/A 0.732	N/A 1.000	N/A 0.784	Could not be meshed due to face-sharing-edge error in surftocgsurf
Dengue virus	N/A 0.732	N/A 1.000	N/A 0.784	Could not create STL file due to coarsening at viable limit taking over 48hrs on Bede
Engineered nano-cage fusion protein	N/A 0.732	N/A 1.000	N/A 0.786	Could not create STL file due to coarsening at viable limit taking over 48hrs on Bede
GroEl	0.000 0.732	0.988 1.000	0.591 0.785	FFEA's model protein for tutorial
Hemoglobin	N/A 0.732	N/A 1.000	N/A 0.775	Could not be meshed due to face-sharing-edge error in surftocgsurf
HER2-trastuzumab-pertuzumab	N/A 0.732	N/A 1.000	N/A 0.781	Could not be meshed due to face-sharing-edge error in surftocgsurf
Homotrimeric poplar cellulose synthase	N/A 0.732	N/A 1.000	N/A 0.785	Could not be meshed due to face-sharing-edge error in surftocgsurf
Human cytoplasmic dynein-1	9.95E-07 0.732	0.986 1.000	0.465 0.786	
SARS-CoV-2 omicron BA.2 spike protein	N/A 0.732	N/A 1.000	N/A 0.783	Could not be meshed due to face-sharing-edge error in surftocgsurf
Yeast V-ATPase – SidK complex	N/A 0.732	N/A 1.000	N/A 0.784	Could not be meshed due to face-sharing-edge error in surftocgsurf

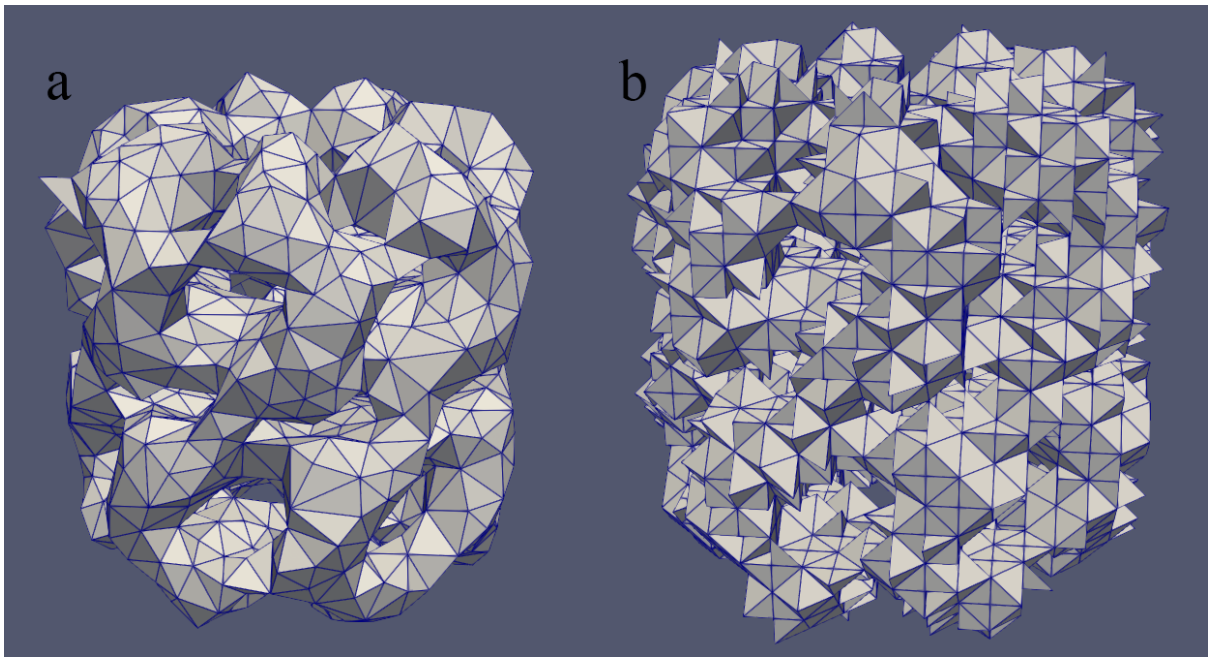

Figure 7.1: GroEL meshed using FFEAtools (a) and MDCL-TMA (b) viewed in ParaView.

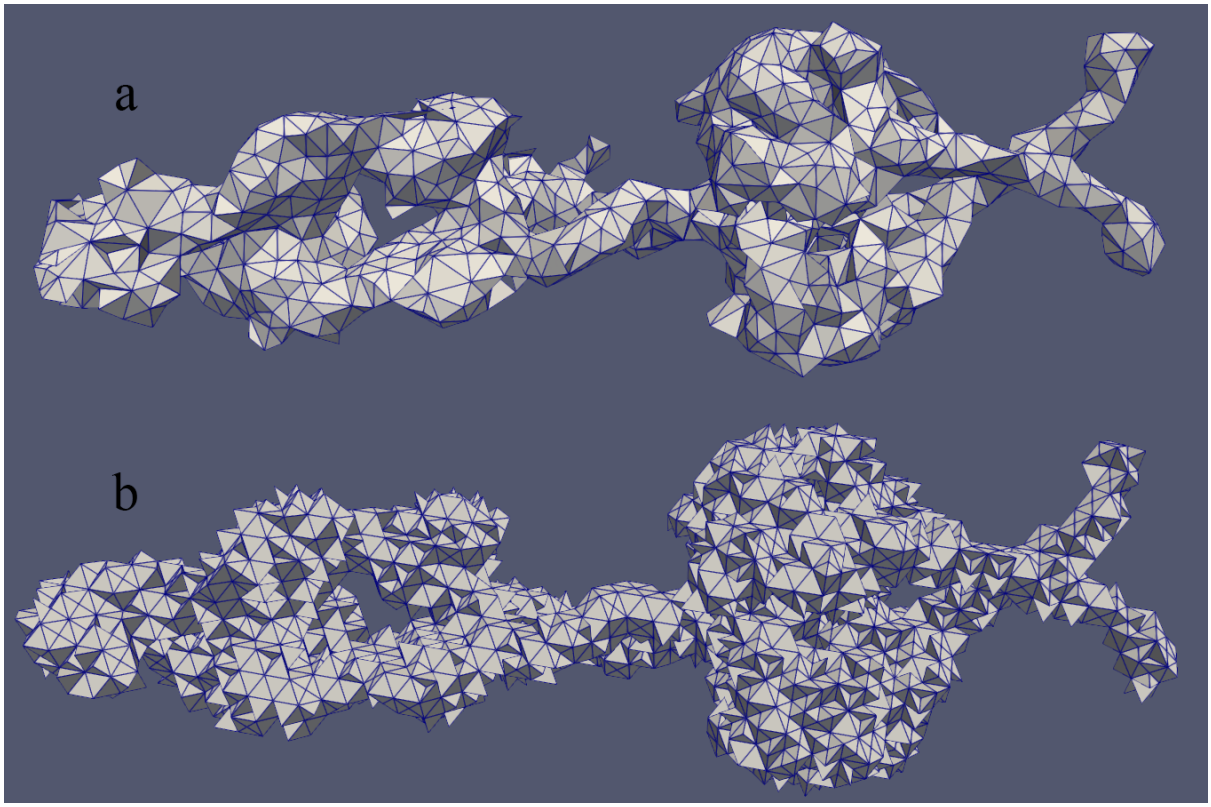

Figure 7.2: Human cytoplasmic dynein-1 meshed using FFEAtools (a) and MDCL-TMA (b) viewed in ParaView.

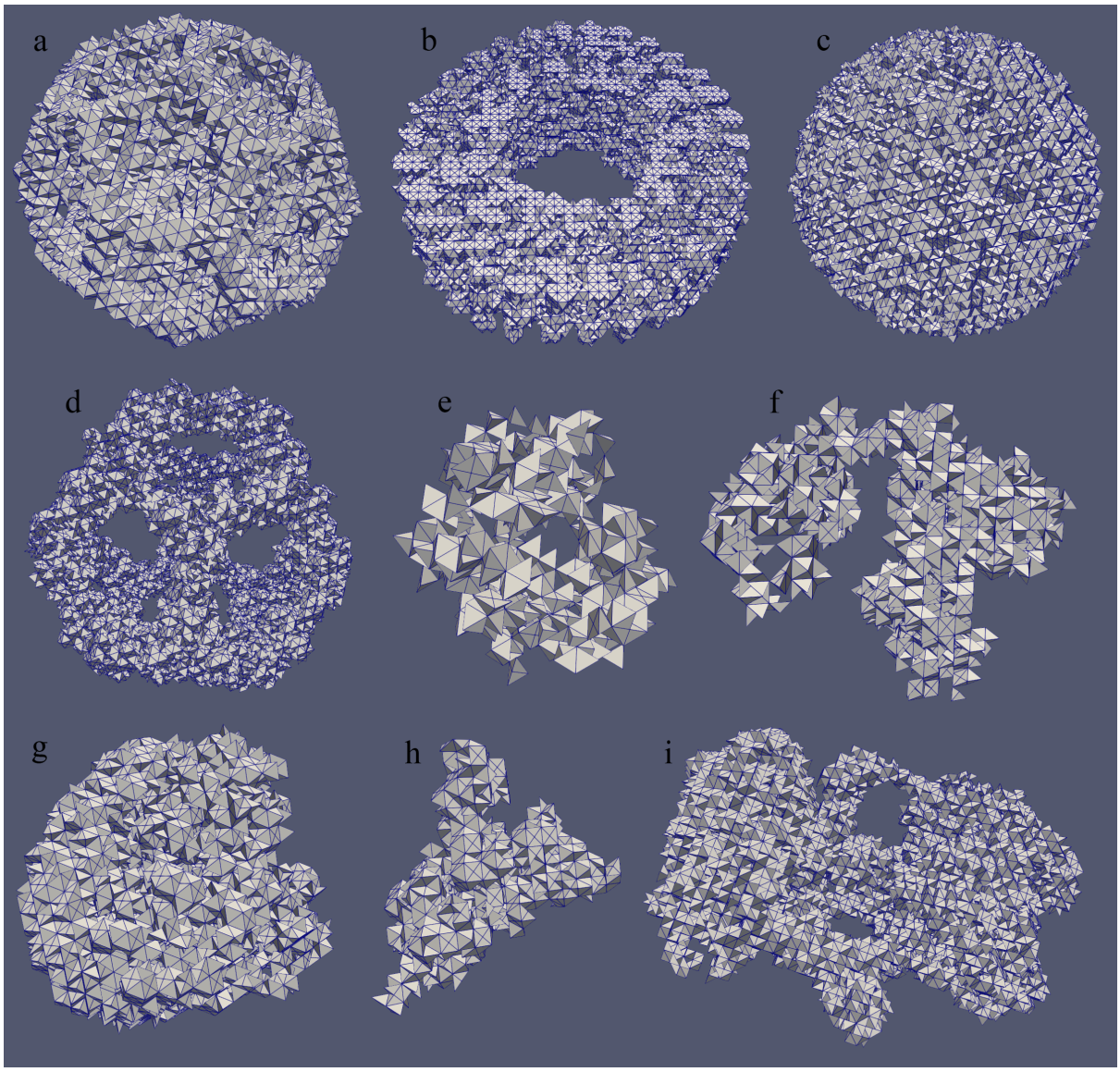

Figure 7.3: MDCL-TMA meshes viewed in ParaView. a) Auxilin-bound clathrin coat. b) CS-tubulin kinesin-13 microtubule complex. c) Dengue virus. d) Engineered nano-cage fusion protein. e) Hemoglobin. f) HER2-trastuzumab-pertuzumab. g) Homotrimeric poplar cellulose synthase. h) SARS-CoV-2 omicron BA.2 spike protein. i) Yeast V-ATPase – SidK complex.

MDCL-TMA meshes are made up of cubes that have a diagonal slice across each face to create the five tetrahedra (Figure 5.3). A square (the face of a cube) divided equally along its diagonal creates two isosceles right-angle triangles. This means that per voxel, there are twelve edges of length x and six edges of length y, with x and y having the same relationship as Pythagoras's theorem for an isosceles right-angle triangle (Equation 7.5), y representing the hypotenuses. If

it is assumed for an example that $x = 1$, then this gives the value of y as follows:

$$2 \times 1^2 = y^2$$
$$2 = y^2$$
$$\sqrt{2} = y$$

This means that the ratio of the length of x to the length of y is $\sqrt{2} : 1$. If this ratio is input into the total number of edge lengths per voxel, the following is derived:

$$12x : 6y$$
$$12 \times 1 : 6 \times \sqrt{2}$$
$$12 : 6\sqrt{2}$$
$$1 : \frac{1}{\sqrt{2}}$$
$$1 : 0.707$$

FFEAtools does not have the same consistency, as the irregular meshing method it uses means that the relationship of edge lengths will differ a lot mesh-to-mesh. The cases recorded found that FFEAtools produced lower edge length ratios than MDCL-TMA.

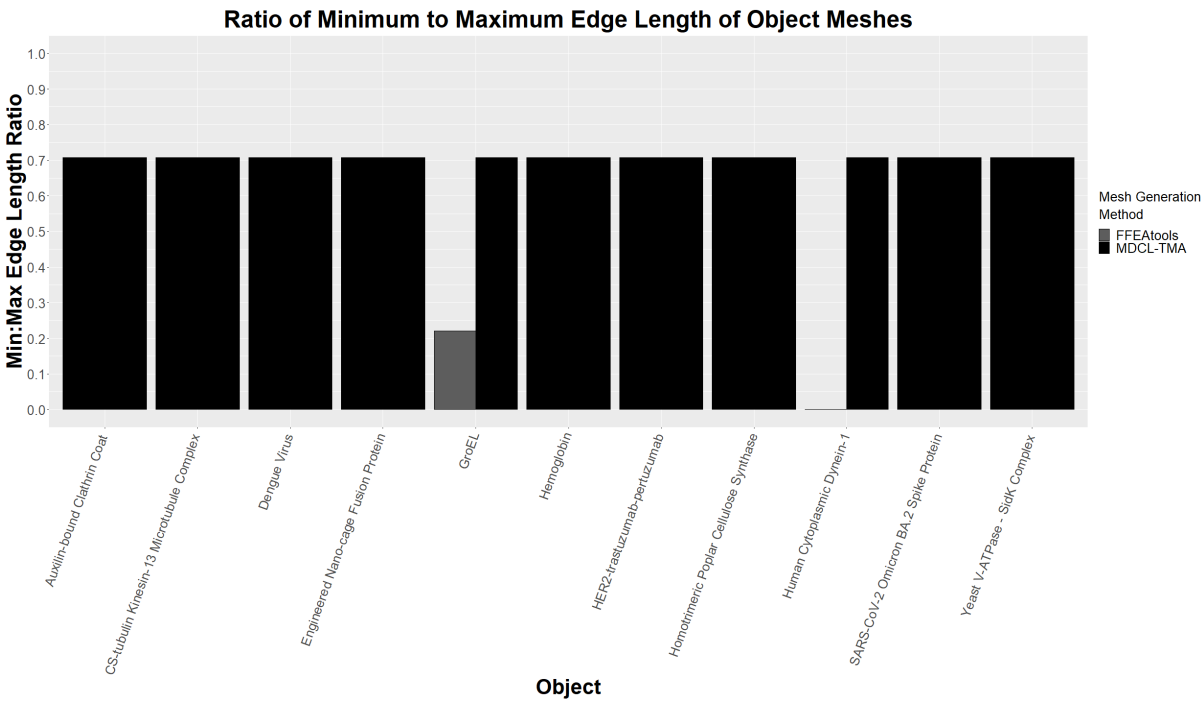

Figure 7.4: Ratio of minimum to maximum edge lengths of meshed objects

Quality-fitness results (Figure 7.5) show similar consistencies, with minimum, maximum, and average QF values being approximately the same for all MDCL-TMA meshes. Minor discrepancies in the average QF values between MDCL-TMA meshes are mainly caused by the interpolation scheme that smooths the surface of the mesh by removing some tetrahedra from fully-packed voxels, which will slightly alter the ratio of trirectangular tetrahedra to regular tetrahedra (Figure 5.3) in the mesh. Some further smaller discrepancies may be the result of finite precision arithmetic errors due to using floating point numbers in the code to store geometric data.

Again, the regularity of the mesh generation method of MDCL-TMA is showing consistent qualities regardless of object size or shape. FFEAtools however produces wider ranges of tetrahedra quality-fitness, including very low minimum values (suggesting sliver elements). This can be seen more easily in the frequency histograms of QF values of mesh tetrahedra (Figure 7.6). MDCL-TMA meshes have a consistent 4:1 ratio of tetrahedra with QF values of approximately 0.732 and 1 respectively, and FFEAtools meshes have more distributed QF values.

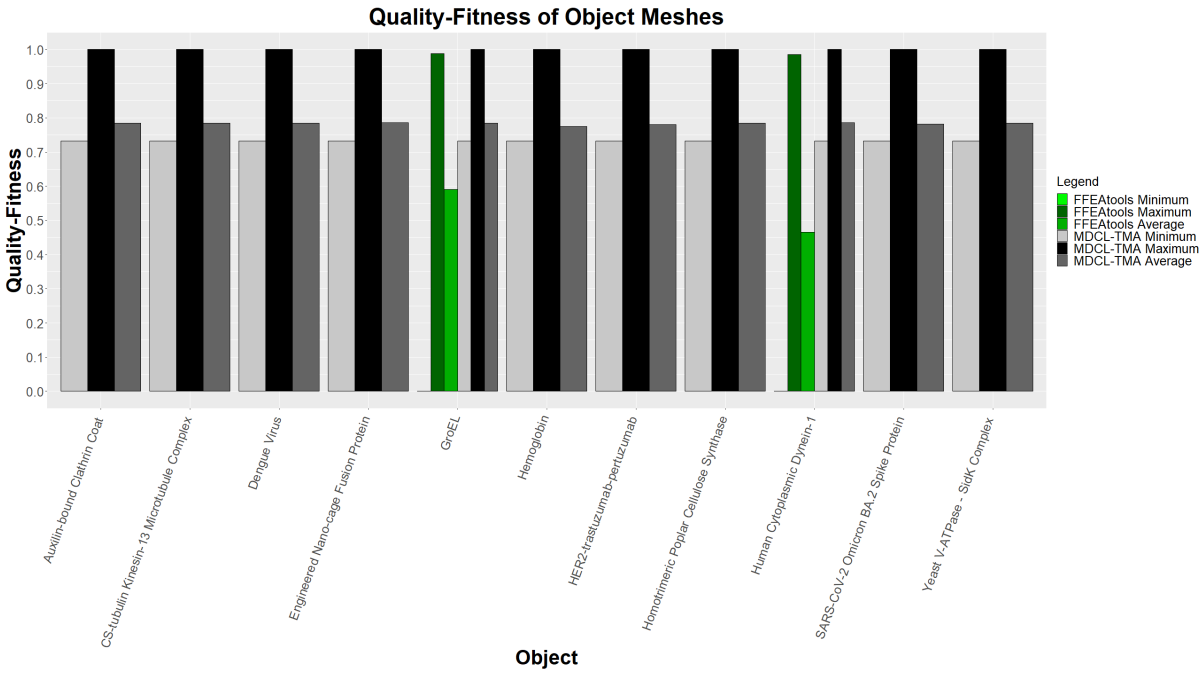

Figure 7.5: Quality-fitness of meshed objects

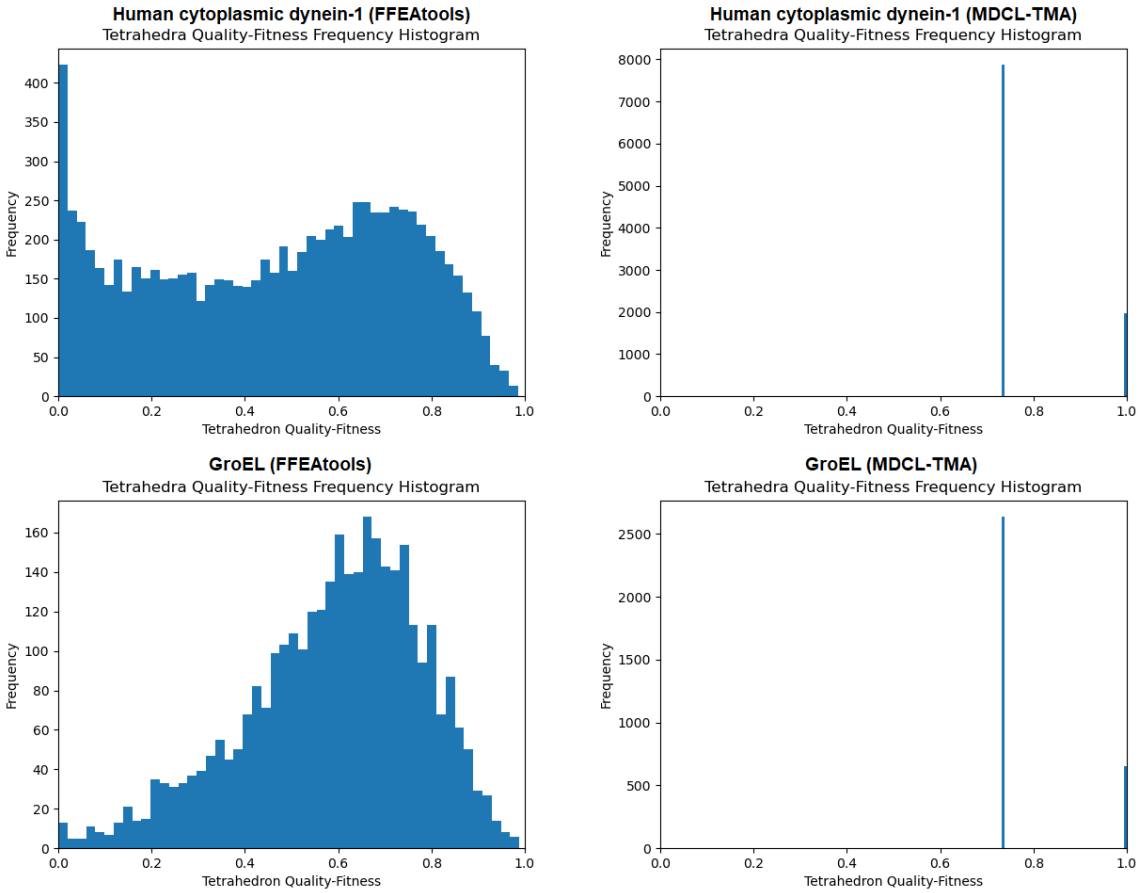

Figure 7.6: Comparison of frequency histograms of mesh quality-fitness. All frequency histograms for MDCL-TMA meshes made from the rest of the dataset of biomolecules not shown here have the same distribution of quality-fitness.

The 4:1 ratio of quality-fitness values appears due to how each voxel is sliced to create five tetrahedra (Figure 5.3); four trirectangular tetrahedra (Figure 7.7) of equal size, and one inner regular tetrahedron.

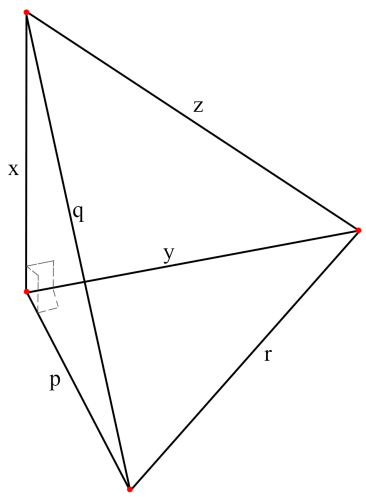

Figure 7.7: Example of a trirectangular tetrahedron

To understand why MDCL-TMA produces meshes with an average quality-fitness value of approximately 0.785 (to 3 d.p.), the quality fitness of a trirectangular tetrahedron must first be derived (a regular tetrahedron simply has a quality-fitness of 1 as quality-fitness is a comparison against regular tetrahedra). Figure 7.7 is used as a reference for all following calculations in this section. It is assumed that edges x, y, and p are of length 1, and therefore due to Pythagoras's theorem of an isosceles triangle (Equation 7.5) edges z, q, and r are of length $\sqrt{2}$.

First, the area of each face must be calculated. The area of a right-angle triangle is calculated using

$$\text{Area} = \frac{ab}{2}, \tag{7.6}$$

and from this these areas can be calculated:

$$\begin{aligned}
\widehat{xyz} &= \frac{xy}{2} = \frac{1 \times 1}{2} = 0.5 \\
\widehat{xpq} &= \frac{xp}{2} = \frac{1 \times 1}{2} = 0.5 \\
\widehat{ypr} &= \frac{yp}{2} = \frac{1 \times 1}{2} = 0.5.
\end{aligned}$$

$\widehat{zqr}$ is an equilateral triangle, the area of which is calculated with

$$\text{Area} = \frac{\sqrt{3}}{4}a^2, \tag{7.7}$$

which gives the following:

$$\widehat{zqr} = \frac{\sqrt{3}}{4} \times \sqrt{2}^2 = \frac{\sqrt{3}}{2}.$$

Next, the volume (V) is needed, which for a trirectangular tetrahedron is calculated with

$$V = \frac{abc}{6}, \tag{7.8}$$

where a, b, and c are the lengths of the edges that originate at the trirectangular tetrahedron's

right-angle vertex. This gives the following:

$$V = \frac{xyp}{6} = \frac{1 \times 1 \times 1}{6} = \frac{1}{6}.$$

With the face areas and the volume, the tetrahedron's inradius can now be calculated with Equation 7.2. This then gives the following:

$$\text{inradius} = \frac{3V}{\widehat{xyz} + \widehat{xpq} + \widehat{ypr} + \widehat{zqr}} = \frac{3 \times \frac{1}{6}}{0.5 + 0.5 + 0.5 + \frac{\sqrt{3}}{2}} = \frac{3 - \sqrt{3}}{6}.$$

Next, the circumradius of the tetrahedron needs to be calculated with Equation 7.3. This gives the following:

$$\text{circumradius} = \frac{\sqrt{(xr + yq + zp)(xr + yq - zp)(xr - yq + zp)(-xr + yq + zp)}}{24V}$$

$$\text{circumradius} = \frac{\sqrt{((1\times\sqrt{2})+(1\times\sqrt{2})+(\sqrt{2}\times 1))((1\times\sqrt{2})+(1\times\sqrt{2})-(\sqrt{2}\times 1))((1\times\sqrt{2})-(1\times\sqrt{2})+(\sqrt{2}\times 1))(-(1\times\sqrt{2})+(1\times\sqrt{2})+(\sqrt{2}\times 1))}}{24\times\frac{1}{6}}$$

$$\text{circumradius} = \frac{\sqrt{3}}{2}.$$

Finally, now that the inradius and circumradius have both been obtained, the quality-fitness of the trirectangular tetrahedron can be calculated using Equation 7.1 to get the following to match the found ratio value:

$$QF = 3\frac{\frac{3-\sqrt{3}}{6}}{\frac{\sqrt{3}}{2}} = -1 + \sqrt{3} \approx 0.732.$$

Now that the quality-fitness of a trirectangular tetrahedron is known (which matches the found value of approximately 0.732 for the lower bound of MDCL-TMA's QF), the average quality-fitness across all tetrahedra per voxel can be obtained with

$$\bar{QF} = \frac{4(-1 + \sqrt{3}) + 1}{5} = \frac{-3 + 4\sqrt{3}}{5} \approx 0.786.$$

Some variation in the edge length ratio or average quality-fitness may occur due to the inter-

polation that smooths the surface by removing some tetrahedra in some voxels at the mesh boundaries, making the ratio of four trirectangular tetrahedra to one regular tetrahedron inexact. This variation may be more overt in meshes made of a small number of voxels, where there are less internal fully-packed voxels of five tetrahedra. The recorded variations of these values in the objects meshed, as seen in Figures 7.4 & 7.5, is quite minor however.

7.1.2 Simulation performance analysis

The second comparison method is to run the meshes in simple but lengthy FFEA simulations using the FFEA executable. The simulations are left to run on N8 Bede (H. 2021) for 48hrs (the maximum job-time allowed on that system) in order to record the number of steps, the wall clock time, and the simulation time reached before the simulation terminates, and what was the stop condition that caused termination (i.e. if the time limit was reached or a fatal error occurred first). This essentially uses the same methodology as the soak tests done in §3.3. The same simulation parameters are used for both mesh generation methods. This allows for comparing the efficiency and stability of the meshes to determine if one method produces meshes that perform better in simulations than the other.

Results

Simulation performance results of the meshed objects are shown in Table 7.2. All FFEAtools meshes performed poorly, only able to run relatively few steps at the same time-step as MDCL-TMA before element inversion occurs (including groEL which is the model protein used for the FFEA tutorial and was generated for this experiment using the same method). In most cases MDCL-TMA meshes reached the time limit stop condition, but MDCL-TMA is not immune to producing simulation results that end in an element inversion. The nano-cage, HER2, and yeast objects all terminated prematurely due to element inversion, however they were still able to get to a further wall clock time than the FFEAtools meshes (Figure 7.8), and could perhaps perform better still at a smaller simulation time-step or a higher coarseness. Hemoglobin is of particular note for managing to reach the time limit stop condition, despite being coarsened to 4Å, under the general stable limit. This suggests MDCL-TMA can produce meshes that are more stable than FFEAtools meshes when close to the lower-limits of coarseness values that FFEA can simulate with.

Table 7.2: Simulation performance analysis results of meshed objects when ran in 48-hour FFEA simulations.

Object	Mesh Generation Method	Time-Step (s)	Steps Reached	Average Steps	Wall Clock Time (Hrs)	Average WCT (Hrs)	Simulation Time (ns)	Average ST (ns)	Stop Condition
Auxilin-bound clathrin coat	MDCL-TMA	e-13	1315000	1294333.$\dot{3}$	47.79	47.821	131	129	Time Limit
			1277000		47.712		127		Time Limit
			1291000		47.961		129		Time Limit
CS-tubulin kinesin-13 microtubule complex	MDCL-TMA	e-14	964000	966333.$\dot{3}$	47.771	47.657	9.6	9.6	Time Limit
			976000		47.688		9.7		Time Limit
			959000		47.513		9.5		Time Limit
Dengue virus	MDCL-TMA	e-13	1145000	1119000	47.765	47.741	114	111.$\dot{3}$	Time Limit
			1114000		47.816		111		Time Limit
			1098000		47.644		109		Time Limit
Engineered nano-cage fusion protein	MDCL-TMA	e-14	22649	24922.$\dot{6}$	0.775	0.860	0.2	0.2	Element Inversion
			44608		1.547		0.4		Element Inversion
			7511		0.257		N/A		Element Inversion
GroEL	FFEAtools	e-13	19	44	6.26E-05	1.27E-04	N/A	N/A	Element Inversion
			25		7.53E-05		N/A		Element Inversion
			88		2.42E-04		N/A		Element Inversion
	MDCL-TMA		9208000	9218000	47.964	47.976	920	921.$\dot{3}$	Time Limit
			9260000		48		926		Time Limit
			9186000		47.965		918		Time Limit
Hemoglobin	MDCL-TMA	e-14	59499000	59606666.$\dot{6}$	47.995	47.998	594	595.7	Time Limit
			59385000		48		593.8		Time Limit
			59936000		47.998		599.3		Time Limit
HER2-trastuzumab-pertuzumab	MDCL-TMA	e-13	3484767	3143606.$\dot{3}$	6.982	6.304	348	313.$\dot{6}$	Element Inversion
			3568782		7.166		356		Element Inversion
			2377270		4.764		237		Element Inversion
Homotrimeric poplar cellulose synthase	MDCL-TMA	e-14	4893000	4899666.$\dot{6}$	47.968	47.97	48.9	48.9$\dot{6}$	Time Limit
			4902000		47.977		49		Time Limit
			4904000		47.965		49		Time Limit
Human cytoplasmic dynein-1	FFEAtools	e-13	1	2.$\dot{3}$	2.03E-05	2.72E-05	N/A	N/A	Element Inversion
			2		2.40E-05		N/A		Element Inversion
			4		3.72E-05		N/A		Element Inversion
	MDCL-TMA		8339000	8317666.$\dot{6}$	47.946	47.956	833	831	Time Limit
			8455000		47.97		845		Time Limit
			8159000		47.951		815		Time Limit
SARS-CoV-2 omicron BA.2 spike protein	MDCL-TMA	e-13	28673000	28823000	47.997	47.999	2867	2882	Time Limit
			28972000		48.003		2897		Time Limit
			28824000		47.998		2882		Time Limit
Yeast V-ATPase – SidK complex	MDCL-TMA	e-14	16348	53654.$\dot{6}$	0.420	1.385	0.1	0.5	Element Inversion
			64062		1.659		0.6		Element Inversion
			80554		2.077		0.8		Element Inversion

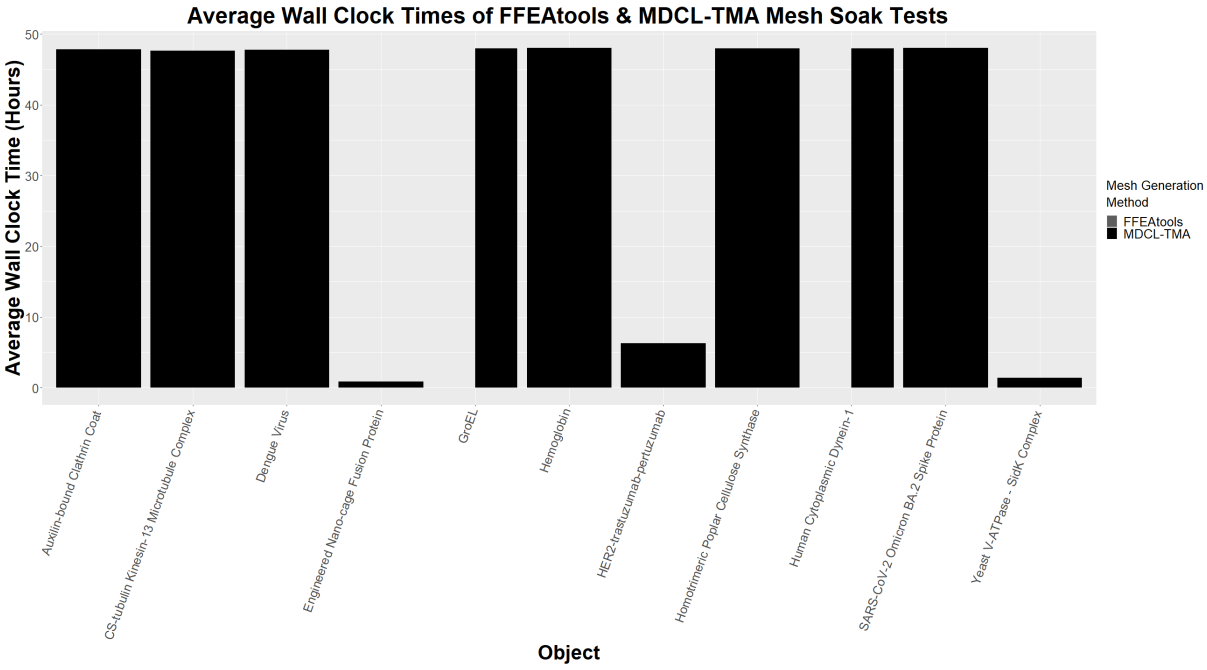

Figure 7.8: Soak test wall clock times of meshed objects. Note that results for FFEAtools meshes of groEL and dynein-1 are greater than zero but are too small to appear on the plot.

7.1.3 Simulation result analysis

The final comparison method also runs meshes in FFEA simulations, but very short ones of 1000 steps and a small time-step of 1e-15s (i.e. a total simulation time of 1ns), so that simulations with roughly-equal dynamics are produced. As FFEA introduces stochastic thermal fluctuations to a simulation, even with the exact same mesh and parameters there will always be minor differences in simulation output (hence why only simulations with qualitatively similar dynamics were obtained). These simulations are conducted three times per mesh to get average values across multiple simulations and to account for potential outliers. They are run so that the results themselves can be compared in order to check the validity of simulations using meshes generated with MDCL-TMA.

FFEA simulations output human-readable trajectory files which include text-based tables of measurements of the centroid position, the strain energy, and the root mean square deviation (RMSD) over time. From each run of these FFEA simulations the average RMSD measured at each checkpoint is calculated, and then an average value across the repeated runs is calculated.

RMSD is used in FFEA as a measurement of the dynamics of a simulation and how much an object fluctuates over time (Solernou et al. 2018). Object elasticity, for example, is correlated to the RMSD, as stiffer objects are less flexible and thus cannot fluctuate as much. The same

object can measure different RMSD values when the material parameters of the object change, hence why material parameters have been kept constant across all meshes for these comparative experiments.

Results

The results of the FFEA simulations of meshed objects are recorded in Table 7.3. The FFEA `.fm` output files also contain FFEA's measurements of the objects' rest volume and object mass, which were also recorded in this table. The objects' rest surface area was also of interest, so this was measured using ParaView and recorded in the table. Later in this section, another groEL MDCL-TMA mesh was required to test a comparison of the meshing algorithms at a similar tetrahedron count, which in this case required a coarse-graining value of 11Å. This mesh's 1ns FFEA simulation results were added to this table.

The FFEAtools mesh of dynein-1 was unable to complete the 1ns simulation during any run, hence why they have a recorded wall clock time of "N/A". These runs did manage to complete a number of steps before ending in element inversion errors, which meant that some data on RMSD and strain energy could be recorded, but this hinders the ability to compare FFEAtools and MDCL-TMA meshing. As it had less time to fluctuate, which impacts its RMSD measurements, this made its RMSD values an invalid comparison to the MDCL-TMA runs, but they have been included for complete data as it was measurable.

The use of FFEAtools or MDCL-TMA is recorded here to have a variable effect on the rest volume or mass of the object. For groEL, the FFEAtools mesh had a smaller rest volume and mass than the MDCL-TMA mesh at the same coarse-graining value, but for dynein-1 the FFEAtools mesh had a *larger* rest volume and mass. The reason for this is apparent when MDCL-TMA and FFEAtools meshes are overlaid on top of each other (see Figure 7.9). The surface geometry of an MDCL-TMA mesh is "spiky", and these "spikes" can either recede into or protrude from the surface geometry created by FFEAtools when overlaid, i.e. the parts of the MDCL-TMA mesh across the surface geometry can either be concave or convex respectively compared to the surface geometry of FFEAtools meshes. If more of the MDCL-TMA mesh is concave compared to the FFEAtools mesh, it will have a smaller volume and also therefore likely a smaller mass, and vice versa.

Table 7.3: Results of 1ns FFEA simulation measurements of meshed objects.

Object	Mesh Generation Method	Object Rest Surface Area (Å²)	Object Rest Volume (Å³)	Object Mass (kg)	Wall Clock Time Taken (s)	Average WCT (s)	Average RMSD	Average ARMSD
Auxilin-bound clathrin coat	MDCL-TMA	7.30E+06	8.95E+07	1.34E-19	137.078 114.066 109.332	120.159	3.14E-11 3.14E-11 3.12E-11	3.13E-11
CS-tubulin kinesin-13 microtubule complex	MDCL-TMA	2.06E+06	7.22E+06	1.08E-20	198.605 222.587 204.705	208.632	5.35E-11 5.39E-11 5.37E-11	5.37E-11
Dengue Virus	MDCL-TMA	3.17E+06	1.16E+07	1.75E-20	156.799 160.945 162.831	160.192	5.07E-11 5.01E-11 5.06E-11	5.05E-11
Engineered nano-cage fusion protein	MDCL-TMA	6.66E+05	1.00E+06	1.50E-21	125.769 113.503 116.315	118.529	7.47E-11 7.50E-11 7.49E-11	7.49E-11
GroEL	FFEAtools (8Å)	1.54E+05	9.00E+05	1.35E-21	6.164 8.846 8.857	7.956	3.68E-11 3.89E-11 3.82E-11	3.79E-11
	MDCL-TMA (8Å)	2.33E+05	9.60E+05	1.44E-21	15.303 18.087 19.242	17.544	5.03E-11 4.97E-11 5.00E-11	5.00E-11
	MDCL-TMA (11Å)	2.35E+05	9.46E+05	1.42E-21	5.658 5.965 6.339	5.988	4.75E-11 4.70E-11 4.72E-11	4.73E-11
Hemoglobin	MDCL-TMA	1.86E+04	1.30E+04	1.95E-23	2.326 1.869 1.748	1.981	1.11E-10 1.11E-10 1.07E-10	1.10E-10
HER2-trastuzumab-pertuzumab	MDCL-TMA	6.47E+04	7.14E+04	1.07E-22	6.392 7.074 6.182	6.549	8.34E-11 8.27E-11 8.31E-11	8.31E-11
Homotrimeric poplar cellulose synthase	MDCL-TMA	2.09E+05	3.50E+05	5.25E-22	30.545 28.215 33.498	30.752	7.06E-11 7.12E-11 6.95E-11	7.04E-11
Human cytoplasmic dynein-1	FFEAtools	2.87E+05	7.72E+06	1.16E-20	N/A N/A N/A	N/A	9.05E-12 9.39E-12 6.45E-12	8.30E-12
	MDCL-TMA	3.67E+05	2.06E+06	3.09E-21	18.073 17.827 20.498	18.799	4.11E-11 4.15E-11 4.15E-11	4.14E-11
SARS-CoV-2 omicron BA.2 spike protein	MDCL-TMA	1.16E+05	2.72E+05	4.08E-22	4.034 4.305 4.076	4.138	5.78E-11 5.60E-11 5.76E-11	5.71E-11
Yeast V-ATPase – SidK complex	MDCL-TMA	5.15E+05	8.00E+05	1.20E-21	104.767 97.120 84.693	95.527	7.14E-11 7.15E-11 7.18E-11	7.16E-11

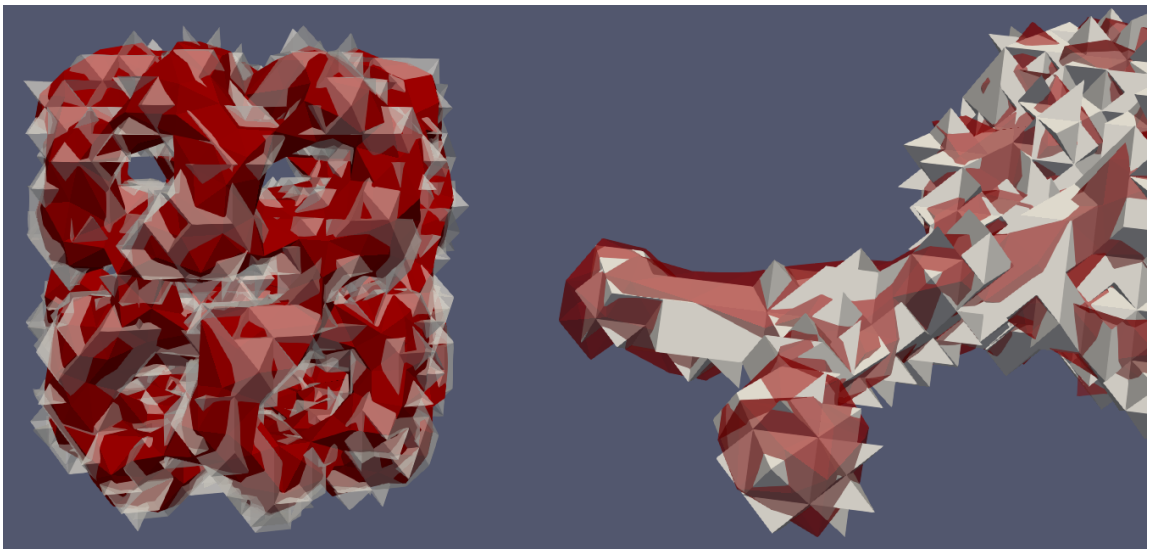

Figure 7.9: Comparison of overlaid FFEAtools meshes (red) and MDCL-TMA meshes (grey) for groEL (left) and the stalks of human cytoplasmic dynein-1 (right), viewed in ParaView. The surface of the MDCL-TMA meshes can be concave or convex at different locations in comparison to the FFEAtools meshes. Depending on if more of the MDCL-TMA surface is concave or convex, it will have respectively a smaller or larger volume and mass compared to the FFEAtools mesh. In this example, the groEL MDCL-TMA mesh has many protrusions out of the FFEAtools mesh, and thus the MDCL-TMA mesh has a larger volume and mass. The dynein-1 MDCL-TMA mesh however has much of it enveloped inside the FFEAtools mesh, and thus the MDCL-TMA mesh has a smaller volume and mass.

FFEAtools meshes had smaller rest surface areas than MDCL-TMA meshes, which is to be expected as FFEAtools tries to make a surface mesh form-fitted to the object, but more comparable objects are required to see if this result is reliable. The MDCL-TMA meshes record taking longer to compute in simulations than FFEAtools meshes, likely a result of MDCL-TMA meshes having larger tetrahedron counts at the same coarse-graining value, but again more comparable objects are needed to see if this is a consistent relationship (it would also be useful to compare wall clock times for objects meshed in both algorithms to similar tetrahedron counts, as the result for groEL meshed in MDCL-TMA to 11Å suggests it potentially may be faster).

Figures 7.10 and 7.11 show plots of the strain energy over time during the 1ns FFEA simulations. All plots follow a similar trend of a rising strain energy with the rate of change lowering over time. Most runs of the same mesh are very similar, the most deviation between runs was seen in hemoglobin, which was meshed with a coarse-graining value of a usually-unstable 4Å. For the two objects meshed in both meshing algorithms, MDCL-TMA measures higher strain energy over time, although to a similar magnitude. In these two experiments, the strain energy of MDCL-TMA meshes to the same coarse-graining value appears approximately double that of the FFEAtools meshes' strain energy.

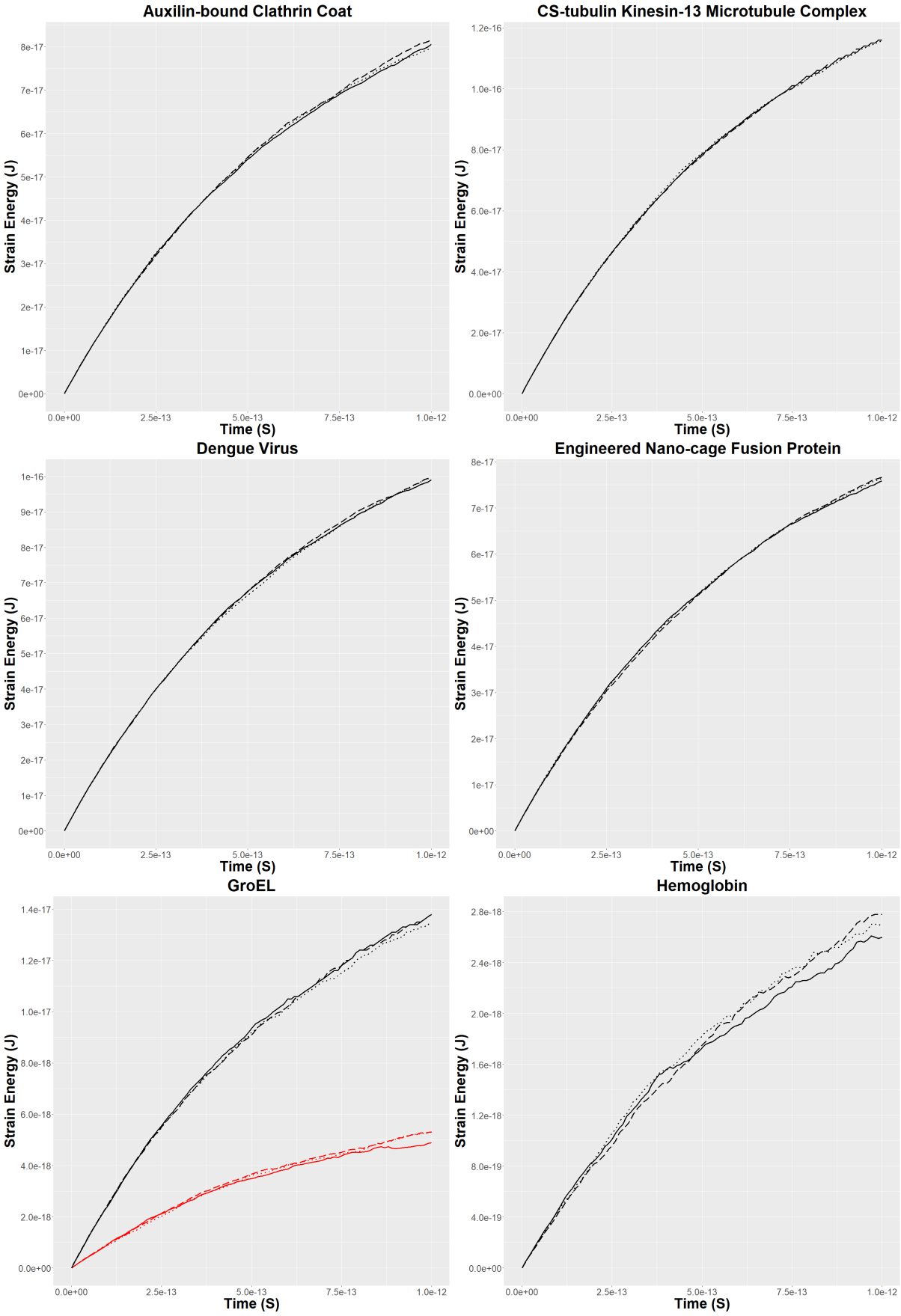

Figure 7.10: Strain energy over time during 1ns FFEA simulations for meshed objects

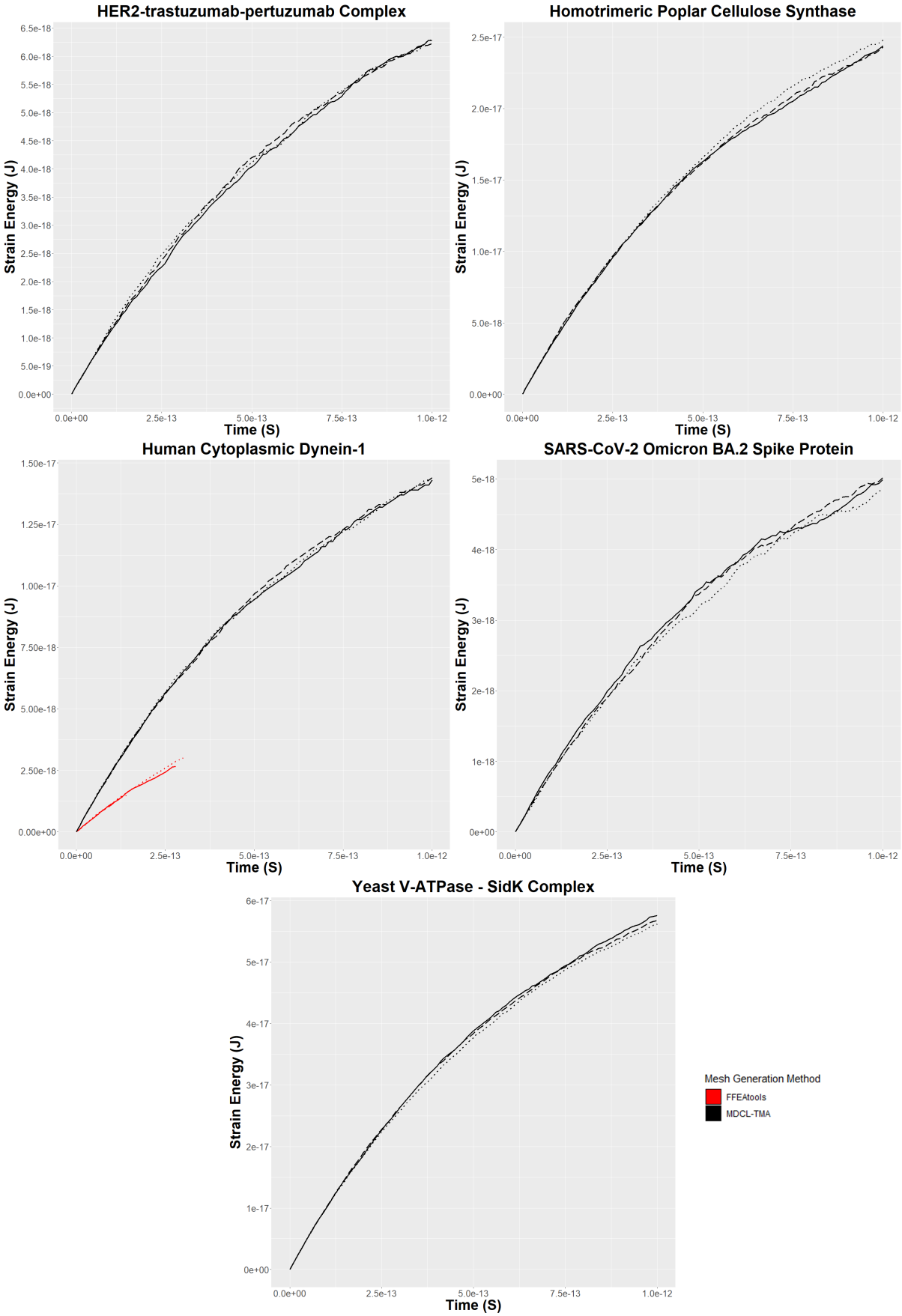

Figure 7.11: Strain energy over time during 1ns FFEA simulations for meshed objects (cont.)

Potential explanations for the difference between strain energy in FFEAtools and MDCL-TMA meshes are the difference in the amount of tetrahedra created for each mesh or the surface area. When groEL was meshed to the same coarse-graining value of 8Å, this produced an FFEAtools mesh containing 3397 tetrahedra, and an MDCL-TMA mesh containing 8597 tetrahedra. If groEL is re-meshed in MDCL-TMA to a coarse-graining value of 11Å, this produces a mesh containing 3288 tetrahedra (information for this mesh was added to Table 7.3), much closer to the amount of tetrahedra in the FFEAtools 8Å mesh. The surface area has remained roughly constant to the previous MDCL-TMA mesh at 2.35E+05Å^2. If the simulation performance experiments are re-done for the new mesh and its strain energy over time is plotted against the FFEAtools mesh, it now records much closer values, which can be seen in Figure 7.12, suggesting that the tetrahedron count is proportionally correlated to strain energy.

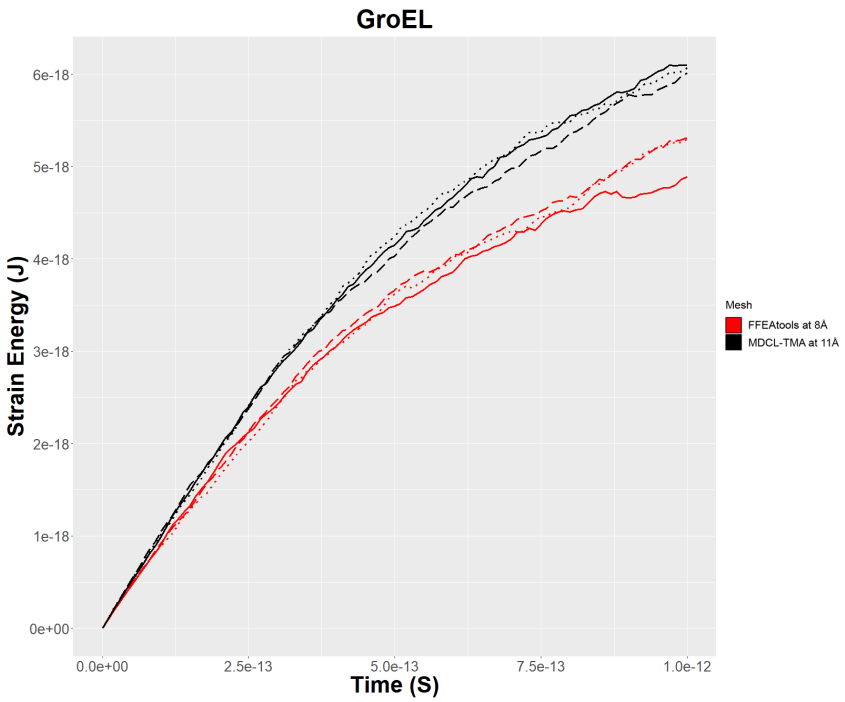

Figure 7.12: Strain energy over time during 1ns FFEA simulations for groEL meshed with FFEAtools to 8Å and with MDCL-TMA to 11Å.

A quick test was performed on FFEAtools meshes to see if they also had a correlation between tetrahedron count and strain energy. This was done by meshing groEL to 5Å, 8Å, and 12Å, which have tetrahedron counts of 10143, 3397, and 1208 respectively. The strain energy over time for each mesh was plot onto the same graph, shown in Figure 7.13. There is still a proportional correlation between tetrahedron count and strain energy, meaning that this is not a new property of MDCL-TMA meshes.

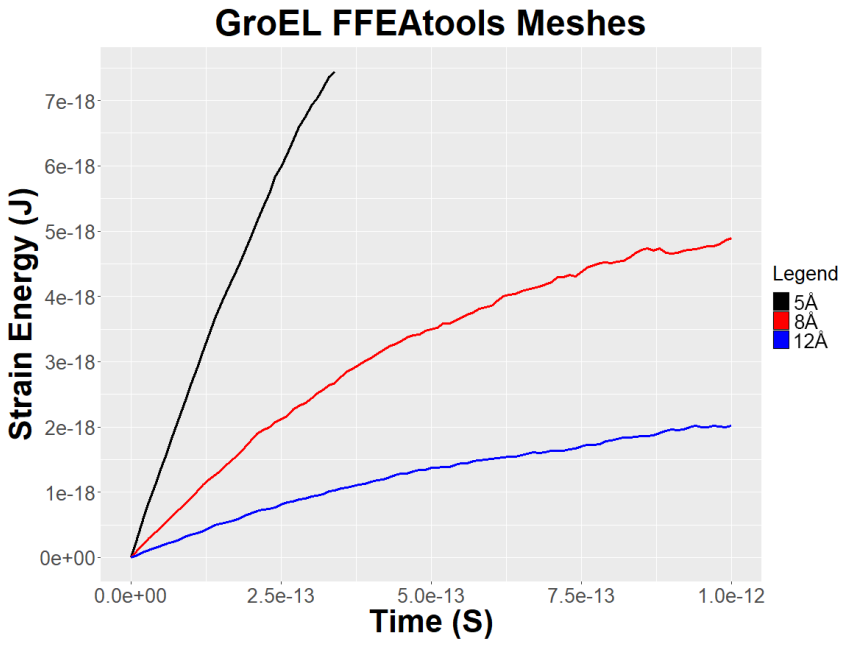

Figure 7.13: Strain energy over time during 1ns FFEA simulations for groEL meshed with FFEAtools to 5Å, 8Å, and 12Å.

In order to see if there is a relationship between strain energy and the movement of the object, a plot was also created that shows meshes of groEL's centroids' deviation from its average position over time (Figures 7.14 and 7.15). Due to visual complexity these plots were split by runs in the second figure for clarity, there is nothing to connect specific runs for each mesh other than the order they were performed.

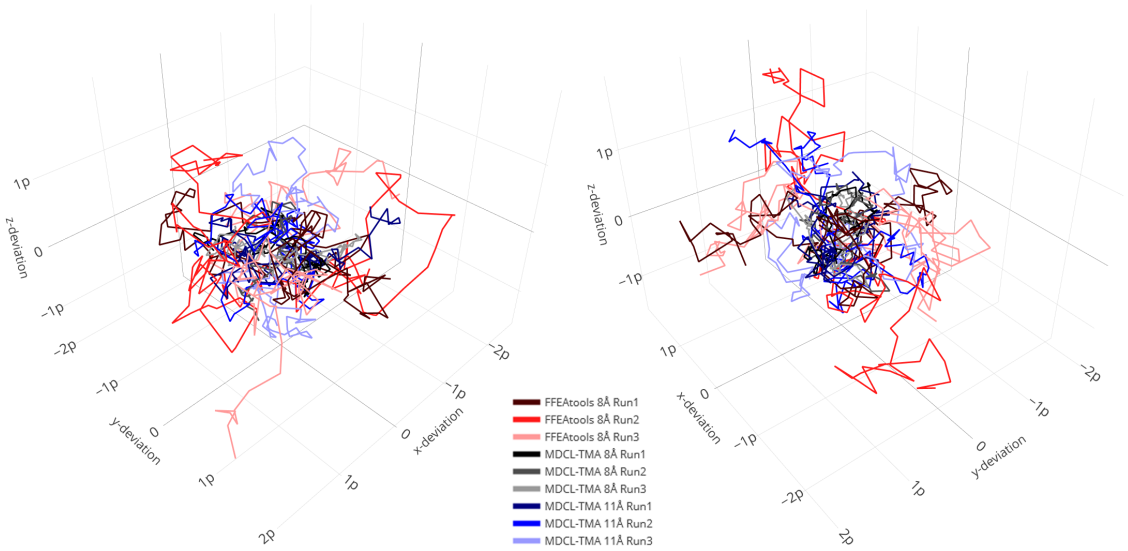

Figure 7.14: Centroid deviation over time during 1ns FFEA simulations for groEL, viewed at two angles. The FFEAtools mesh (red lines) has the largest deviations in the position of its centroid over time, and the 8Å MDCL-TMA mesh (black lines) has the smallest centroid deviations.

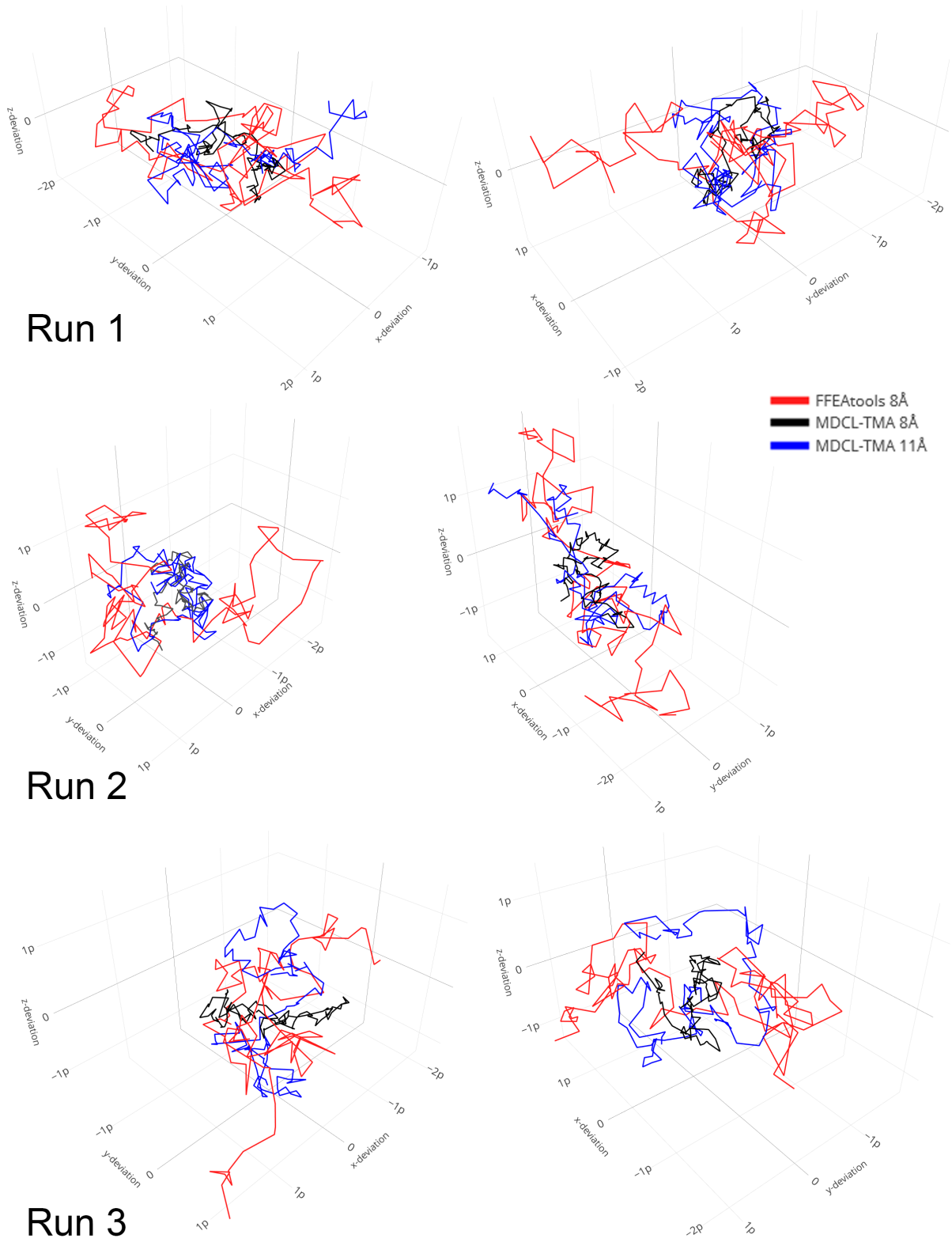

Figure 7.15: Centroid deviation over time during 1ns FFEA simulations for groEL split into runs (separated by row), each viewed at two angles.

Compared to the FFEAtools mesh, the MDCL-TMA mesh at the same coarse-graining value of 8Å has a much smaller centroid deviation, meaning it moves a lot less. The MDCL-TMA

mesh at 11Å with a similar tetrahedra count to the FFEAtools mesh has a closer magnitude of centroid deviation. This is similar to how strain energy plots became closer in recorded values when using a mesh of a similar tetrahedron count, further suggesting the proportional correlation between tetrahedron count and strain energy, and now as well an inverse correlation between tetrahedron count and object position deviation. However, as there are still some differences, this suggests that tetrahedron count is not the only correlating factor. The surface area of the mesh may still be this factor, as the values for the rest surface area of the MDCL-TMA meshes are almost the same.

Figure 7.16 compares the average RMSD across the simulations. While varied, the average RMSD values are all to a similar magnitude, rather than having a huge range between lower and upper bounds. However, an oddity is noted in the average RMSD values for the groEL mesh. While the average RMSD is correlated to the object's strain energy, it is inversely correlated to centroid deviation, i.e. the FFEAtools mesh, which has larger deviations in the position of its centroid over time, has a smaller average RMSD than the MDCL-TMA meshes (both the 8Å and 11Å meshes). If RMSD is meant to be a measurement of the dynamics and fluctuations in the simulation, it would be expected that the MDCL-TMA meshes with smaller centroid deviations would have *lower* RMSD values. The same occurs for dynein-1, however as the 1ns FFEA simulation did not complete for that object, it cannot be determined if the average RMSD of the MDCL-TMA mesh would have stayed lower than the FFEAtools mesh.

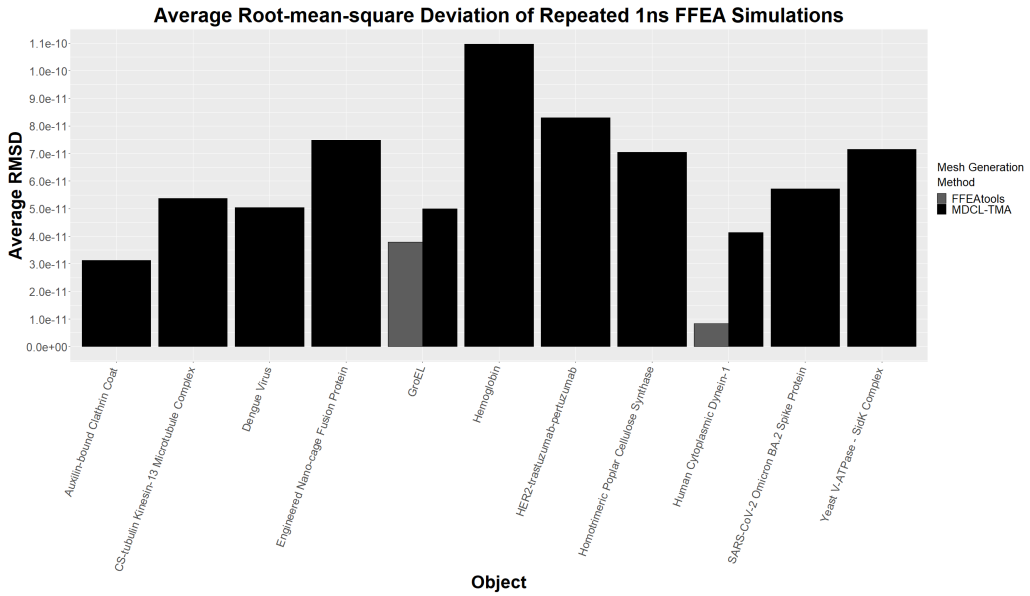

Figure 7.16: Average root-mean-square deviation from repeated 1ns FFEA simulations of meshed objects. Note that the results for dynein-1 are not comparable to each other as the simulation using the FFEAtools mesh did not complete the 1ns run.

One potential explanation for the discrepancy in RMSD is that FFEAtools produces surface meshes, whereas MDCL-TMA meshes have interior faces. This change in geometry may be affecting some measurements. To investigate this, a surface mesh was extracted from the 11Å groEL MDCL-TMA mesh. This was done by using the "extract surface" tool in ParaView, exporting it as an `.stl` surface mesh, and then using TetGen to convert it into `.ele`, `.face`, and `.node` files for use in FFEA. The 1ns FFEA simulation experiment was then repeated for this new surface mesh. Details of the mesh, and results of the 1ns FFEA simulations, are recorded in Table 7.4.

Table 7.4: Mesh details and results of 1ns FFEA simulations of groEL MDCL-TMA surface mesh at 11Å.

Number of Tetrahedra	**Object Rest Surface Area (Å²)**	**Object Rest Volume (Å³)**	**Object Mass (kg)**	**Wall Clock Time Taken (s)**	**Average WCT (s)**	**Average RMSD**	**Average ARMSD**
3014	2.35E+05	9.47E+05	1.42E-21	5.648	5.700131	4.64E-11	4.59E-11
				5.500		4.61E-11	
				5.952		4.51E-11	

Figures 7.17 and 7.18 plots the centroid deviation of the surface MDCL-TMA mesh against its original MDCL-TMA mesh and the FFEAtools mesh for groEL. The surface MDCL-TMA mesh deviates similarly, if slightly more, than the FFEAtools mesh. The average RMSD for the surface mesh is lower than the full MDCL-TMA meshes, but is now more comparable to the RMSD of the FFEAtools mesh, being slightly higher while deviating slightly more.

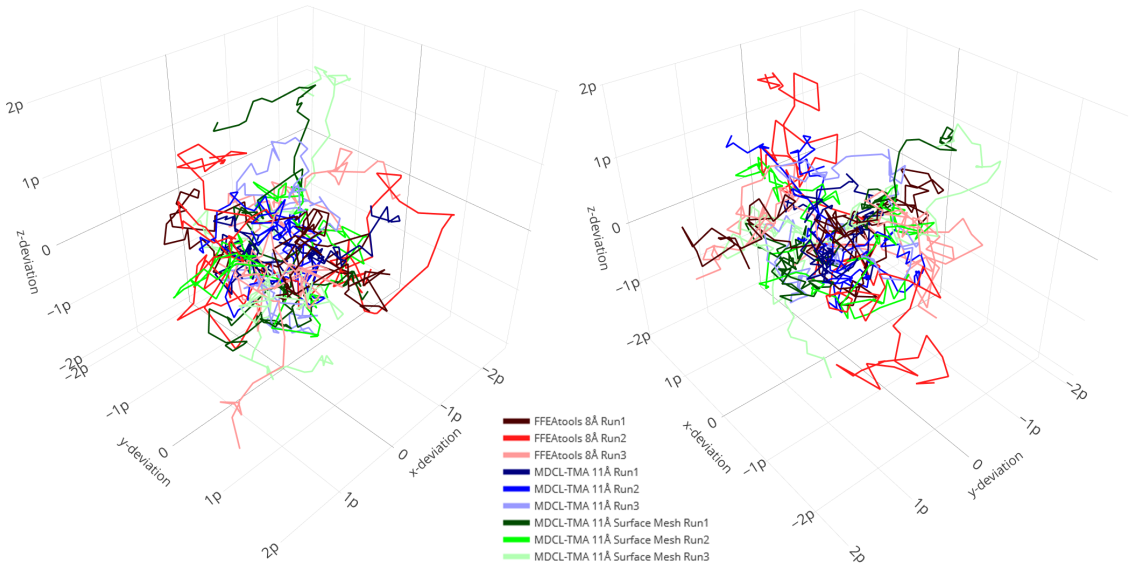

Figure 7.17: Centroid deviation over time during 1ns FFEA simulations for groEL with MDCL-TMA surface mesh, viewed at two angles. The surface mesh of the 11Å MDCL-TMA mesh (green lines) now has a much closer deviation in the position of its centroid over time in comparison to the FFEAtools mesh (red lines).

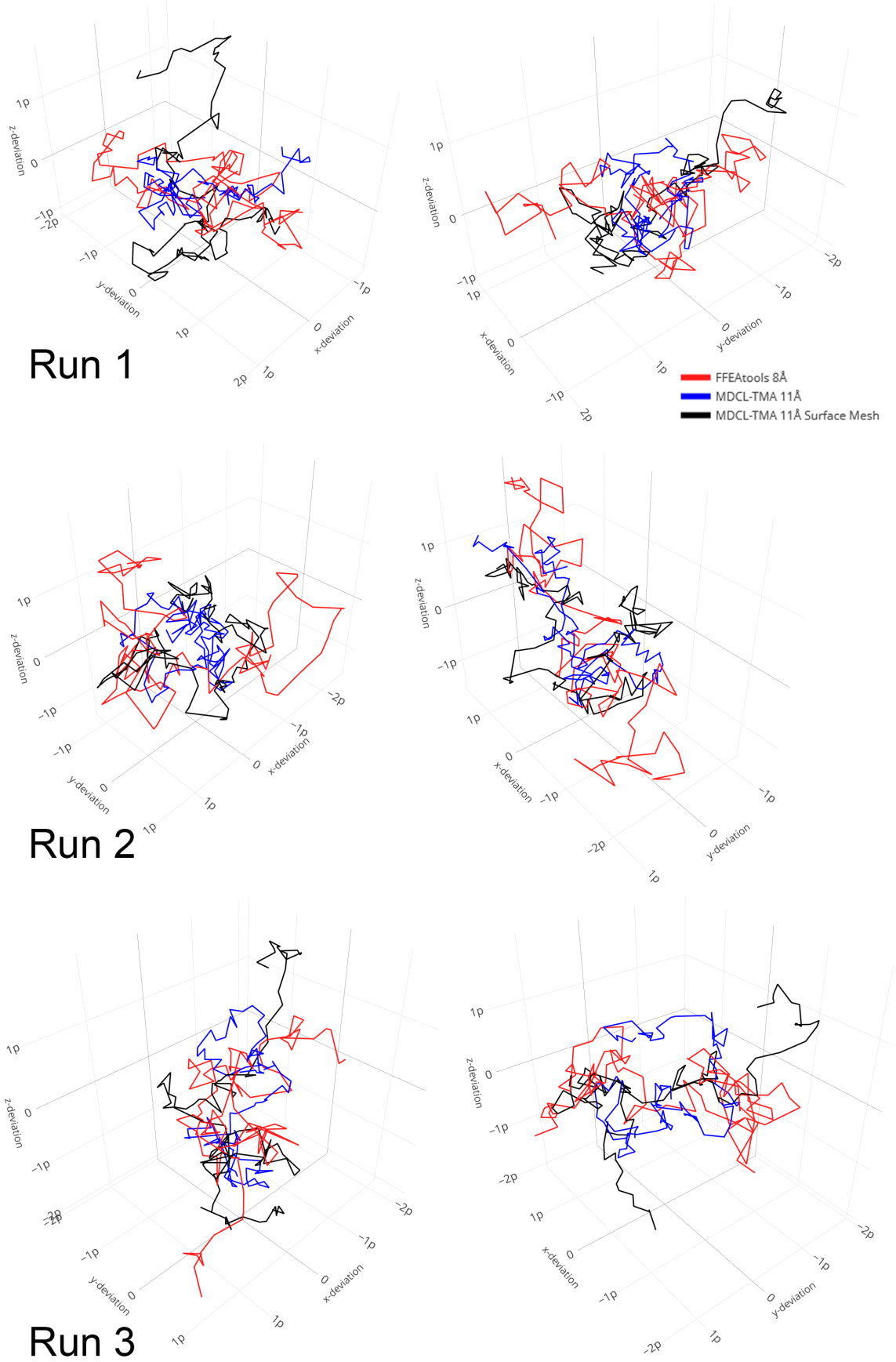

Figure 7.18: Centroid deviation over time during 1ns FFEA simulations for groEL, with MDCL-TMA surface mesh, split into runs (separated by row), each viewed at two angles.

While the results lack more comparison between FFEAtools and MDCL-TMA meshes to test this relationship rigorously, it is thought that the internal elements in MDCL-TMA meshes are causing this observed increase in average RMSD. The centroid deviation measurements are not accounting for the summation of movement across all points, just the movement of the object's centroid. While an MDCL-TMA mesh may move around its centroid less than a comparable FFEAtools mesh, the total fluctuations may still be larger due to the movement of internal elements, meaning a larger average RMSD.

7.2 Current limitations

It must be made clear that while the results of the previous sections appear damning to the capabilities of FFEAtools's meshing algorithm, these results do not mean that FFEAtools *cannot* mesh these objects, only that it is a challenge to mesh something using the default workflow (as confirmed by user complaints about the time-cost and trial-and-error process of meshing in FFEAtools in §4.2.1). With further time for tweaking parameters, FFEAtools is capable of meshing geometrically-complex objects (see Figure 7.19), however there is not sufficient time in this research to put into ensuring each object is meshed, nor can it be demonstrated clearly how to replicate the more in-depth meshing process for these experiments.

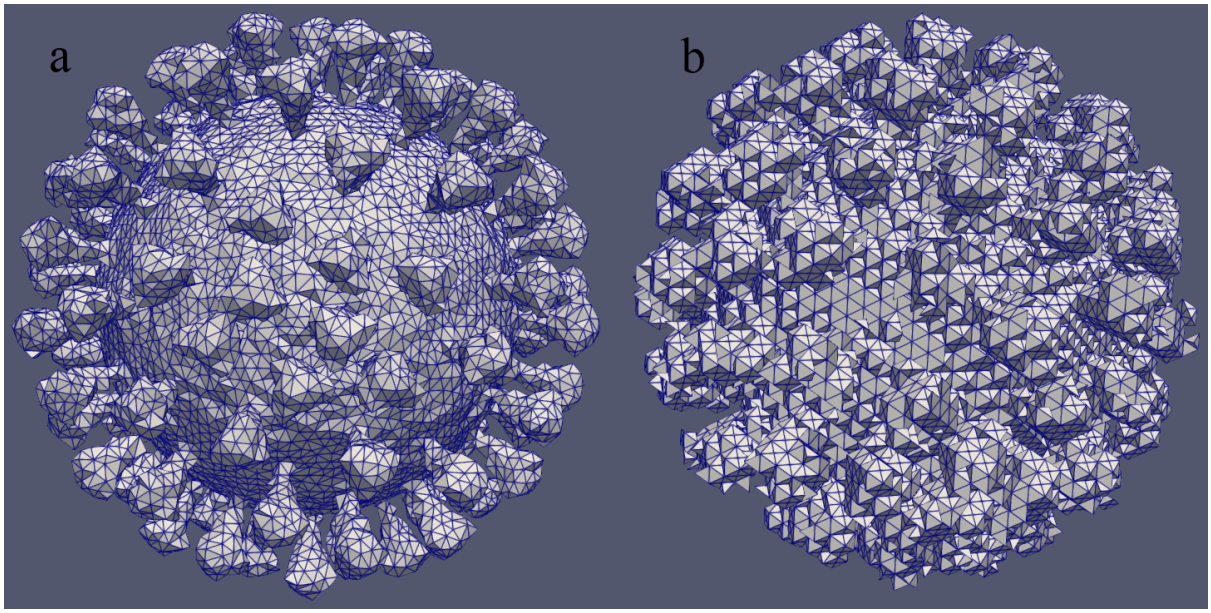

Figure 7.19: SARS-CoV-2 virion meshed a) in FFEAtools coarsened to 25Å and b) in MDCL-TMA coarsened to 40Å.

A subjective downside of MDCL-TMA is that the meshes produced are not as aesthetically

pleasing as meshes produced by FFEAtools (see Figures 7.1, 7.2, & 7.19), at least in this researcher's opinion. For the purposes of scientific experiments and the validity of results, this is not very important. However, one purpose of scientific visualization is to communicate research to wider audiences, and some visual clarity can be lost when using MDCL-TMA, depending on the geometry of the object and the coarse-graining value chosen.

While it is arguable if this is a limitation, MDCL-TMA produces meshes with internal elements, whereas FFEAtools produces surface meshes. This can create discrepancies in expected values of RMSD (§7.1.3). If a user wishes to turn an MDCL-TMA mesh into one with only surface elements, that currently requires external tools such as ParaView (though it is not a difficult or lengthy process). It is possible to update MDCL-TMA so that it produces surface meshes (or at least to give the user the option to do so) through removing internal elements. However, it would be worth asking FFEA users if this causes them an issue to warrant adding surfacing features to MDCL-TMA, as useful results have been obtained before with an earlier version of the MDCL-TMA algorithm (Gravett 2022), which also produced internal elements (§5.3).

7.3 Comparative analysis conclusions

From the comparative analysis across this chapter, one thing that is clear is that the default workflow for meshing in MDCL-TMA has a higher success rate than FFEAtools's default workflow, with all eleven objects being meshed in MDCL-TMA and only two in FFEAtools. The regularity of the algorithm's method of meshing, combined with the relatively fewer steps with fewer parameters to tweak seems to have paid off in making MDCL-TMA a more user-friendly meshing algorithm than FFEAtools's. Analysis of the experiences of other users is needed however before being able to claim that MDCL-TMA *is* more user-friendly for meshing than FFEAtools. Ideally this would be done by re-evaluating FFEA as performed earlier in this thesis, especially the qualitative analysis survey conducted in Chapter 4, however time-constraints prevented this from happening.

MDCL-TMA's ability to mesh is not perfect however, as the current Python implementation can come to a fatal segmentation fault when attempting to mesh an object too large. It is hoped that a more robust code implementation of MDCL-TMA can be developed to overcome this error in the future, but this is beyond the scope of this project due to time-constraints.

MDCL-TMA produced meshes of consistent quality, with all meshes producing almost the exact same values for the ratio of the minimum to maximum edge length, and for the quality-fitnesses (with some deviation in these values due to the interpolation to smooth the mesh surface). The FFEAtools meshes did not have this consistency. It is likely that MDCL-TMA will almost always produce meshes of a consistent quality, due to the consistency of the geometry of MDCL-TMA meshes, as shown by the mathematical proofs in §7.1.1.

The meshes produced, which can be seen in Figures 7.1, 7.2, and 7.3, do seem less visually-clear when using MDCL-TMA, compared to FFEAtools's meshing method of trying to form-fit to the object. This has no impact on the validity of their simulation results, but may negatively impact scientific communication through the visualization of FFEA results.

MDCL-TMA meshes appear to perform better and more reliably in FFEA simulations than meshes made with FFEAtools. Both FFEAtools meshes ended in element inversions during 48-hour simulations, whereas eight MDCL-TMA meshes reached the maximum wall clock time. Three MDCL-TMA meshes did result in element inversion errors, which shows MDCL-TMA is not immune to this problem, but those still managed to reach much higher wall clock times than the FFEAtools meshes. One very interesting result came from the object hemoglobin, which was meshed to a coarse-graining value of 4Å. Element inversion problems tend to become common when objects are meshed under a coarse-graining value of 5Å, and yet this object was able to reach the full 48-hour wall clock time. This suggests that MDCL-TMA produces a huge improvement to mesh stability over the FFEAtools meshing algorithm.

MDCL-TMA and FFEAtools meshes appear to experience the same kind of rates of change in strain energy over time, which implies that MDCL-TMA has not negatively impacted strain energy calculations in FFEA simulations. However, MDCL-TMA meshes when coarsened the same as their FFEAtools counterparts appear to result in a recorded higher strain energy over time. It is possible that this is due to MDCL-TMA meshes containing a larger number of tetrahedra than FFEAtools meshes at the same coarse-graining value. By re-meshing an object in MDCL-TMA to a similar tetrahedra count, rather than the same coarse-graining value, it was recorded to now have similar values of strain energy, implying that the number of tetrahedra in a mesh is proportionally correlated to strain energy, however this does not yet imply causation.

A relationship between object motion and strain energy was tested by plotting the deviation of the object's centroid over time. MDCL-TMA meshes were found to deviate much less than

FFEAtools meshes at either the same coarse-graining value or a similar tetrahedra count, however ones at a similar tetrahedra count have closer magnitudes of deviation. All this implies that as MDCL-TMA meshes gain higher tetrahedra counts, it experiences smaller centroid deviations and higher values of strain energy. Other factors however may have an impact on the strain energy, such as the surface area. More comparable FFEAtools and MDCL-TMA meshes are required to test this thoroughly.

When measuring average values of RMSD, which is used in FFEA simulations as an indication of the magnitude of fluctuations (higher RMSD values meaning larger fluctuations), an odd result occurred for the groEL meshes. The FFEAtools mesh had larger centroid deviations than that of the MDCL-TMA meshes, but had *lower* average RMSDs, when the opposite was expected. However, FFEAtools meshes are surface meshes, whereas MDCL-TMA meshes contain internal elements. An MDCL-TMA mesh was converted into a surface mesh by removing these internal elements, and the centroid deviations and RMSD values were remeasured. This MDCL-TMA surface mesh deviated from its centroid similarly to the FFEAtools mesh, if not slightly more, and its average RMSD was slightly higher than the FFEAtools mesh. This suggests that the fluctuating internal elements was raising the RMSD values even if it deviated away less from its centroid. Again, however, more comparable meshes are required to test this relationship rigorously.

Considering the results of this analysis, it is believed here that MDCL-TMA meshes behave as expected in FFEA simulations and produce valid results. Useful results have already been obtained with an earlier version of the meshing algorithm (Gravett 2022). Some measurements between FFEA simulations that use FFEAtools meshes and those that use MDCL-TMA meshes may not be directly comparable, such as RMSD, but this appears to simply be due to the geometric differences between the meshing algorithms, rather than MDCL-TMA introducing faulty calculations into FFEA. This is no different to experiencing variation in measurements between FFEAtools meshes coarse-grained to different values.

While FFEA users may need to make some adjustments to their workflow to use MDCL-TMA meshes instead of FFEAtools meshes, the advantages MDCL-TMA holds are believed here to more than account for that, especially when reducing the complexity and time-cost of creating meshes for FFEA simulations. In theory, this shorter and less-complex meshing method will result in an improved user experience, as it will be easier to learn and execute. It has also

been shown to produce more stable meshes for FFEA simulations, so there will be a further reduction in time-loss from having to restart simulation preparations due to an unworkable mesh. By reducing some of the common frustrations users face as expressed by participants in the qualitative survey in Chapter 4, this will hopefully improve user retention, and thus improve the software's longevity. In order to measure this, and also to see how this feature may have improved maintainability through its simpler source code, the qualitative survey should be re-conducted after users and developers have had time to utilise this new feature, however this is beyond the scope of this project due to time-constraints.

Chapter 8

Towards automated KOBRA rod identification

While a new meshing algorithm (detailed in Chapters 6 and 7) is one potential way of improving FFEA's sustainable qualities through developing new functionality, there are other possible features that could be developed for this purpose. Gravett (2022), in the future perspectives at the end of their thesis, makes the following suggestion:

> *[A]dapting the [marching tetrahedra] algorithm to also generate rods, or combinations of rods and blobs depending on the shape of the cryo-EM map would be a useful feature. For this to be automated may require machine learning to predict the type of representation (blob or rod) based on the shape of the map.*

Automation of the meshing process (or part of it) has been stated as a desired feature by FFEA users in the past (§4.2.1). Machine learning has been successfully used before to automate cryo-EM data identification and classification (Gravett, Cocking et al. 2021), so it seems plausible that it could be applied to determining the presence of rod structures in biomolecules for FFEA simulations. This would be especially useful for determining structures made up of multiple rods, or combined rod-blob structures, as if fully automated it would bypass the need for users to manually set up these connections to other meshes via adjusting text fields with no immediate visual feedback. In this chapter, work is began towards automating rod identification and KOBRA structure creation and parametrisation.

FFEA uses two formats of biomolecule representation; "blob" models for globular biomolecules,

and Kirchoff Biological Rod Algorithm (KOBRA) "rod" models for better representation of the dynamics of elongated biomolecules (see §2.4). In order to begin automating the process of identifying and generating KOBRA rods for a biomolecule, the algorithm first needs to be able to detect whether or not an input biomolecule contains any rod structures at all. This means a classification algorithm is needed to sort input biomolecules into two categories; "does contain rods", and "does not contain rods". This chapter details the design and development of such a classifier, and discusses potential future work for achieving the goal of fully automating the setup of KOBRA rod structures for FFEA systems. This makes up another attempt at the "Develop new functionality to improve sustainable qualities of research software" step of the general methodology (Figure 1.1), though as this is only a prototype of new functionality the step is not done to completion this time. The chapter is then concluded by considering the development of this prototype feature against the second and third questions of the research scope (§1.1), to see how the sustainable qualities of research software can be improved, and if improving these qualities will improve the user experience, maintainability, and longevity of the software.

8.1 Classification algorithms

To obtain an algorithm that can classify data for this particular problem, this section will explore appropriate machine learning techniques. Machine learning is a category of statistical analysis algorithms that attempt to solve problems through "learning" by "remembering" the results of previous analysis attempts through generalising the connection between inputs and the accuracy of found outputs. These algorithms then attempt to solve that problem for new situations (i.e. different input data) based on its generalised understanding of the function that it has "learnt" from previous experience (Marsland 2015). This methodology is generally used for solving tasks that are too complex to program, such as tasks inherently understandable to animals but not to machines (e.g. understanding images and sounds), or tasks that could not be solved by a human (or at the very least not in a reasonable time frame) due to the dataset being too large or complex (Shalev-Shwartz and Ben-David 2014).

In this classification problem, the input data are known values, as are the range of desired outputs, i.e. for each biomolecule, a human can look at the shape and determine from visual analysis whether or not it contains rod structures. The goal is to create an algorithm so that

machines can do this task instead, so that when combined later with an algorithm to determine the location of rod structures and automatically generate KOBRA models, it will reduce the time a researcher has to spend on setting up FFEA simulations with KOBRA. The category of machine learning algorithms suitable to this problem are called supervised learning algorithms. In supervised learning, the algorithm is given a training dataset where each entry is labelled with the class(es) they should be categorised into, and it attempts to "learn" the connections between the inputs and the output class through generalisation (Marsland 2015). Afterwards, the algorithm is then typically given a new dataset to test their "knowledge" on, where the classes are hidden from the machine learning algorithm. The model attempts to classify the test dataset, and the accuracy and efficiency of the algorithm can be determined from how many data objects it correctly classified (Shalev-Shwartz and Ben-David 2014).

Classifying the presence of rods in biomolecules was attempted with three machine learning algorithms; k-nearest neighbours, decision trees, and support vector machines. All of these algorithms are supervised learning methods.

All these machine learning algorithms were programmed using the `scikit-learn` Python library (Pedregosa et al. 2011), which implements a variety of machine learning algorithms with a focus on ease-of-use. The main reason for its use in this research is this ease of implementation, meaning machine learning systems can be programmed with minimal time-cost. The consistent format of input parametrisation and output results also makes it easy to implement and compare multiple machine learning algorithms.

The following subsections explains how these algorithms attempt to solve classification problems, their advantages and disadvantages, their general use cases, and why they are used for this research.

8.1.1 K-nearest neighbours

K-nearest neighbours (k-NN) is a machine learning algorithm that attempts to produce unknown labels (or classes) of a queried piece of data via analysing neighbouring data points with known labels (Aggarwal 2015). The k in k-NN refers to the number of neighbours analysed to predict a data point's label, and a value must be chosen to be large enough to not be influenced by noise in the data, but small enough to reduce the computational complexity (Marsland 2015). Generally, k should be a value that is *not* a multiple of the number of different classes to prevent

ties when determining the class of label. Figure 8.1 shows an example of the k-NN process for a 2D binary problem.

k-NN is a very simple machine learning algorithm, which makes it applicable to a wide range of scenarios, and it can analyse most types of data providing that the "distance" between data points in a search space is quantifiable (Shalev-Shwartz and Ben-David 2014; Aggarwal 2015). One of the major flaws with k-NN however is that it is not a very scalable algorithm (Aggarwal 2015). The computational speed and efficiency of the algorithm will decline as the size of the analysed dataset increases, as predictions have a quadratic time complexity ($O(n^2)$). The accuracy of the algorithm can also be negatively impacted by datasets with a higher dimensionality (i.e. they have a high number of input variables), as it can struggle to determine which input variables are actually of importance to the problem to be solved and have its results affected by irrelevant data. These problems can be alleviated through careful selection of relevant data features, and by reducing the amount of data the algorithm has to sort by the use of indexing, sampling, or aggregating data into clusters.

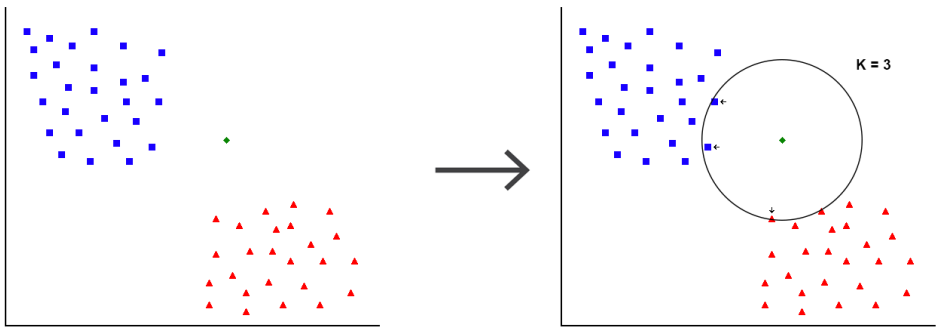

Figure 8.1: Example of the k-nearest neighbours method for a 2D binary classification problem. The two classes are represented by blue squares and red triangles, and form visually-obvious clusters in the search space. The green diamond represents the data point with an unknown class. With $k = 3$, a selection radius expands from the unclassified data point until it encompasses its three nearest neighbours. Of these neighbours the algorithm determines their most common class, and the queried data point is labelled with that class. In this case, its three nearest neighbours are two blue squares and one red triangle, so the data is labelled as being in the blue square class.

This research uses k-NN simply because of its ease of implementation, requiring very little work to do so. Regardless of how suitable this method is to this research's classification problem compared to the other classification algorithms in this section, it seemed nonsensical not to include due to taking mere minutes to program.

In the `scikit-learn` implementation of k-NN, a user can supply a range of values for k, and

the algorithm will try all of them and return the results with the highest accuracy. This means that as long as an appropriate range is supplied, there is no need to carefully parametrise the value of k and the algorithm will determine its appropriate value.

8.1.2 Decision trees

The decision tree method breaks down a classification problem into a set of decisions about the data features, starting at a root node that branches into further nodes for each answer to a decision (Marsland 2015). The algorithm attempts to classify data by progressing along the tree taking decisions until it reaches an end node (or leaf) which it then takes as the class to label that data point. More-important decisions should be closer to the root so that final decisions can be reached as efficiently as possible. Decision trees use a greedy selection method to construct trees, where at each step the algorithm returns the locally-optimal decision (Shalev-Shwartz and Ben-David 2014) to attempt to order nodes by importance. The locally-optimal decision for this algorithm is the data feature that returns the highest entropy of information, i.e. the data feature that when known provides the most additional information on how datapoints would be classified (Marsland 2015). Figure 8.2 shows a simple example of a decision tree for a binary classification problem.

Figure 8.2: Example of a decision tree for choosing to wear a coat (a binary classification problem). The root decision is "Is it raining?" because regardless of any other factor, I will want to wear a coat if there is rain, so I want to always check that first. If there is no rain, I may still want for a coat to keep me warm if the weather is cold, so that is the next thing I check. Finally, if there is no practical reason I need to wear a coat, I may still wish to wear one for the fashionable aesthetic, however as this is a frivolous reason I do not want it to be a main factor in my choice, so it is the furthest decision away from the root node.

Decision trees have a low computational cost to generate, and an even lower computational cost to use (Marsland 2015). This is because predictions are logarithmic to the number of data samples ($O(\log n)$), making decision trees an efficient and scalable algorithm. Decision trees are also easy for humans to comprehend, as one just needs to follow the tree down a path of decisions to obtain an answer to a problem (Marsland 2015), thus making the problem and its results more communicable to an audience, as well as making it easy to understand how to adapt the tree to better fit the problem.

A problem that can occur in machine learning systems is "overfitting". This is when an algorithm is trained too much on a training sample of data that it "learns" any inaccuracies, such as noise, as important generalisations to make for future data, essentially making a model that is too complex and too specific to the training sample to be useful (Marsland 2015). A significant disadvantage of decision trees is that the method is prone to overfitting by creating trees that are too large, as there is no way to algorithmically predict optimal tree length. This problem can be alleviated however through the process of "pruning", i.e. removing or replacing unnecessary nodes and subtrees from the decision tree (Aggarwal 2015). These unnecessary segments of the decision tree can be replaced with such things as leaf nodes to turn an over-complex decision into a final decision, or replacing lengthy subtrees with smaller and more concise subtrees. Careful selection of data features, such as avoiding redundant data features, can also help prevent decision trees from growing too large and overfitting in the first place.

Decision trees were chosen for this research as the method has been previously successfully implemented for image and shape classification problems, similar problems to this research's geometric classification problem. Keskin et al. (2012) used randomised decision forests (forests being a collection of multiple trees) to analyse 2D images of human hands with depth data to estimate poses by classifying pixels of the image data into parts of the hand, with their models recording accuracies of 90.9 - 91.2%. Xu, Liu et al. (2020) discusses the widespread use of deep neural networks (DNN) in hyperspectral (i.e. pertaining to electromagnetic information) image classification problems, noting that the complexity of DNN models can lead to overfitting with increased model parameters, and can have large hardware and training requirements. They proposed the use of a feature fusion and multi-layered gradient boosting decision tree (FF-DT) for hyperspectral image classification, with the goal of having a model that can more efficiently extract and classify spectral and spatial features of the data. Across three datasets,

the researchers found FF-DT, compared to previous DNN methods, had simpler requirements for training and model parameters, while obtaining classification results of a consistent high accuracy.

8.1.3 Support vector machines

When a typical machine learning classification algorithm attempts to separate data into classes, generally there is no regard to making the best-fitting classification line (i.e. the line that separates the classes) (Marsland 2015). If a classification line successfully splits current data into classes, many machine learning algorithms will simply consider that a success, regardless of impact on the classification of future data. Figures 8.3 and 8.4 show an illustrated example.

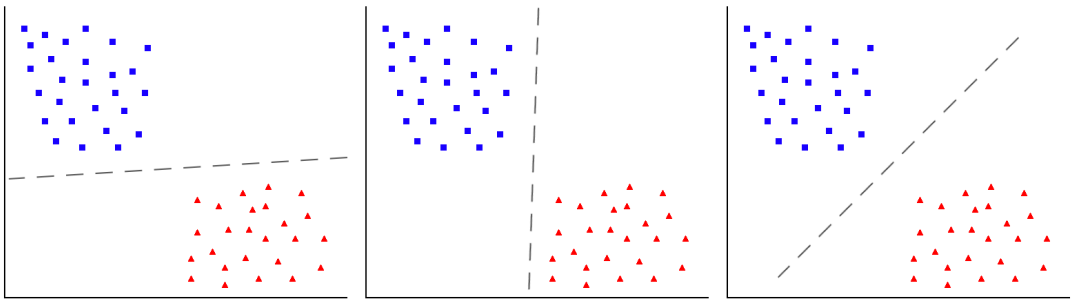

Figure 8.3: Three different linear classification lines for the same dataset. The two classes are represented by blue squares and red triangles, and the classification line is represented by dashed grey lines. Despite being very different classification lines, they all successfully separate the current data, so which one is the best for future data?

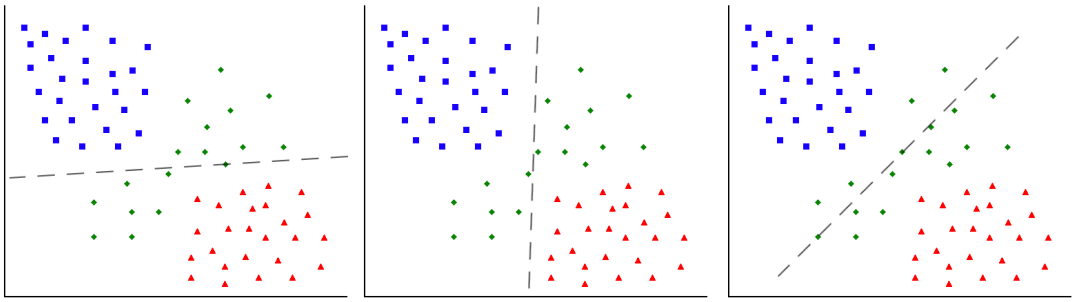

Figure 8.4: Dataset and classification lines from Figure 8.3 with new data. The new unclassified data is represented by green diamonds. As can be seen, despite the classification lines successfully separating the two known classes, they will result in wildly different classifications of the new unclassified data.

Support vector machines (SVM) introduce margins to classification lines in order to create optimal separations between classes (Aggarwal 2015; Marsland 2015). From an existing classification line, one can measure the perpendicular distance from the line to the closest datapoint,

and draw a cylinder or hyper-cylinder (depending on dimensionality) around the classification line with a radius equal to this measured distance. The classification line that produces the largest radius for the margin is called the maximum margin classifier and is used as the best-fitting classification line, and the datapoints in each class closest to the margins are called support vectors (hence the name of the algorithm). Figure 8.5 illustrates this idea. An advantage to this method is that after training on a dataset, only the support vectors are needed for future classification, as the other datapoints have no effect on the classification line, and this can save data storage space (Marsland 2015).

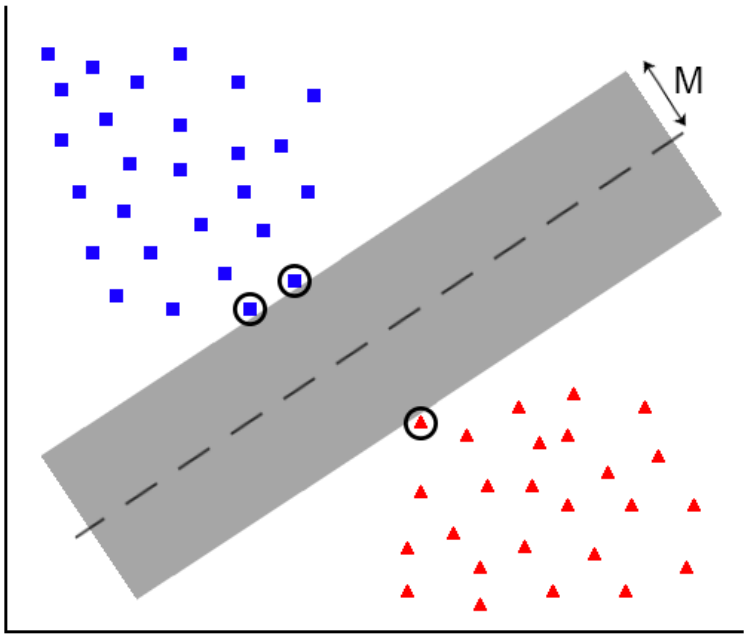

Figure 8.5: Dataset from Figure 8.3 with maximum margin linear classifier. The grey cylinder of radius M represents the margin around the classification line, and the circled datapoints are the support vectors. Through finding the classification line with the largest value of M, this has created the best-fitting separator between the two classes.

A limitation of many machine learning classification methods is that they can only identify straight line classifiers, i.e. groups of data in a search space can only be separated by a straight line or a hyperplane (depending on dimensionality). This stops these methods from being able to categorise data that do not form linearly-separable clusters (Marsland 2015). There are workarounds to this problem, such as adding new dimensionality to a dataset based on existing data features in such a way that classes now *can* be separated linearly. SVM does this through kernel functions which transforms the input data into a linearly-separable state (Aggarwal 2015; Marsland 2015), which is equivalent to having non-linear classification lines.

`scikit-learn` SVMs have access to four in-built kernel functions; linear, polynomial, radial basis function (RBF), and sigmoid, where the linear kernel applies no transformation to the data and acts as a standard straight line classifier.

One of the main advantages of SVM is its ability to produce accurate results with smaller datasets, and with highly dimensional datasets (Shalev-Shwartz and Ben-David 2014; Marsland 2015). This can make SVM a suitable classification algorithm for problems failed by other machine learning methods, where samples of data are limited or have more dimensions than data samples. A major disadvantage of this method however is that predictions are slow due to a high computational complexity. In the `scikit-learn` implementation of SVM, the complexity can range from $O(n_{\text{features}} \times n^2_{\text{samples}})$ to $O(n_{\text{features}} \times n^3_{\text{samples}})$ depending on the efficiency of the solver, which is dependent on the dataset used (scikit-learn Developers 2023). This makes SVM very unscalable, and generally unsuitable for use on large datasets (Marsland 2015).

SVM was chosen for this research as the method has been previously successfully implemented for image and shape classification problems. Conceicao et al. (2010) noted issues with existing classification methods of analysing microwave images of breast cancer tumours, such as the analysis of image contrast being unable to distinguish between those benign and malignant. The researchers instead used tumour information obtained from radar target signatures, which provides data on the size, shape, and texture of tumours, and then applied SVM to important information extracted from the data. Their models recorded high accuracies in shape and size classification, although this accuracy decreased as the granularity of the data increased. Islam et al. (2017) applied multi-class SVM to segmented images of plant matter to detect disease, using diseases in potato plants as an exemplar. The images of diseased leaves are segmented by healthy and diseased regions so that the characteristics of these leaves are easily understood by the machine learning algorithm. The SVM models applied to this data recorded a classification accuracy of 95%.

The size of the dataset used for this research is of a limited size due to the time-cost of preparing data (see §8.2), therefore SVM was also chosen so that it may be able to obtain accurate results from this relatively small dataset. It is also very easy in `scikit-learn` to switch between different SVM kernels by changing a single parameter, meaning all available kernels can be easily tested and compared.

8.1.4 K-fold cross-validation

In order to obtain comparable results from using these classification algorithms, an appropriate statistical model validation method is required to determine the efficiency when applied to the rod detection problem. The validation method chosen for this problem is k-fold cross-validation, which partitions the training dataset into k number of subsets (called "folds"). For each fold, the algorithm is trained on every other fold and then tested on the current fold, returning the accuracy from the test dataset. This is independently repeated for every fold, and then the mean accuracy is calculated to get a more reliable estimate of how likely the algorithm will correctly classify future unlabelled data (Shalev-Shwartz and Ben-David 2014).

A key advantage to k-fold cross-validation is that all data is used for training and testing, meaning no data is left behind in the algorithm's "learning" process, rather than having to set aside some data purely for validation purposes (Shalev-Shwartz and Ben-David 2014). This makes it a valuable validation method for smaller datasets, which is the main reason for choosing this method, as the dataset for the rod classification problem is relatively small (see §8.2).

8.2 Electron density map classification

In the first attempt at classifying the presence of rod structures in biomolecules, the electron density map point cloud data of biomolecules was used as a dataset for the aforementioned machine learning algorithms to classify. The list of all biomolecule data used for this dataset can be found in Appendix B in Table B.1. The dataset contains 30 biomolecules that contain rod structures, and 33 biomolecules that contain no rod structures, for a total of 63 biomolecules. This data was obtained from the Electron Microscopy Data Bank (European Molecular Biology Laboratory – European Bioinformatics Institute 2023) in the format of `.map` files, which were converted to `.mrc` files with the "save as" function in ChimeraX (see §5.1 for more details on file types). This dataset is relatively small, as it required manual visual analysis to select appropriate biomolecules that could be sorted based on whether or not they would contain rod structures. Processing this data into other formats was also a slow and manual process when attempting alternative classification methods such as in §8.3, limiting the amount of data that could be obtained in the time frame of this research.

This classification method used the Python library `mrcfile` (Science and Technology Facilities

Council 2017) to parse and extract data and header data from `.mrc` electron density map files. This data, alongside a binary label to state whether or not the biomolecule contained rod structures, was written to a `.csv` file. The `.csv` file's rows were then shuffled randomly to ensure no ordering bias could impact the results, and this shuffled `.csv` file could then be parsed by the machine learning algorithms to train and test with this dataset to produce k-fold cross-validation scores, which could be used to compare the accuracy of each algorithm for this problem. This general methodology was attempted as it was simple and fast to implement, and sometimes the simplest classification methods can be very effective (Jiang et al. 2023).

Simply using the electron density map data as data for the classification was not a viable method, as this unstructured data was in the format of arrays of varying lengths that could take up GBs of data storage. Therefore, features of the data were collected, alongside parts of the electron density map header data (see Cheng et al. (2015) for further information on `.mrc` file data), in order for the machine learning algorithms to attempt to determine geometric features. This gave a dataset with the following data features:

- The x, y, and z dimensions of the data array.
- The total number of elements in the data array.
- The total bytes in the data array.
- The x, y, and z dimensions of a unit cell
- The number of intervals along the x, y, and z axes in a unit cell.
- The minimum, maximum, and mean density values.
- A binary label stating whether or not the biomolecule contains rod structures determined by manual visual inspection.

8.2.1 Results

Table 8.1 shows the results of attempting to classify the presence of rods in electron density maps using the machine learning algorithms discussed in the previous section. For the k-fold cross-validation, $k_{cv} = 5$. Cross-validation scores range from 0-1, with 0 meaning the algorithm is never correct, and 1 meaning the algorithm is always correct.

The results are poor, with decision trees and SVM with a sigmoid kernel achieving the highest

mean k-fold cross-validation scores of $0.3\dot{6}$. This means that all algorithms performed worse than random chance, which for a binary classification problem like this would be equivalent to a cross-validation score of 0.5. SVM with a linear kernel failed to predict any classifications at all.

Table 8.1: K-fold cross-validation scores for rod classification from electron density map point cloud data where $k_{cv} = 5$. Runs that failed to produce a result are denoted by "N/A".

Algorithm	**K-Fold Cross-Validation Score**					
	1	**2**	**3**	**4**	**5**	**Mean**
k-NN ($k = 12$)	0.5	$0.\dot{3}$	0	$0.\dot{3}$	$0.\dot{3}$	$0.2\dot{9}$
Decision Trees	0.5	$0.\dot{3}$	0	0	1	$0.3\dot{6}$
SVM - Linear	N/A	N/A	N/A	N/A	N/A	N/A
SVM - Polynomial	0.25	$0.\dot{3}$	0	$0.\dot{3}$	$0.\dot{3}$	$0.24\dot{9}$
SVM - RBF	0.25	$0.\dot{3}$	$0.\dot{3}$	$0.\dot{3}$	0	$0.24\dot{9}$
SVM - Sigmoid	0.5	$0.\dot{3}$	$0.\dot{3}$	$0.\dot{3}$	$0.\dot{3}$	$0.3\dot{6}$

This method to attempt classifying the presence of rod structures from the point cloud data has resulted in failure. This is likely due to the machine learning algorithms not being able to parse enough useful and discernible information from this data format. The information stored in the header of `.mrc` files does not give enough information alone to thoroughly describe the geometry of the object, and the machine learning algorithms were unable to properly parse the unstructured main data. This all suggests that applying supervised machine learning algorithms to MRC electron density maps this way is not an appropriate method for classifying their geometry.

While this attempt to classify geometries from the electron density map point cloud data did not yield usable results, there are still other potential methods of applying classification algorithms to point cloud data that may produce better results. There are methods that have been developed specifically to classify point cloud data, such as PointNet (Qi et al. 2017), PointHop (Zhang et al. 2020), and PointCNN (Li et al. 2018).

Implementing point cloud classification methods to this rod structure classification problem would likely be a valid method of obtaining more accurate results. However, time constraints prevented being able to schedule training with these methods to apply them to this problem. Future researchers may wish to attempt this method to see if it is a robust approach to classifying biomolecules with rod structures.

8.3 Curvature cluster analysis classification

The next attempt to try and classify the presence of rod structures in biomolecules was to analyse the curvatures across the surfaces of molecule meshes. In geometry, curvatures describe how a bending line or surface deviates from being a straight line or plane (Klaf 1956). As different types of shape bend in different ways, so too will the curvatures be different, and this can be used to describe the geometry of a shape, such as if it contains rod-like structures or not. This method attempts to use this information as input data to a machine learning algorithm so that it can learn to distinguish geometries from curvature values.

There are three types of curvature which will be used to describe the surface geometry of the data; the principal, Gaussian, and mean curvatures. The principal curvatures are defined as the pair of eigenvalues (a minimum and a maximum) of the shape operator (Sternberg 2012), the shape operator of a point p being calculated by

$$S_p(v) = -\nabla_v U, \tag{8.1}$$

where S_p is the shape operator, v is the tangent vector of the surface, and U is the unit normal vector field (O'Neill 2006). Gaussian curvature describes the geometry through the intrinsic measurement of angles and distances on the surface, without directional reference (Sternberg 2012). Mean curvature is an extrinsic measurement of curvature, and is defined by the mean of the two principal curvatures for a given point on the surface (Kenmotsu 2003).

The VTK C++ and Python library (Kitware Inc. 2023) contains functions for extracting the mean and Gaussian curvatures at each node in a mesh. With these values, the minimum and maximum principal curvatures can be calculated, and all of these curvatures can be analysed and compared by a machine learning algorithm to attempt to understand the geometry of the mesh. VTK calculates mean curvature (κ_H) with

$$\kappa_H = \overline{H_e}, \tag{8.2}$$

where $\overline{H_e}$ is the average over the edge neighbours H_e, calculated by

$$H_e = l(e) \times \alpha(e), \tag{8.3}$$

where e is an edge, l is the length, and α is the dihederal angle between edge neighbours in radians between $-\pi$ and π (Kitware Inc. 2023). Gaussian curvature (κ_G) is calculated in VTK with

$$\kappa_G = 2\pi - n_v f_v(\alpha), \tag{8.4}$$

where at vertex v, n_v are the facet neighbours and $f_v(\alpha)$ is the radian angle of facet f at the vertex (Kitware Inc. 2023).

With the values of mean and Gaussian curvatures, the maximum (κ_1) and minimum (κ_2) principal curvatures of a given point or node n can be calculated with

$$\kappa_1(n) = \kappa_H(n) + \sqrt{\kappa_G^2(n) - \kappa_G(n)} \tag{8.5}$$

and

$$\kappa_2(n) = \kappa_H(n) - \sqrt{\kappa_G^2(n) - \kappa_G(n)} \tag{8.6}$$

respectively (Meyer et al. 2003). In code, these curvatures will be stored in arrays indexed by the respective node in the mesh, which can then be normalised for the machine learning algorithm to parse (see §8.3.3).

8.3.1 Validation of method

Before the curvature analysis method can be applied, it needs to be validated against known cases with expected results to ensure that everything is being calculated correctly. This can be done by using test shapes of a simple geometry and inputting meshes of those shapes into the algorithm. Table 8.2 details the meshes of test shapes used for these validation tests. The sphere was used to represent an FFEA blob object, and the prolate spheroid to represent an FFEA KOBRA rod object. The torus represents neither, and is used as a test of the effect of surface topography on curvature. These meshes were created using Blender and exported as `.stl` files (as this format easily keeps element connectivity information), and then these were converted into `.vtk` files in ParaView as that is the format used as input for the algorithm.

Table 8.2: The radii and the number of segments across planes for meshes of test objects for curvature analysis validation.

Shape	**Radius(x)**	**Radius(z)**	**Segments(x-y)**	**Segments(x-z)**
Sphere	0.5	0.5	8	8
			16	16
			32	32
			64	64
Prolate Spheroid	0.5	10	8	160
			16	320
			32	640
			64	1280
Torus	1	0.25	48	12
			96	24
			192	48
			256[1]	96

The first thing to validate is if the algorithm gives accurate approximations of principal curvatures. This can be done by testing recorded results against the expected curvature of a completely-smooth spheroid which can be calculated with

$$\text{Curvature}_x = \frac{1}{R_x}, \tag{8.7}$$

where R_x is the radius in the given direction x. This gives an expected curvature of 2 for a smooth sphere of radius 0.5. For a smooth prolate spheroid of radius 0.5 and 10 in the x and z directions respectively, this gives expected curvatures of 2 and 0.1 in the respective directions.

These expected curvature values are then tested against discretized meshes of the same object at different resolutions to see if as the resolution is increased, that the recorded values of curvature converge towards the expected values. This is done by plotting scatter graphs of the values of minimum and maximum principle curvatures of nodes along the meshes, and by plotting convergence lines where the expected curvature values are. Figures 8.6 and 8.7 shows these scatter plots for the sphere and prolate spheroid respectively.

The outliers in these graphs are attributed to the two points at each end of the spheroid meshes where the segments converge. Figure 8.8 visualises these outliers on the meshes.

[1] 384 segments would have been used here to keep the ratios the same, but 256 was the upper limit that Blender allowed.

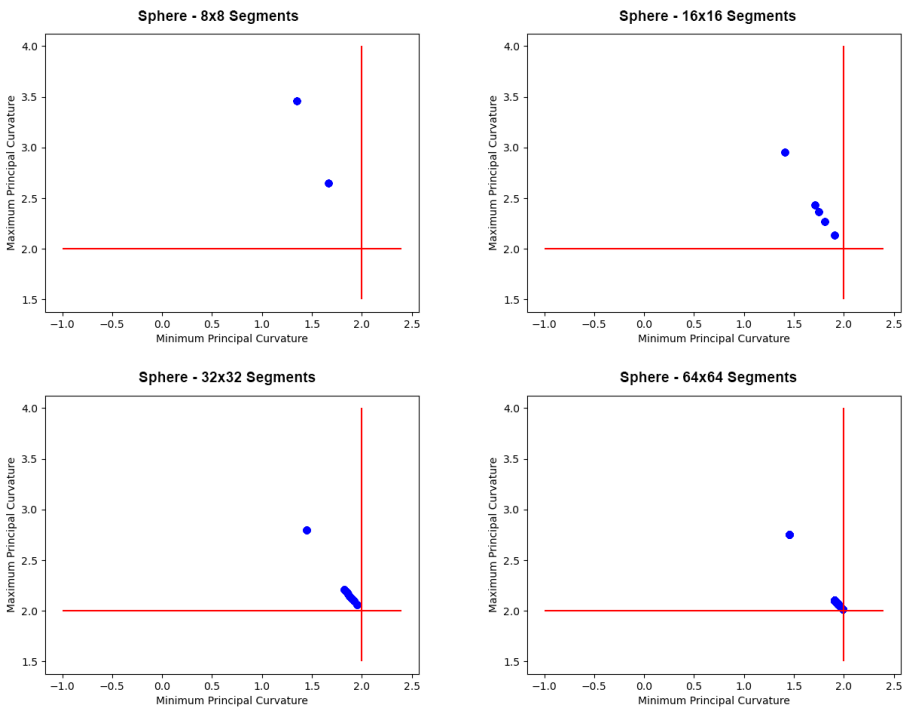

Figure 8.6: Scatter plots of sphere convergence tests for curvature analysis. The solid red lines represent the expected curvature of a smooth spheroid of radius 0.5 as calculated by Equation 8.7. As the resolution of the sphere increases, the points on the scatter plot converge to where the two lines cross.

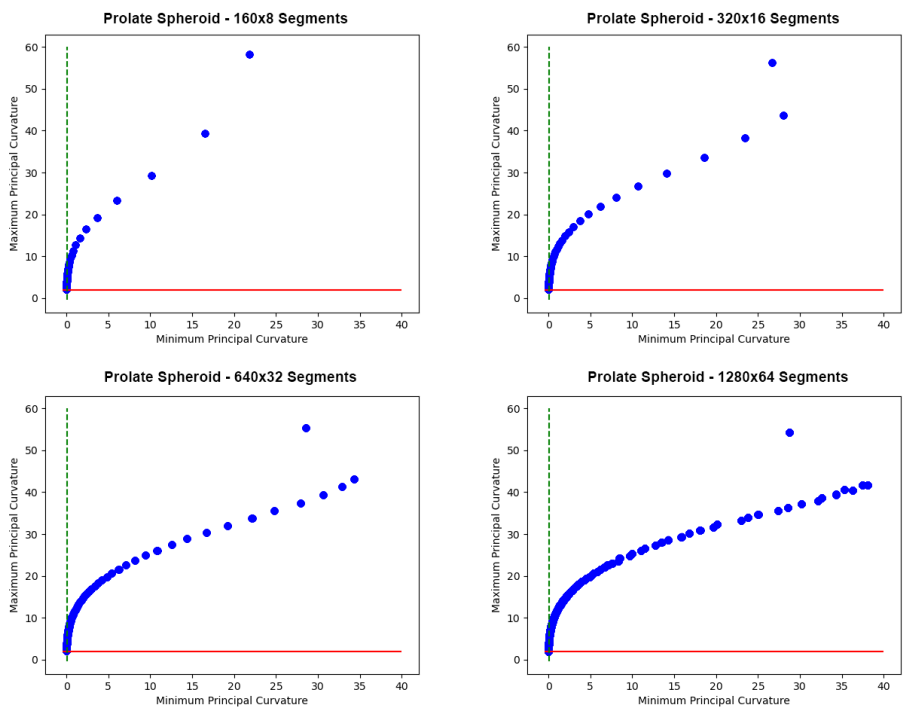

Figure 8.7: Scatter plots of prolate spheroid convergence tests for curvature analysis. The solid red line represents the expected curvature of a smooth spheroid of radius 0.5 as calculated by Equation 8.7, and the dashed green line represents the expected curvature at radius 10. In all cases, the plots of curvatures trend towards the point where the two lines cross.

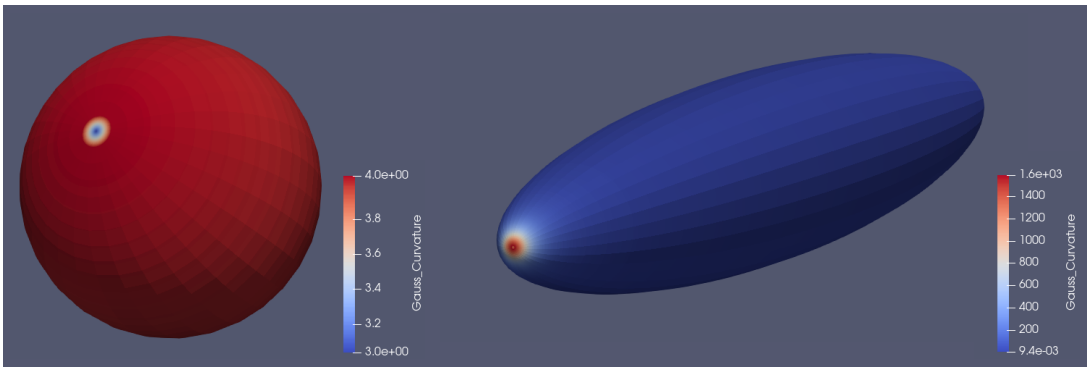

Figure 8.8: Sphere and prolate spheroid curvature heat maps viewed in ParaView. The tips of each mesh where the segments converge show outliers in the values of Gaussian curvature.

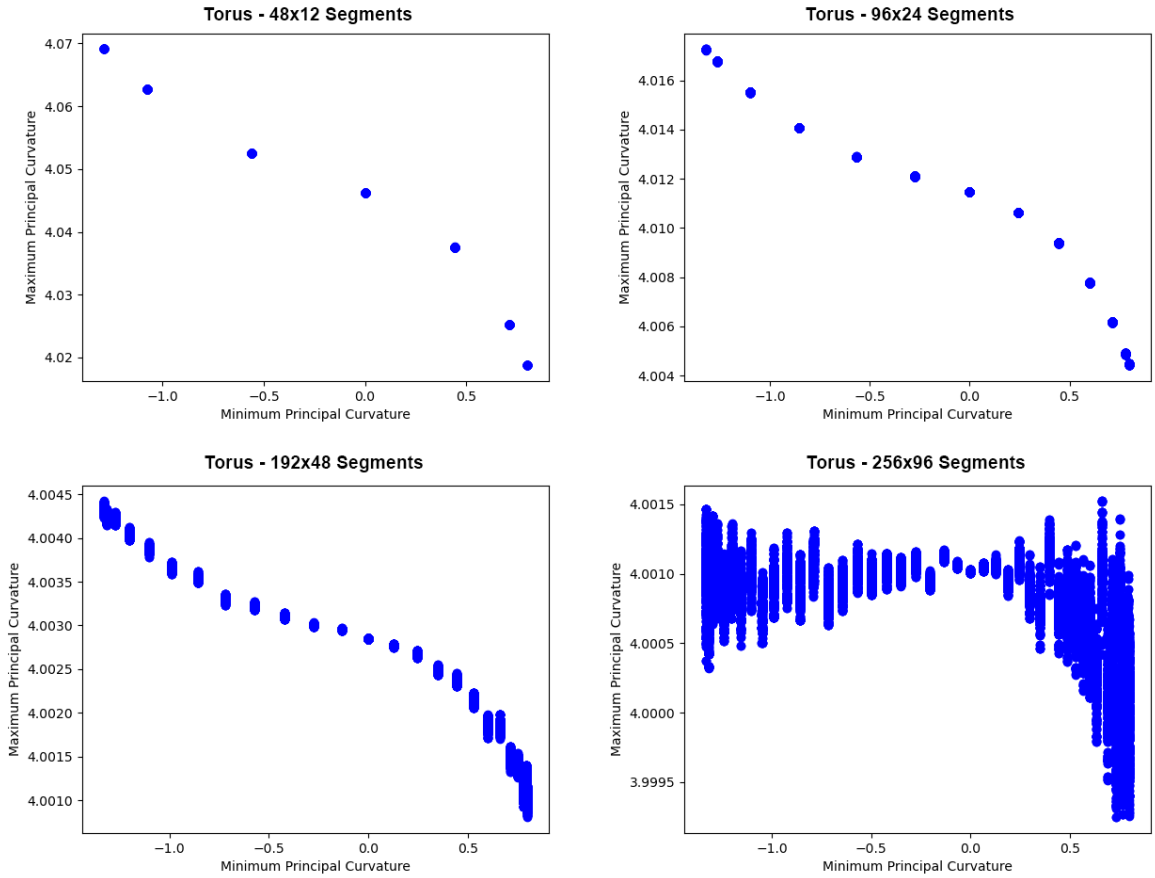

Figure 8.9: Scatter plots of torus curvatures. Just over half the points on the graph record negative minimum principal curvatures, and just under half record positive minimum principal curvatures. The data shows the negative curvature values are of a higher magnitude than the positive values, suggesting that the negative curvature values are along the inside of the torus, as the inner surface of the torus will have a smaller radius than its outer surface, producing a higher magnitude of curvature. This all implies that the polarity of curvature is dependant on the surface being concave or convex, with negative values corresponding concave surfaces and positive values corresponding to convex surfaces.

While the curvatures are being calculated mostly as expected, when this algorithm was applied to real molecule data it recorded some negative values for curvature. Positive values of curvature

describe a convex surface, and negative values describe a concave surface. It needs to be validated that the polarity of the curvature is accurately describing these types of surfaces in this application. This was tested using a simple shape that has both concave and convex surface geometry - a torus. Figure 8.9 shows the scatter plot of minimum and maximum principal curvatures for meshes of tori at different resolutions.

To verify these implications of the relationship between curvature polarity and the surface being concave or convex, a heat map of curvatures along the torus is shown in Figure 8.10. As can be seen, the surface along the inside of the hole of the torus has negative values of Gaussian curvature (used to calculate principal curvatures by Equations 8.5 and 8.6), and the surface along the outside has positive curvature values. This means that the recorded negative values from analysing real molecule data are accurate descriptors of surface curvature, which means this method is valid for analysing the surface curvatures of biomolecule data.

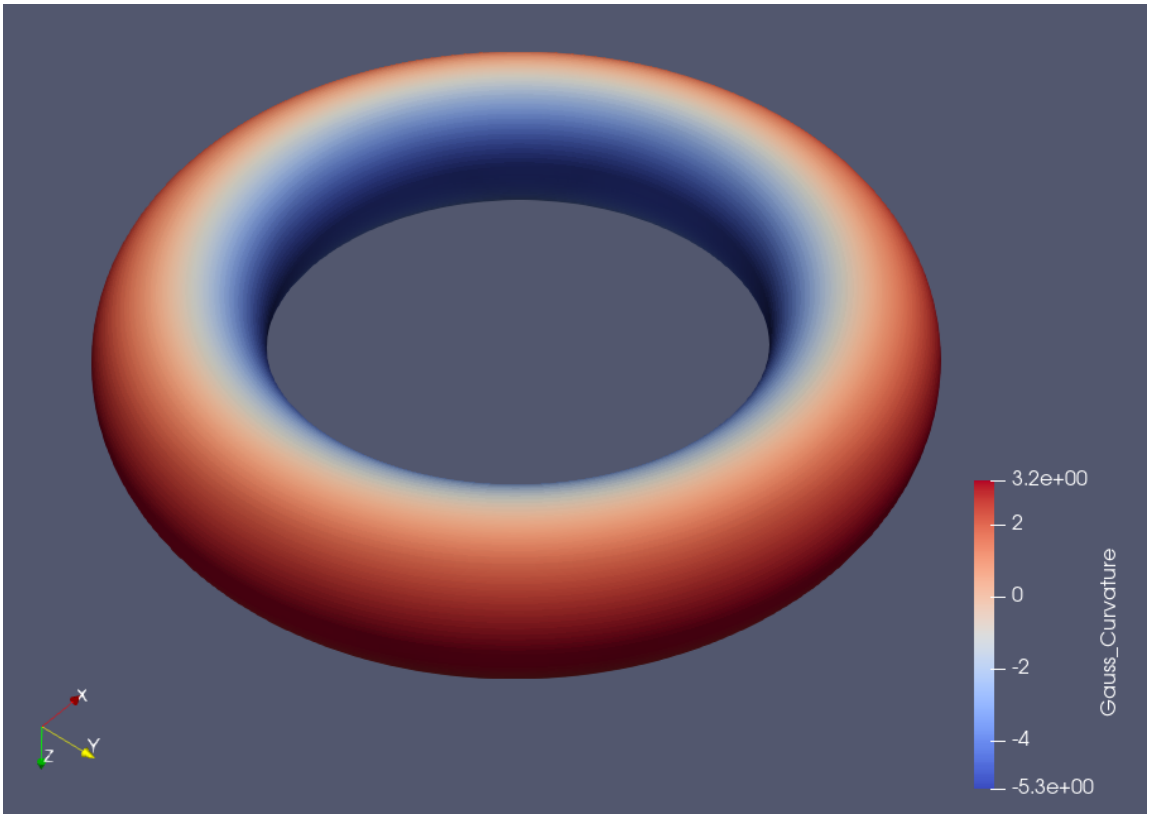

Figure 8.10: Torus curvature heat map viewed in ParaView. The elements are shaded on a gradient from blue to red as the values of Gaussian curvature go from negative to positive. The inner concave surface of the torus shows negative Gaussian curvatures, and the outer convex surface shows positive curvatures.

8.3.2 MDCL-TMA as input

The first attempt to use curvature analysis to classify the presence of rod structures in biomolecules was performed on MDCL-TMA meshes. As MDCL-TMA produces meshes of a consistent geometry, it was thought that it therefore may be easy for a machine learning algorithm to understand this geometry. The data used in the previous attempt (§8.2) of electron density maps from EMDB acts as the base data for these experiments, except they are now processed into MDCL-TMA meshes using the operations described in §7.1. The VTK files of MDCL-TMA meshes produced are used as input as that is the format the data parser can read. Table B.1 shows what biomolecules were used and their isolevel supplied as the threshold. If this method is viable, each biomolecule will be meshed to a few different resolutions (depending on the range of suitable resolutions for each object) in order to get an expanded dataset.

To test whether or not this method would be viable, scatter plots of mesh curvatures were graphed to see if discernable clusters formed for the two classes of biomolecule. Figures 8.11 and 8.12 show these plots on a sample of MDCL-TMA meshes of biomolecules.

As can be seen, there are no discernable clusters of curvatures for each class of biomolecule. This would make it difficult for a machine learning algorithm to understand the differences between classes to predict labels, and therefore using MDCL-TMA meshes as input data seems implausible.

There are two potential reasons this input format fails as data for this classification problem. The first is that MDCL-TMA meshes will have comparatively lower resolutions than an isosurface created from the point cloud data. As seen in the validation of this method of curvature analysis (§8.3.1), the test shapes produced far less-accurate convergence results at lower resolutions. The second is that due to the geometry of splitting cubes diagonally into tetrahedra, the surface of many MDCL-TMA meshes will be “jagged”, and the curvatures of each up and down spike across the surface will cancel each other out. It also makes it difficult for curvatures to discern between the actual edges of an object and a spike in the surface (see Figure 8.13 for an illustrated example), making the detection of rod structures difficult.

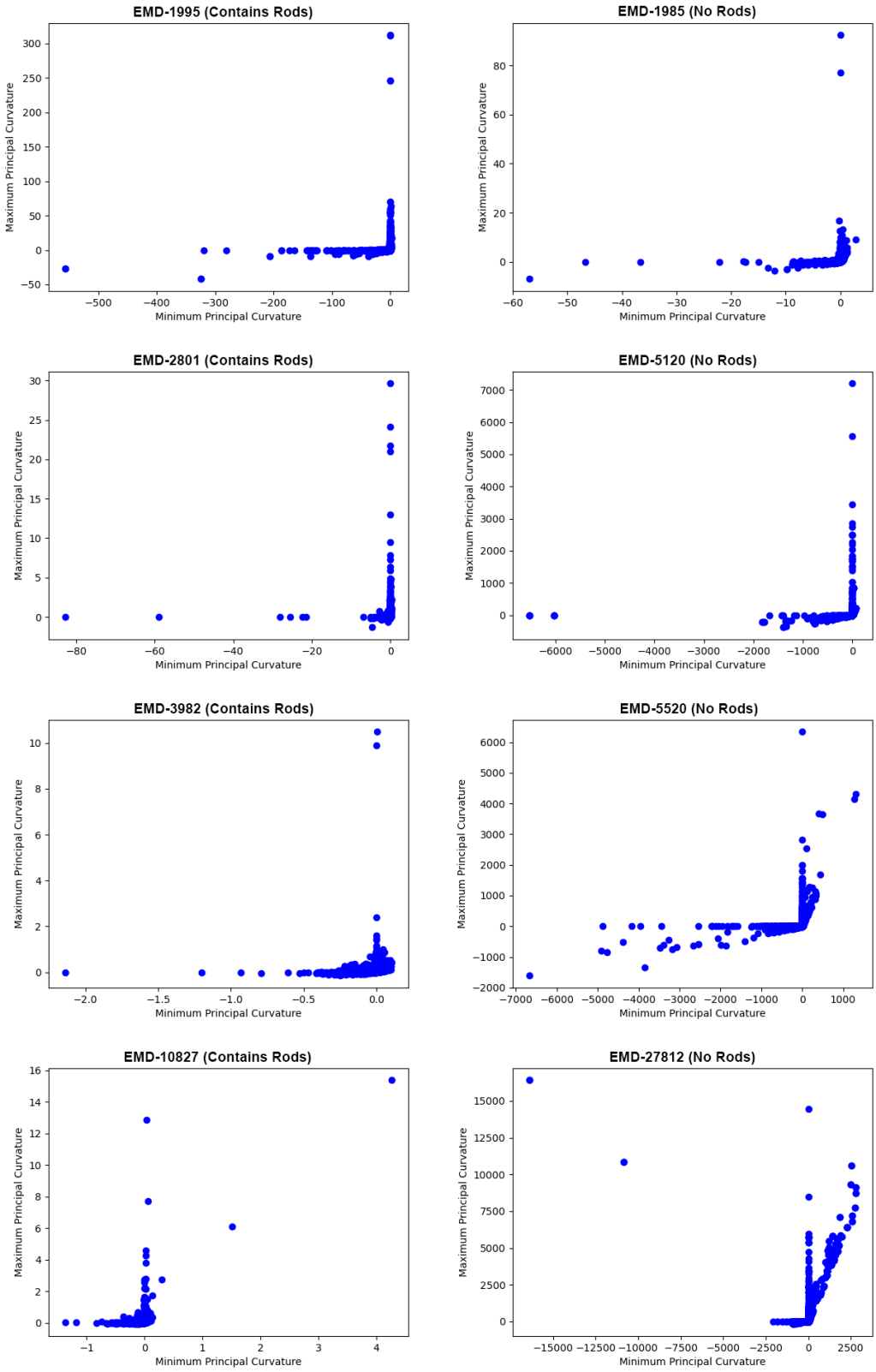

Figure 8.11: Scatter plots of MDCL-TMA mesh principal curvatures. Graphs in the left column are for biomolecules containing rod structures, and graphs in the right column are for biomolecules containing no rods. There is little difference between the principal curvature clusters created for meshes of both classes, and classifying through visual analysis would be difficult.

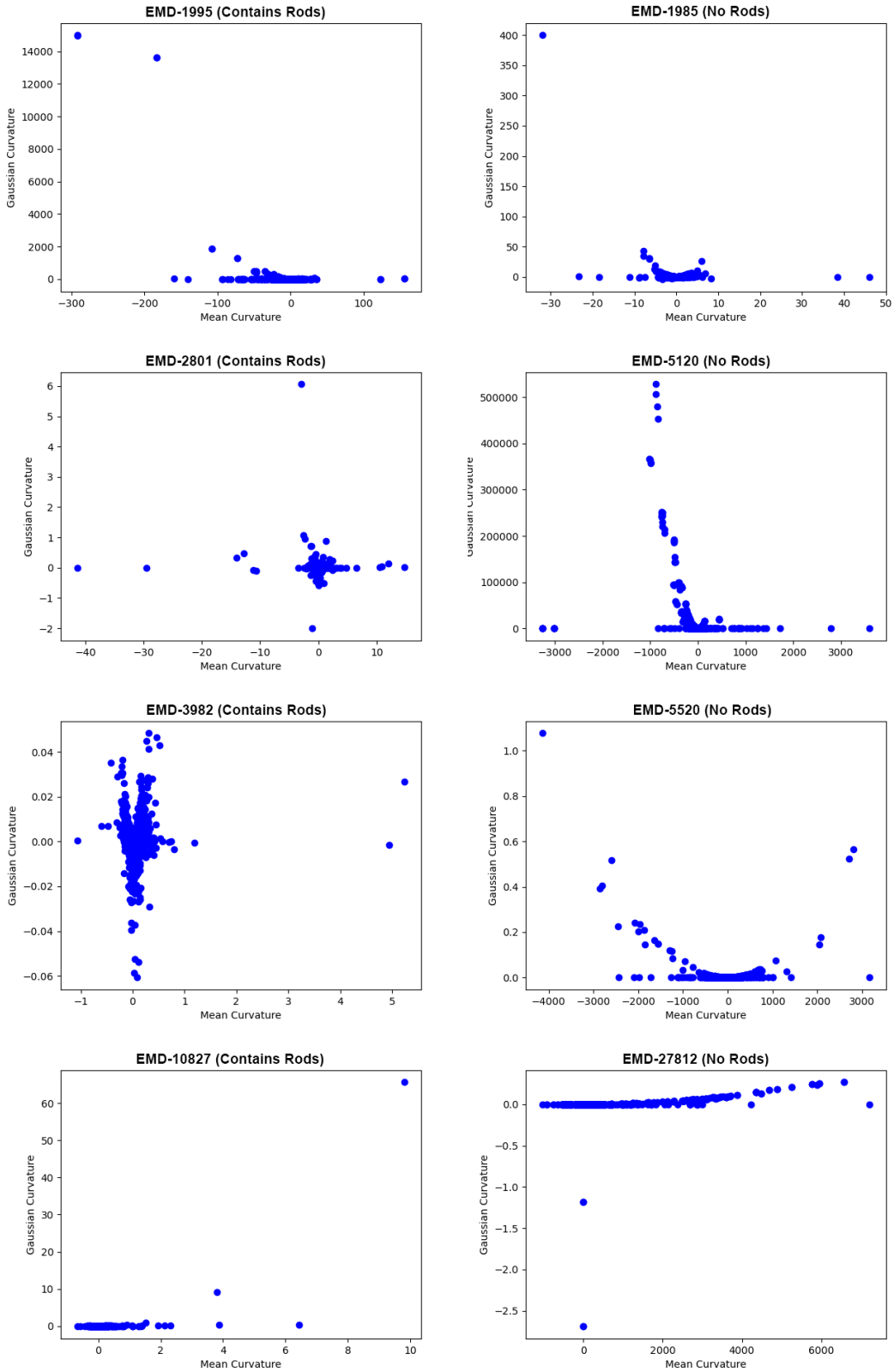

Figure 8.12: Scatter plots of MDCL-TMA mesh mean and Gaussian curvatures. Graphs in the left column are for biomolecules containing rod structures, and graphs in the right column are for biomolecules containing no rods. While there are some differences between mean and Gaussian curvature clusters created for the classes of meshes, there is too much overlap in some shapes of clusters, which would make classifying through visual analysis still a difficult task.

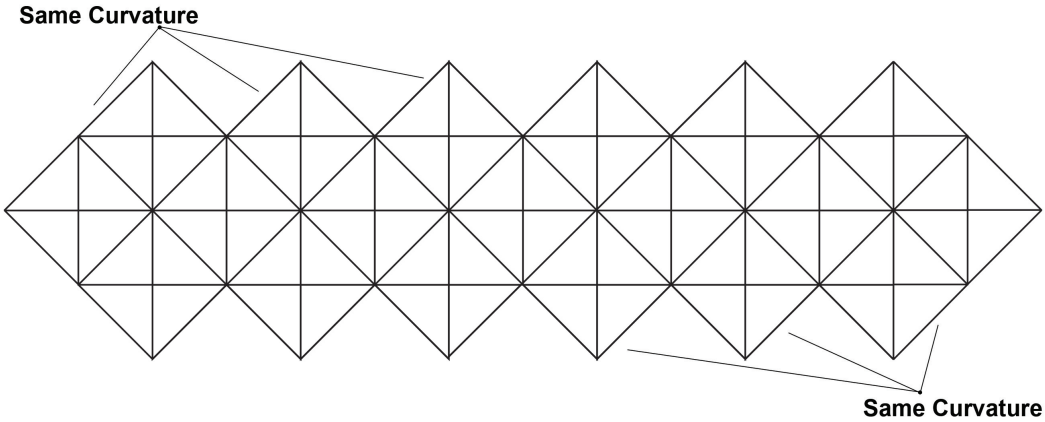

Figure 8.13: 2D example of MDCL-TMA mesh showcasing issue with curvature analysis. Due to the way MDCL-TMA produces regular geometries, the curvature of a tetrahedra edge at the end of a shape can have the same curvature of a tetrahedra edge in the middle of the surface.

8.3.3 Isosurfaces as input

Knowing that MDCL-TMA meshes are unsuitable to use as input to this problem, and that a possible reason for this was the relatively low resolution of the meshes, the next attempt uses meshes of isosurfaces of biomolecules in order to obtain meshes with a maximal resolution.

Isosurface meshes were obtained from electron density `.map` files from EMDB by processing them with the FFEAtools command `emmaptosurf`, which will output a `.obj` file containing mesh data of the biomolecule at the user-specified isolevel. Appendix B states the isolevel used for each electron density map file. The `.obj` file is then opened in ParaView and exported to the VTK format for the data parser to read.

Data features and normalisation

The data parser extracts curvature and other geometric data features from the VTK isosurface meshes. This data, alongside a binary label to state whether or not the biomolecule contained rod structures, was written to a `.csv` file. The `.csv` file's rows were then shuffled randomly to ensure no ordering bias could impact the results, and this shuffled `.csv` file could then be parsed by the machine learning algorithms.

The following data features are used as input for the machine learning algorithms to parse:

- The mean values of the biomolecule isosurface mesh's maximum principle curvature, minimum principle curvature, mean curvature, and Gaussian curvature at each node in the

mesh.

- The x, y, and z dimensions of the bounding box of the isosurface, normalised by setting the largest dimension equal to 1 and scaling the other two dimensions accordingly.
- The volume and the surface area of the bounding box based the normalised values.
- The volume and the surface area of the biomolecule isosurface mesh based the normalised values.
- A binary label stating whether or not the biomolecule contains rod structures determined by manual visual inspection.

The Python VTK library contains functions that extracts the values of mean and Gaussian curvatures at each node in a mesh, which can be stored in arrays. From this, the minimum and maximum principal curvatures can be calculated with Equations 8.5 and 8.6 and also stored in arrays. These arrays cannot be easily compared between different meshes, as they will contain different numbers of nodes and thus produces arrays of different lengths. This is why they are reduced to single comparable figures by obtaining an average for each type of curvature. This should retain most of the information needed, as if the curvatures are skewed in a particular direction (implying the presence of cylindrical geometries), the mean will also be skewed towards that direction.

The dimensions of the bounding box are used to detect obvious rod cases. If a bounding box has one of their dimensions much larger than the other two (i.e. the bounding box is an elongated rectangular prism), then the molecule inside is almost certainly some kind of elongated shape, and thus very likely to be a rod. A bounding box that is closer to a cube implies a more globular molecule inside, but it will not be able to determine from that alone if the molecule is a blob or a hybrid of rods and blobs.

The volume of the bounding box and the mesh can be compared to get an indication of the mesh's geometry. A more globular molecule is likely to fill out more of the space inside the bounding box, so a smaller ratio between the two volumes will imply a blob structure. If a molecule contains a rod structure, such as ones that extrude out of a main body, or ones that result from topological holes in the molecule's structure, there will be more "empty space" inside of the bounding box. With more empty space in the bounding box, there will be a larger ratio between the two volumes, so that larger ratio can be used to imply that a molecule contains

rod structures. Figure 8.14 illustrates this concept.

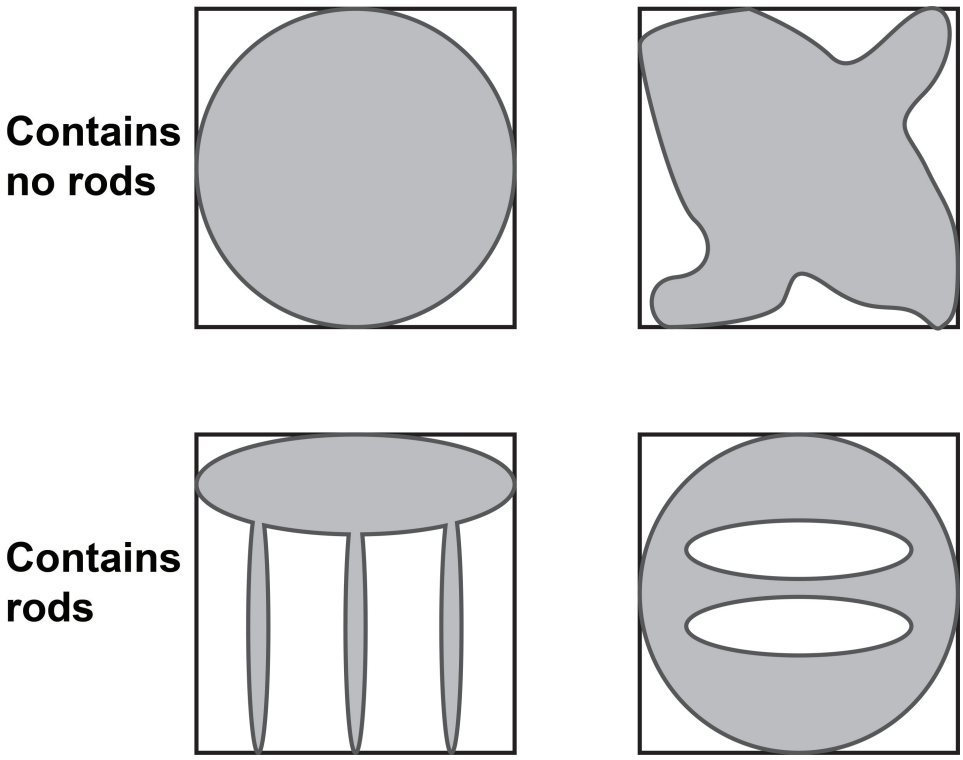

Figure 8.14: 2D examples of how different molecule structures fill the volume of a bounding box. Molecules that contain no rod structures fill out a large amount of the volume of the bounding box, whereas molecules that contain rods create more "empty space" in the bounding box due to protruding rods taking up a small volume of space, or by creating topological holes in the structure of the molecule.

The surface area of the bounding box and mesh can similarly be compared. For cases where the biomolecule is not overtly a rod due to the bounding box being an elongated prism, the boundary box will have the same kind of geometric properties as a blob structure, and this includes its surface area. For more-globular shapes, the surface area will be relatively small. However, if a molecule has rod structures that extrude from the main body or result from topological holes in the structure, this will increase the surface area of the object. Therefore, if the surface area of the molecule is close to the the value of the surface area of the bounding box, that will imply that the molecule is a blob structure. However, if the surface area of the molecule is much larger than the surface area of the bounding box, that will imply that the molecule contains rod structures.

In order to make it easier for the machine learning algorithms to parse information and compare the results of different molecules, it is necessary to normalise some of this data by constraining them to a set range of values. Normalisation of the data begins with the dimensions of the

bounding box. The largest dimension is set to equal 1, so that the normalised bounding box dimensions for all biomolecules in the dataset will be in the range of 0-1. The unknown normalised values can be calculated by determining a scale factor from known normalised values with

$$\text{Normalisation Scale Factor} = \frac{\text{Normalised Value}}{\text{Real Value}}. \tag{8.8}$$

For the dimensions of the bounding box, a normalisation scale factor can be obtained from the largest dimension normalised to equal 1 by dividing the normalised value of 1 by the real length value of the dimension. The values of the other two dimensions can then be calculated by multiplying their real length values by this normalisation scale factor.

The Python VTK library contains functions that can extract the volume and surface area of a mesh, but these values also need to be normalised. As it is known what the real and normalised dimensions of the bounding box are, the real and normalised volume and surface area of the bounding box can be calculated, and then the normalised values can be divided by the real values (as per Equation 8.8) to get normalisation scale factors each for the volume and surface area. The real values of the biomolecule isosurface mesh's volume and surface area can then be multiplied by the relevant scale factors to obtain normalised values.

While cases have been made for the inclusion of all data features described in this subsection, there is a perfect correlation between the dimensions of the bounding box, and the volume and surface area of the bounding box. This means that potentially, to a machine learning algorithm, some of these data features are redundant and may hinder the ability of the algorithms to classify data by being confused by too much information. Therefore, experiments are performed in three sets; one where all data features are included, one where the three dimensions of the bounding box are removed as data features, and one where the volume and surface area of the bounding box are removed as data features. This will allow for comparison to see if removing one set of potentially-redundant information improves the cross-validation scores of the models.

Results

Table 8.3 shows the results of attempting to classify the presence of rods in biomolecule isosurfaces using the machine learning algorithms discussed in §8.1. For the k-fold cross-validation, $k_{cv} = 5$. Cross-validation scores range from 0-1, with 0 meaning the algorithm is never cor-

rect, and 1 meaning the algorithm is always correct. This set of experiments used isosurfaces created from the data shown in Appendix B in Table B.1, which contains 30 biomolecules that contain rod structures, and 33 biomolecules that contain no rod structures, for a total of 63 biomolecules. The appendix also states the isolevels used for each biomolecule.

Table 8.3: K-fold cross-validation scores for rod classification from biomolecule isosurfaces where $k_{cv} = 5$. These experiments use the dataset described in Table B.1. The first set of results used all data features available. The second set removes the bounding box dimensions as data features. The third set removes the volume and surface area of the bounding box as data features. The dataset is in the same shuffled order for each set of experiments so that the only difference between them are the data features used.

Algorithm	**K-Fold Cross-Validation Score**					
	1	**2**	**3**	**4**	**5**	**Mean**
All data features						
k-NN ($k = 7$)	1	$0.\dot{3}$	$0.\dot{6}$	1	1	0.8
Decision Trees	$0.\dot{3}$	1	$0.\dot{6}$	0.5	0.5	0.6
SVM - Linear	1	1	$0.\dot{6}$	1	1	$0.9\dot{3}$
SVM - Polynomial	$0.\dot{6}$	$0.\dot{3}$	$0.\dot{6}$	1	1	$0.7\dot{3}$
SVM - RBF	1	$0.\dot{6}$	$0.\dot{6}$	1	1	$0.8\dot{6}$
SVM - Sigmoid	1	1	$0.\dot{3}$	1	0.5	$0.7\dot{6}$
No bounding box dimensions						
k-NN ($k = 2$)	1	1	$0.\dot{3}$	1	1	$0.8\dot{6}$
Decision Trees	1	1	$0.\dot{6}$	1	1	$0.9\dot{3}$
SVM - Linear	1	1	$0.\dot{6}$	1	1	$0.9\dot{3}$
SVM - Polynomial	$0.\dot{6}$	$0.\dot{3}$	$0.\dot{6}$	1	1	$0.7\dot{3}$
SVM - RBF	1	$0.\dot{6}$	$0.\dot{6}$	1	1	$0.8\dot{6}$
SVM - Sigmoid	1	1	$0.\dot{6}$	1	0.5	$0.8\dot{3}$
No bounding box volume and surface area						
k-NN ($k = 4$)	1	1	$0.\dot{6}$	1	1	$0.9\dot{3}$
Decision Trees	$0.\dot{3}$	1	$0.\dot{6}$	0.5	0.5	0.6
SVM - Linear	1	1	$0.\dot{6}$	1	1	$0.9\dot{3}$
SVM - Polynomial	$0.\dot{6}$	$0.\dot{3}$	$0.\dot{3}$	0.5	1	$0.5\dot{6}$
SVM - RBF	1	$0.\dot{6}$	$0.\dot{6}$	1	1	$0.8\dot{6}$
SVM - Sigmoid	1	1	$0.\dot{6}$	1	1	$0.9\dot{3}$

The results show much improvement from the results obtained earlier in Table 8.1. Unlike earlier, all algorithms performed better than random chance (a cross-validation score of 0.5). Removing one group of data features almost-universally either improves mean cross-validation scores or has no effect on them, with the only exception being SVM with a polynomial kernel, with its mean cross-validation score dropping from $0.7\dot{3}$ to $0.5\dot{6}$ when removing the volume and surface area of the bounding box as data features. SVM with a linear kernel using all data

features, decision trees and SVM with a linear kernel removing the bounding box dimensions as data features, and k-NN and SVM with linear and sigmoid kernels removing the bounding box volume and surface area as data features, all obtained the best mean cross-validation score of $0.9\dot{3}$.

While the results have shown improvement in the accuracy of classification in comparison to attempting to analyse the electron density map point cloud data, the reliability of the results is questionable. Many of the algorithms show wide ranges in cross-validation scores between runs, the widest range being scores of $0.\dot{3}$ to 1 recorded in multiple cases. This calls into question how well these algorithms will actually perform with future unclassified data. This unreliability is likely the result of using a small dataset due to the time cost of obtaining and processing biomolecule data. In order to obtain more reliable results, time was spent expanding the size of the dataset.

Table 8.4 shows the results of attempting to classify the presence of rods in biomolecule isosurfaces with a larger dataset. This set of experiments used isosurfaces created from the data shown in Appendix B in both Tables B.1 and B.2. The combined datasets contain 107 biomolecules that contain rod structures, and 107 biomolecules that contain no rod structures, for a total of 214 biomolecules. The k-fold cross-validation still uses $k_{cv} = 5$.

While the recorded mean cross-validation scores are a bit lower than those obtained in Table 8.3, there is a much tighter range of scores between runs, with the widest range of scores now being $0.\dot{4}$ to $0.\dot{7}$ for SVM with a sigmoid kernel where the dimensions of the bounding box are removed as data features. This implies that these results are more reliable for predicting how well future unclassified data would be classified by these algorithms.

Compared to the previous set of experiments, the results for this set of experiments shows mixed results for removing one group of data features. k-NN was found to perform worse when removing any data features, decision trees showed an improved mean cross-validation score when removing bounding box dimensions as data features, SVM with a linear kernel showed improved scores when removing the bounding box surface area and volume but worse scores when removing dimensions, SVM with a polynomial kernel performed worse when removing data features, SVM with an RBF kernel showed improved scores when removing the bounding box surface area and volume, and finally SVM with a sigmoid kernel performed very slightly worse when removing the bounding box dimensions but performed better when removing the

bounding box volume and surface area. This shows that the selection of data features should be tailored to fine-tuning the machine learning algorithm being used.

Table 8.4: K-fold cross-validation scores for rod classification from biomolecule isosurfaces where $k_{cv} = 5$. These experiments use the datasets described in Tables B.1 and B.2 combined. The first set of results used all data features available. The second set removes the bounding box dimensions as data features. The third set removes the volume and surface area of the bounding box as data features. The dataset is in the same shuffled order for each set of experiments so that the only difference between them are the data features used.

Algorithm	**K-Fold Cross-Validation Score**					
	1	**2**	**3**	**4**	**5**	**Mean**
All data features						
k-NN ($k = 6$)	$0.\dot{7}$	$0.\dot{7}$	$0.\dot{6}$	0.875	0.75	0.769
Decision Trees	$0.\dot{8}$	$0.\dot{7}$	$0.\dot{7}$	1	0.75	$0.83\dot{8}$
SVM - Linear	$0.\dot{8}$	$0.\dot{6}$	$0.\dot{5}$	0.875	0.75	0.747
SVM - Polynomial	$0.\dot{6}$	$0.\dot{6}$	$0.\dot{5}$	0.75	0.75	$0.6\dot{7}$
SVM - RBF	$0.\dot{6}$	$0.\dot{6}$	$0.\dot{6}$	0.875	0.625	0.7
SVM - Sigmoid	$0.\dot{6}$	$0.\dot{4}$	$0.\dot{4}$	0.75	0.625	0.586
No bounding box dimensions						
k-NN ($k = 2$)	$0.\dot{8}$	$0.\dot{6}$	$0.\dot{5}$	0.875	0.75	0.747
Decision Trees	1	$0.\dot{8}$	$0.\dot{6}$	0.875	0.875	$0.86\dot{1}$
SVM - Linear	$0.\dot{6}$	$0.\dot{5}$	$0.\dot{6}$	0.875	0.625	$0.6\dot{7}$
SVM - Polynomial	$0.\dot{6}$	$0.\dot{6}$	$0.\dot{5}$	0.75	0.625	0.653
SVM - RBF	$0.\dot{6}$	$0.\dot{6}$	$0.\dot{6}$	0.875	0.625	0.7
SVM - Sigmoid	$0.\dot{7}$	$0.\dot{4}$	$0.\dot{4}$	0.625	0.625	$0.58\dot{3}$
No bounding box volume and surface area						
k-NN ($k = 3$)	$0.\dot{8}$	$0.\dot{6}$	$0.\dot{5}$	0.875	0.75	0.747
Decision Trees	$0.\dot{8}$	$0.\dot{7}$	$0.\dot{7}$	1	0.75	$0.83\dot{8}$
SVM - Linear	$0.\dot{8}$	$0.\dot{7}$	$0.\dot{6}$	0.875	0.75	0.792
SVM - Polynomial	$0.\dot{5}$	$0.\dot{5}$	$0.\dot{6}$	0.625	0.375	$0.\dot{5}$
SVM - RBF	$0.\dot{7}$	$0.\dot{7}$	$0.\dot{6}$	0.875	0.75	0.769
SVM - Sigmoid	$0.\dot{8}$	$0.\dot{6}$	$0.\dot{6}$	0.625	0.625	$0.69\dot{4}$

The best mean cross-validation score obtained in these sets of experiments was $0.86\dot{1}$ for decision trees with the dimensions of the bounding box removed as data features. This is only a slight improvement of the score of $0.83\dot{8}$ obtained by decision trees in the other two sets, so when it comes to maximising cross-validation scores, the choice of algorithm is more important than the choice of these data features.

While additional work is likely warranted to further improve the accuracy of classifying this problem, these results prove the concept that machine learning can be used to predict the

presence of rod structures in biomolecules. Based on these preliminary results, it appears that the decision trees method is the most suitable classification method for this problem, and therefore future work should likely be focused on using this algorithm and tailoring the dataset and the parsing of the data to it.

8.4 Chapter conclusions and future prospects

In this chapter, multiple attempts were made to try and use supervised machine learning classification algorithms to understand the geometry of biomolecule data and classify them into two categories; biomolecules that contain rod structures, and biomolecules that contain no rod structures.

The first attempt had machine learning algorithms try to learn from the original point cloud data of electron density maps that describe these biomolecule structures. This attempt failed however, with all algorithms tested performing worse than random chance. This happened because the machine learning algorithms could not understand the geometry of biomolecule structures from the header information contained in electron density map files, and it was also unable to parse useful information from the unstructured data. Attempts were made to use attributes of the unstructured data in the hope that the machine learning algorithms could derive useful information, but this method did not work.

The next attempts tried to categorise biomolecules through analysing the curvature of the surface geometry of meshes made from electron density map files. As different types of shapes experience different curvatures, the idea was that machine learning algorithms would be able to understand geometries with this information and be able to detect rod structures.

The first attempt that used this method used MDCL-TMA meshes as input, with the idea being that as MDCL-TMA meshes produce a consistent geometry, a machine learning algorithm may find its geometry easy to understand and discern between categories. This was not the case however, as analysis of curvature plots showed little distinction between the two classes of biomolecules. This was potentially due to MDCL-TMA meshes being too low-resolution, as during validation of this method with test shapes it was noted that low-resolution meshes produce less-accurate convergence results. Another possible reason is that the “jagged” surfaces of MDCL-TMA meshes produce many areas that experience the same curvatures as the far edges

of the shape, making the information it conveys indiscernible.

The next attempt tried to counter the problems experienced when using MDCL-TMA meshes by using high-resolution isosurface meshes as input. This yielded much better results when classifying with machine learning algorithms, with decision trees performing the best when the dimensions of the bounding box are removed as data features, producing a 5-fold cross-validation score of 0.86$\dot{1}$. These results prove the concept that machine learning can be used to predict the presence of rod structures in biomolecule geometry.

There are potential ways that future research could improve on these results. The simplest method would be to increase the size of the dataset, as time constraints prevented the collection and processing of further data. The results shown in Tables 8.3 and 8.4 show the significance of the size of the dataset to obtaining reliable results, with the larger dataset producing results with much smaller ranges of cross-validation scores between runs. More care could also be taken to ensure processed data is free from noise, as noise can impact the dimensions of the bounding box, potentially skewing some results. There is also a subjective nature to the human categorisation of biomolecules into those that contain rod structures and those that do not. Whether or not an FFEA user wishes to represent a region with a KOBRA rod can be dependant on what exactly the user wishes to analyse about the biomolecule in question. As only a single researcher categorised the data in these experiments, there is a potential for bias in the dataset. This could be alleviated by having multiple people label the dataset to gain consensus on how each biomolecule should be categorised.

Different algorithms could also be attempted to see if decision trees is truly the best-fitting model for this classification problem. An artificial neural network, such as a multi-layer perceptron, a convolutional neural network, or deep learning could be tried, as they all have been previously successfully implemented for biological image classification problems (Affonso et al. 2017). Alternatively, the method of analysing electron density map geometries could be re-attempted by using a more specific method of point cloud classification, with tools such as PointNet, PointHop, and PointCNN.

The next step in trying to fully automate the generation of KOBRA rods in FFEA systems is to design an algorithm that can find the location(s) of a rod structure(s) in a biomolecule that is confirmed to contain rod structures. There are existing computer vision techniques for 3D shape feature extraction that could be employed for this task (Mari et al. 2019). One such

technique is feature line extraction, which attempts to detect lines on the object surface that represent prominent geometric features, such as lines of curvature, and crest and ridge lines that describe surface topography. Another approach is mesh segmentation, which attempts to separate areas of a geometry into their own regions according to a given criterion (such as the curvature). Graphs that describe shapes can also be plotted and analysed to determine regional geometry. All these approaches can build off the existing work done here through utilising the extracted curvature information from biomolecules, and therefore are promising methods to start with for attempting this next step in automating KOBRA generation.

Once a method has been devised to locate rod structures in biomolecules, this information needs to be fed into the FFEA mesh generation process to replace designated rod regions with KOBRA rod structures. This will include needing to set up any connections to other structures in the FFEA system. If successful, this will hopefully give FFEA users a requested feature as stated by participants in the qualitative survey in Chapter 4, and improve the ease-of-use of setting up experiments in FFEA. If experiment setup can be made easier, this should make further improvements to the user-experience of FFEA, and thus encourage a better user-retention amongst the userbase, increasing the longevity of the software. If this feature is completed, it is recommended that it be rolled out to the public and then another qualitative survey be conducted to measure the impact it would have on the user-experience, maintainability, and longevity of FFEA.

Chapter 9

Conclusions and future prospects

In this thesis, the concept of sustainable practices for research software engineering (RSE) has been studied and applied to the molecular dynamics method Fluctuating Finite Element Analysis (FFEA) and its software suite. The specific goal of the case study was to analyse FFEA through the lens of these sustainable RSE practices to determine the true bottlenecks in the workflow of the software, and to find solutions to these problems in order to prevent FFEA from experiencing software decay and low user-retention, so that in the future no one would have to waste resources on reinventing such a tool. The general goal of this research was to test the methodology as outlined in §1.2 to see if it is an effective strategy for adapting the sustainable qualities of research software to improve its user experience, maintainability, and longevity. This chapter summarises the progress that has been made towards these goals, and discusses future prospects to potentially further improve upon the research conducted here. This makes up the final step in our general methodology as outlined by Figure 1.1, "Draw conclusions".

9.1 Summary of progress

Chapter 2 of this thesis explored the field of research software engineering, and its practices to design, develop and maintain sustainable software. The field of RSE recognises a systemic problem with the nature of producing software for research purposes, in that this goal-oriented software often has minimal attention put into ensuring the software itself is of good quality and is *maintained* in good quality. This problem is also furthered by the developers of research

software often being experts in their own field, *not* in software engineering, with little-to-no training in software development. This leads to much of this software rapidly decaying, meaning that research software tools are constantly having to be remade for the same or similar purposes, wasting the time and resources of researchers.

It is undesirable for this threat of software decay to occur in FFEA, which has seen over a decade of work from a large team of contributors and an establishment of a community of users (and even had it not yet, it would still be good practice to ensure its sustainability). FFEA, however, is not without its issues in terms of its performance and user experience. As a computer scientist, it would have been tempting to try and solve this by optimising the FFEA source code and improve the computational efficiency, which may have had a significant impact on the performance, however this is not the only computational solution to try and improve molecular dynamics software. Time and resources could have been spent on trying to simply "make the software run better" when there could be much more pressing problems resulting from other aspects of the software, such as its human-computer interaction or the design of its processes. Therefore, Chapters 3 and 4 were dedicated to a thorough evaluation of the FFEA software suite in order to determine what was truly impeding the sustainability of FFEA.

Chapter 3 first analysed FFEA quantitatively, in order to determine potential problems in the performance of the FFEA runner (the executable which performs the molecular dynamics simulations). This analysis included static code analysis through linting, dynamic code analysis through running datasets of biomolecule meshes in the FFEA runner and measuring performance, and soak testing. Chapter 4 then analysed the whole FFEA software suite qualitatively through surveying recent users and developers experiences, in order to determine problems in the user experience of FFEA (and to potentially corroborate any issues with software performance found in the previous chapter).

From the results of this evaluation it was found that while there are some issues with the source code, such as poor linting scores and a potentially-significant memory leak, these did not seem to be fatal issues requiring high-priority fixes, just that they should be addressed in future maintenance. A much more significant problem became apparent when surveying users of FFEA, in that there is a huge cost-investment in the set up of experiments due to the process being difficult and time-consuming, potentially requiring months of work. This results primarily from the lengthy number of steps in generating meshes, all of which have the potential

to produce an unstable output that cannot be used in the next step, and the trial-and-error nature of parametrising FFEA systems by changing text fields, instead of having a GUI with instant feedback. This meant that the major bottleneck to the workflow of FFEA users was this simulation setup process.

Finding this key bottleneck showed that this general method of evaluating research software was successful for the specific case study of FFEA, and so provides a foundation for how to apply a holistic evaluation to other general research software. By testing this method against the specific case study in this research, it was found that there were some limitations to it, and that there are further adaptations that could be made to this evaluation method so that it can be applied to a wider range of general research software and gain further relevant information about them (see §9.2). This will help to refine the general method for future research.

In this research, two methods were developed to attempt to alleviate this bottleneck; the development of an alternative meshing algorithm, and the development of machine learning algorithms to work towards automating the generation of Kirchoff Biological Rod Algorithm (KOBRA) meshes.

9.1.1 Marching diamond crystal lattice tetrahedral meshing algorithm

Chapter 5 discussed an alternative method of meshing for FFEA that was began in the work of Gravett (2022). Their research began the development of an algorithm to process 3D image data of biomolecules in voxel format into tetrahedral meshes of a consistent geometry. This was done by applying the marching diamond crystal lattice algorithm, which diagonally slices voxels into five tetrahedra (four trirectangular and one regular), with the inner tetrahedron alternating directions between neighbouring voxels to produce a more isotropic mesh with more uniform material properties. This algorithm was successfully implemented for their research, however it was not in a complete state to be used by a wider audience. There were missing quality of life features, such as having to manually pre-process data via coarse-graining externally, and there were missing features from the algorithm itself, as at the time the algorithm could not select for specific tetrahedra, meaning all meshes were restricted to producing cubic voxel shapes.

Chapter 6 detailed the development work performed on the new meshing algorithm as part of this thesis. For ease of reference, the new meshing algorithm was given the tentative name of the "Marching Diamond Crystal Lattice Tetrahedral Meshing Algorithm" (MDCL-TMA). Several

improvements to the usability of the algorithm were made here by adding an internal coarse-graining function, a basic stability check of output meshes, and adding command-line options for user-input variables (as opposed to having to make changes in-script). The capabilities of MDCL-TMA were improved by developing an interpolation scheme to smooth the mesh surface by applying the density threshold that selects which mesh elements to include in the final output to every tetrahedron, rather than only to the voxels. This resulted in more accurate surface geometry, and therefore a more accurate distribution of the biomolecule's density and the results of the physics applied to the model.

Chapter 7 then performed comparative analysis on the MDCL-TMA prototype against the original FFEAtools method of meshing to see if MDCL-TMA improves upon the capability and stability of meshes for FFEA systems. This analysis was conducted through comparing the quality of the static meshes produced by each algorithm, comparing the performance of FFEA simulations with each type of mesh, and comparing the results of FFEA simulations with each type of mesh.

A clear result from this comparative analysis was that the default workflow for MDCL-TMA meshing had a higher success rate for producing meshes than the FFEAtools default workflow. All eleven objects used as a dataset for these experiments were able to be meshed in MDCL-TMA. The original FFEAtools meshing algorithm, however, was only able to create two meshes from these eleven objects. The consistent geometry of MDCL-TMA meshes, alongside the relatively fewer steps and parameters in the process compared to FFEAtools, appears to make MDCL-TMA a more user-friendly process for generating meshes for FFEA systems. Future work should include analysing the experiences of others using MDCL-TMA to see if this holds true.

The meshes produced by MDCL-TMA measured very consistent quality metrics of almost exactly the same values between different biomolecules. Meshes produced by FFEAtools did not measure consistent quality metrics. The consistency of MDCL-TMA geometry likely means the algorithm will almost always produce meshes of almost the same quality, corroborated by the mathematical proofs shown in §7.1.1. MDCL-TMA meshes were also recorded to perform better and more reliably in FFEA simulations than FFEAtools meshes, with MDCL-TMA meshes resulting in less simulation-terminating element inversion errors and achieving higher wall clock times. This suggests that MDCL-TMA produces more stable meshes than FFEAtools, however

a user must still take some care in parametrisation to prevent element inversion errors, such as using appropriate coarse-graining values and simulation time steps.

From comparing the FFEA simulation results using both MDCL-TMA and FFEAtools meshes, it was found that MDCL-TMA does not negatively impact strain energy calculations, however an MDCL-TMA mesh may experience a higher strain energy over time compared to FFEAtools meshes of the same coarse-graining value, likely due to MDCL-TMA meshes of the same coarse-graining value containing a larger number of tetrahedra than FFEAtools meshes. This was tested by comparing an MDCL-TMA mesh with a similar number of tetrahedra as an FFEAtools mesh of the same biomolecule (with different coarse-graining values), which now recorded similar values of strain energy. MDCL-TMA meshes were also found to have their centroid deviate less than FFEAtools meshes of the same coarse-graining value in FFEA simulations, but again meshes of the same biomolecule with similar numbers of tetrahedra produced similar results in centroid deviation. Future work is required in order to thoroughly test and determine the relationships between mesh geometry and recorded physical properties during simulation, as other factors may be impacting these relationships, such as the surface area of the mesh.

A key difference between MDCL-TMA meshes and FFEAtools meshes is that MDCL-TMA meshes contain internal elements, whereas FFEAtools produces surface meshes. This appears to cause discrepancies in the values of RMSD, which FFEA simulations use to indicate the magnitude of fluctuations. It is currently unknown whether or not this is problematic, as this would require some expert input from FFEA users. There is at least one known case of obtaining useful results with an earlier version of MDCL-TMA that still produced internal elements (Gravett 2022), so this may not be an issue, or at least not a universal one. MDCL-TMA could easily be adapted in the future to produce surface meshes by removing internal elements, or to give a user the option to do so.

From the analysis performed so far, it is believed here that MDCL-TMA meshes behave as expected in FFEA simulations and produces valid results, with discrepancies between simulation results appearing to be the result of geometric differences between the meshing algorithms, not the result of MDCL-TMA introducing faulty calculations into the FFEA runner. FFEA users may need to adjust their workflow to use MDCL-TMA over FFEAtools for meshing, but at the advantage of greatly reducing the complexity of FFEA mesh generation. This hopefully makes for an easier user experience that reduces the time cost of setting up FFEA experiments.

With an easier, less time-consuming workflow, this should make it easier to teach new users how to operate FFEA, and give existing FFEA users a much-requested reduction in the number of manual trial-and-error steps needed for mesh generation. All this will help to improve user retention, thus increasing the longevity of the software to make it more sustainable. Currently, the source code of the MDCL-TMA Python prototype (Gravett and Rogers 2023) is forked from the main repository for FFEA meshing (Gravett, Rogers et al. 2023). Future work is required to integrate it into the main development path.

Through the successful development of MDCL-TMA, it has been shown that the general methodology of this research as outlined in §1.2 has successfully lead to new functionality which shows improvement to the research software's sustainable qualities for this specific case of FFEA. While future work is required to fully test MDCL-TMA's impact on FFEA's sutainable qualities (see §9.3), this research shows that the general method has promise for leading development of research software to understanding what relevant and effective changes and additions can be made to the software to improve the user experience.

9.1.2 Machine learning for classifying biomolecule geometry

The automation of the meshing process for FFEA has been a desired feature of FFEA users, as seen in Chapter 4. Gravett (2022) made note in the future perspectives in their thesis that adapting the new meshing algorithm to generate KOBRA rods and combined rod-blob models would be a useful feature, and that this automation may require machine learning. The work in Chapter 8 built towards this automation through the design and development of a classification algorithm to predict whether or not a biomolecule contains rod structures that should use KOBRA representation.

Three different supervised machine learning classification algorithms were tried for this problem; k-nearest neighbours, decision trees, and support vector machines. There were multiple attempts made at using these algorithms to classify a dataset of biomolecules into those that contain rod structures and those that do not.

The first method attempted to classify biomolecule data through machine learning algorithms trying to learn from the original point cloud data of electron density maps in MRC format. This attempt resulted in all algorithms performing worse than random chance, and thus was a failure. This failure resulted from the machine learning algorithms being unable to understand

biomolecule geometry from the header information stored in electron density map files, and being unable to parse useful information from the unstructured geometric data.

The second method attempted to classify biomolecules through analysing the curvature of the surfaces of meshes generated from electron density map files. A first attempt with this method used MDCL-TMA meshes as input, hypothesising that the consistent geometry of MDCL-TMA meshes would be easy for machine learning algorithms to comprehend. This turned out to be incorrect, as scatter plots of biomolecule mesh curvatures showed minimal difference between the two classes of biomolecule. This was likely the result of MDCL-TMA's "jagged" surface geometry experiencing the same curvatures as far edges of the shape, making the geometric information it conveys indiscernible to machine learning algorithms. The low resolution was also a potential factor, as test cases showed the accuracy of curvature results decreasing as the resolution does.

The second attempt with the curvature analysis method used high-resolution isosurface meshes as input, in an attempt to counter the problems experience when using MDCL-TMA meshes as input. This produced better classification results, with decision trees achieving the highest 5-fold cross-validation score of $0.86\dot{1}$ when removing the dimensions of the bounding box as data features. The results of this experiment proves the concept that machine learning can be used to predict the presence of rod structures in biomolecules.

While the general method as outlined in §1.2 has once again lead to figuring out a relevant addition to FFEA that would theoretically improve its sustainable qualities, as this is only a proof of concept it cannot yet be said how this reflects on the general method, only that it has been potentially useful for the specific case of FFEA.

9.1.3 Generic applications to research software sustainability

Through applying the general methodology outlined in §1.2 to the case study of FFEA, this research has been able to test this proposed method of improving the sustainable qualities of research software through studying the software's research topic, thoroughly evaluating the current qualities of the software, and designing and developing appropriate features based on evaluation results and understanding of the research topic.

The successful production of new features for FFEA with improved sustainable qualities has shown that this methodology has potential for wider application. Had this research not per-

formed a thorough evaluation of the software from a quantitative and qualitative perspective as outlined in the general methodology, it could have been easy to make an uninformed decision on what would improve the sustainability of FFEA. For example, development could have have been focused on improving the runtime performance of FFEA based on an early finding of a memory leak in the FFEA source code (as seen in §3.2.1). Attention could have been placed solely on this and improvements may have even been made to the software by doing so, but this would have missed the much larger problem, the much larger time-losses, that came from the instability of meshes generated during setup (as explained in §4.4). One cannot hope to fix the biggest issues without understanding the big picture, and this general methodology ensures that an RSE does not get their attention stuck on a single issue.

When it came to new development of features for FFEA to improve its sustainable qualities, the general methodology provided a helpful process for guiding this development. Starting with an analysis to understand the current functionality new development aims to change or replace prevented repeating past mistakes by seeing where exactly the current functionality was causing quality issues, and also helped to prevent creating new mistakes to see what steps (if any) in the current functionality was sufficient and should be brought into the new functionality (which would help an RSE avoid the issue of fixing what is not broken). This understanding also serves as inspiration for the design of new functionality by providing a comparable example, instead of requiring an RSE to design something from scratch.

Finally, analysing the new functionality through comparative analysis against the old functionality and evaluating the limitations of the new functionality allows the RSE to clearly communicate the results of their work in order to highlight improvements made to the software with this functionality (which may be useful in situations where one has to convince the rest of the development team or a funding body of a worthwhile change), and it makes clear to those involved in future development where improvements can be made. This ensures that the next development cycle starts off well by having clear information on recent changes and future suggestions.

9.2 Limitations of this research

While this research was successful in evaluating FFEA 2.7.2 in order to determine a key bottleneck in the workflow that hindered the user experience, and while there was success in creating

a new feature for FFEA that addresses this bottleneck, improves the user experience, and thus theoretically improves the software's longevity through being easier to learn and use, there were limitations of the research conducted that prevent the general method outlined in §1.2 from reaching its full potential.

There was a missed opportunity in the methods chosen for static code analysis during the quantitative evaluation of FFEA 2.7.2 (Chapter 3). The source code was not measured for established quality metrics such as from ISO standards, which would have provided a better set of measurable qualities for general research software, and would have allowed for easier comparison between other research software and later versions of the same software. The mistake of not including this kind of analysis came from thinking too specifically about the case study of FFEA, and not enough about how to analyse general research software. Any attempt to re-conduct this research or iterate upon it should consider adding such analysis to the general method of evaluating research software.

The methodology of dynamic code analysis allowed for enough understanding of the performance of the software in this specific case study in order to determine the key bottleneck impacting the user experience, however it was not in-depth enough to pinpoint the causes of inefficiencies in the source code. The general evaluation method may benefit from slightly adapting the dynamic code analysis to a more thorough method for better understanding research software performance. This may also include additional stress testing with various resource loads to determine what loads may significantly impact the efficiency of the source code and the software performance.

The qualitative analysis (Chapter 4) was successful for gaining a holistic understanding of the user and developer experience across all aspects of FFEA, and the questions used could easily be applied to any generic research software by changing the topic software of the questions. There was however one potential gap in these questions in that they did not ask much about the participants themselves (such as their level of research experience) to determine differences in the results between different categories of participants. In this specific case as well, there was only a very small sample size of five participants due to how niche FFEA is. It would be worth performing this evaluation method on a more popular piece of research software to more thoroughly test how effective the method is, in order to see if it needs further adapting with more general questions.

While this project allowed MDCL-TMA to be developed into a feature-complete state (Chapter 6), and a thorough comparative analysis showed that MDCL-TMA produces valid meshes that are generally more stable than meshes produced by FFEAtools (Chapter 7), there is a limit to how much it can be said that MDCL-TMA has improved the user experience, maintainability, and the software longevity of FFEA as per question 3 in the research scope (§1.1). Ideally, this would have been done by rolling out MDCL-TMA publicly for FFEA users, re-conducting the evaluation as performed in Chapters 3 and 4, and then comparing the results before and after the introduction of MDCL-TMA to determine any measurable effects on FFEA's sustainability qualities. The time-constraints of this research project prevented this from occurring however. It is suggested that the FFEA development team re-perform the evaluation of FFEA (with the limitations outlined in this section considered) in order to test the sustainability of this new feature, and to consider what parts of FFEA require further work to improve its sustainable qualities.

9.3 Future prospects

While there has been enough testing of MDCL-TMA to prove the concept of the method and its effectiveness, there is still more analysis of MDCL-TMA to be done. Most importantly is a qualitative analysis through surveying users. The comparative analysis done demonstrates quantitatively a reduction in the number of steps in the meshing process for MDCL-TMA compared to FFEAtools, and also shows how much more reliable it is for producing an output (with FFEAtools failing to produce meshes for many of the test cases). This, however, does not prove that users *will* find the MDCL-TMA workflow to be a better user experience. MDCL-TMA should be rolled out to FFEA users, and once given time to try it, a similar qualitative analysis study as the one found in Chapter 4 should be performed to obtain a clear comparison in the user experience before and after the introduction of MDCL-TMA.

The effectiveness of the comparative analysis is hindered by the difficulty of meshing biomolecules with FFEAtools, which limited the number of comparable cases. As discussed in the qualitative analysis of FFEA (Chapter 4) the generation of FFEAtools meshes can be a very time-consuming process, and it was not suitable to try and obtain more FFEAtools meshes in the time frame of this research. In the future, if the time could be found, it would be very helpful to improve the reliability of these experiments by having more biomolecules meshed in

FFEAtools to compare and contrast with MDCL-TMA meshes.

MDCL-TMA currently has a prototype implementation written in Python. While fairly feature-complete, this prototype should continue to be iterated upon to introduce any additional features (such as potentially removing internal elements), and to ensure a robust algorithm performance so that large biomolecules can always be meshed in a timely manner. As this meshing method was developed to improve the sustainability of the FFEA software, so too must it be ensured that the code for MDCL-TMA itself is maintained to be sustainable.

In working towards automated KOBRA rod implementation, this research was only able to prove the concept that machine learning could be used to predict the presence of rod structures in biomolecules. There is a lot more work to be done to achieve the final goal of having the setup of KOBRA rods in FFEA simulations be fully automatic. First of all, the results of the current rod classification algorithm could potentially be improved with a larger dataset with more care put into data processing (such as to remove all noise), as the time frame of this project limited the time that could be spent on obtaining a high quality dataset. There are also different classification algorithms that could be tried, mainly different kinds of artificial neural network, to see if they produce higher cross-validation scores. Alternatively, the analysis of electron density maps could be re-tried with algorithms more suited to the data type, such as point cloud classification algorithms (Qi et al. 2017; Li et al. 2018; Zhang et al. 2020).

The next step towards the final goal of full automation requires an algorithm to find the location of rods in a biomolecule structure, rather than just whether or not a biomolecule contains any rods. Computer vision approaches such as feature line extraction, mesh segmentation, and shape graph analysis could build off the existing work here by trying to identify rod-like regions from analysing the curvatures across the biomolecule surface, but additional shape feature extraction methods are also possible if these prove unsuitable. Once these rod regions are located, a program will be required to take all the mesh elements contained within the regions obtained by the previous algorithm, and replace them with a KOBRA rod structure that connects to any neighbouring structures. If implemented successfully, this should further reduce the complexity and time-cost of setting up FFEA experiments. Hopefully, this will improve software sustainability through being easier-to-use and thus having a better user retention.

Finally, while it has been shown that general method outlined in §1.2 has been effective for the specific case of FFEA, further research is needed to test and refine this method for generic

research software. There are the clear limitations to address to make improvements (§9.2), but to thoroughly test this methodology this research needs to be re-conducted on other pieces of research software. With a collection of case studies all following the same (but refined) method, the results can then be compared to see how effective the methodology is when applied to generic research software, and what further limitations may become apparent. From doing this, the RSE community would have a pathway to developing best practices for improving research software sustainability through evidence-based research, and would have access to exemplar research for how to apply these practices to specific research software topics.

9.4 Final remarks

This research has applied research software engineering sustainable practices to the molecular dynamics software suite Fluctuating Finite Element Analysis. A thorough analysis of the code performance and the user experiences allowed for the identification of the severe bottlenecks in the workflow, making FFEA more difficult to use and learn than it needed to be. With a clear understanding of the real problems affecting the software, this project was able to design and develop new features of the FFEA software suite that would alleviate these problems, and thus improve the user experience for a better user retention to stave off software decay.

As quoted at the start of Chapter 1, Welch (2020) makes the point that "*We cannot depend on the exponential growth of computing speed to solve ever more difficult problems [in computational biophysics]*". This research showcases that there are other computational methods for improving the capabilities of molecular dynamics software. If Welch (2020) argues the point that computational biophysics cannot improve its capabilities by relying on improving computing speeds, then this thesis builds upon the point by showing that it does not *have* to.

As discussed in the literature review in §2.1, and as found for FFEA in this research's qualitative analysis in Chapter 4, many of the issues with research software stem from being developed by researchers untrained in software development, and from a lack of resources devoted to maintenance. If the scientific community truly wishes to address this wider issue, then research policies must be changed to allow more resource allocation to the sustainable development and maintenance of research software. If this cannot be achieved, then in the meantime researchers must continue to develop tactics for analysing and improving the sustainability of currently-unsustainable research software, such as the general methods presented in this thesis.

References

Advanced Visualizer 3.0 Manual (n.d.). Alias Wavefront. Chap. B1. Object Files (.obj). Not archived. Author unknown. Published between 1991-1995.

Affonso, C., Rossi, A. L. D., Vieira, F. H. A. and de Carvalho, A. C. P. de L. F. (2017). "Deep learning for biological image classification". In: *Expert Systems with Applications* 85, pp. 114–122. DOI: 10.1016/j.eswa.2017.05.039.

Aggarwal, C. C., ed. (2015). *Data Classification: Algorithms and Applications*. CRC Press. ISBN: 978-1-4665-8674-1.

Archer (2024). *Case Studies*. URL: https://www.archer2.ac.uk/research/case-studies/ (visited on 01/12/2024).

Avila, L. S., Ayachit, U., Barré, S., Baumes, J., Bertel, F., Blue, R., Cole, D., DeMarle, D., Geveci, B., Hoffman, W. A., King, B., Krishnan, K., Law, C. C., Martin, K. M., McLendon, W., Pebay, P., Russell, N., Schroeder, W. J., Shead, T., Shepherd, J., Wilson, A. and Wylie, B. (2010). *The VTK User's Guide*. 11th ed. Kitware Inc., pp. 469–482. ISBN: 978-1-930934-23-8.

Ayachit, U. (2015). *The ParaView Guide, A Parallel Visualization Application*. Kitware Inc. ISBN: 978-1930934290.

Al-Azzawi, A., Ouadou, A., Duan, Y. and Cheng, J. (2020). "Auto3DCryoMap: an automated particle alignment approach for 3D cryo-EM density map reconstruction". In: *BMC Bioinformatics* 21.534. DOI: 10.1186/s12859-020-03885-9.

Al-Azzawi, A., Ouadou, A., Max, H., Duan, Y., Tanner, J. J. and Cheng, J. (2020). "DeepCryoPicker: fully automated deep neural network for single protein particle picking in cryo-EM". In: *BMC Bioinformatics* 21.509. DOI: 10.1186/s12859-020-03809-7.

Al-Azzawi, A., Ouadou, A., Tanner, J. J. and Cheng, J. (2019). "AutoCryoPicker: an unsupervised learning approach for fully automated single particle picking in Cryo-EM images". In: *BMC Bioinformatics* 20.326. DOI: 10.1186/s12859-019-2926-y.

B., J. (2020). *Marching Diamond Crystal Lattice*. URL: https://github.com/d3x0r/MarchingTetrahedra (visited on 19/04/2023).

Baxter, R., Hong, N. C., Gorissen, D., Hetherington, J. and Todorov, I. (2012). "The Research Software Engineer". In: Digital Research 2012.

Berman, H. M., Westbrook, J., Feng, Z., Gilliland, G., Bhat, T. N., Weissig, H., Shindyalov, I. N. and Bourne, P. E. (2000). "The Protein Data Bank". In: *Nucleic Acids Research* 28 (1), pp. 235–242. DOI: 10.1093/nar/28.1.235.

Better Scientific Software (2024). *All Resources*. URL: https://bssw.io/items (visited on 01/12/2024).

Blender Online Community (2023). *Blender 3.6 Manual*. Blender Foundation. URL: https://docs.blender.org/manual (visited on 12/10/2023).

Burns, M. (1993). *Automated Fabrication: Improving Productivity in Manufacturing*. PTR Prentice Hall. Chap. 6.5. ISBN: 9780131194625.

Carver, J. C., Cosden, I. A., Hill, C., Gesing, S. and Katz, D. S. (2021). "Sustaining Research Software via Research Software Engineers and Professional Associations". In: *2021 IEEE/ACM International Workshop on Body of Knowledge for Software Sustainability (BoKSS)*. DOI: 10.1109/BoKSS52540.2021.00016.

Cheng, A., Henderson, R., Mastronarde, D., Ludtke, S. J., Schoenmakers, R. H. M., Short, J., Marabini, R., Dallakyan, S., Agard, D. and Winn, M. (2015). "MRC2014: Extensions to the MRC format header for electron cryo-microscopy and tomography". In: *Journal of Structural Biology* 192.2, pp. 146–150. DOI: 10.1016/j.jsb.2015.04.002.

Cignoni, P., Callieri, M., Corsini, M., Dellepiane, M., Ganovelli, F. and Ranzuglia, G. (2008). "MeshLab: an Open-Source Mesh Processing Tool". In: *Sixth Eurographics Italian Chapter Conference*, pp. 129–136.

Cocking, R., Gravett, M., Hanson, B., Harlen, O., Harris, S., Leng, J., Oliver, R., Read, D., Richardson, R., Ridley, T., Rogers, J., Solernou, A. and Welch, R. (2021a). *FFEA: Fluctuating Finite Element Analysis*. Chap. FFEA Input File Reference. URL: `https://ffea.readthedocs.io/en/latest/keywordReference.html` (visited on 19/04/2023).

– (2021b). *FFEA: Fluctuating Finite Element Analysis*. Chap. Tutorial. URL: `https://ffea.readthedocs.io/en/latest/Tutorial.html` (visited on 19/04/2023).

– (2023). *FFEA*. URL: `https://bitbucket.org/FFEA/ffea/src/master/` (visited on 20/09/2023).

Cohen, J., Katz, D. S., Barker, M., Haines, R. and Hong, N. C. (2018). "Building a Sustainable Structure for Research Software Engineering Activities". In: IEEE 14th International Conference on e-Science, pp. 31–32. DOI: `10.1109/eScience.2018.00015`.

Cohen, J., Katz, D. S., Barker, M., Hong, N. C., Haines, R. and Jay, C. (2021). "The Four Pillars of Research Software Engineering". In: *IEEE Software* 38 (1), pp. 97–105. DOI: `10.1109/MS.2020.2973362`.

Conceicao, R. C., O'Halloran, M., Glavin, M. and Jones, E. (2010). "Support Vector Machines for the Classification of Early-Stage Breast Cancer Based on Radar Target Signatures". In: *Progress In Electromagnetics Research B* 23, pp. 311–327. DOI: `10.2528/PIERB10062407`.

D'Agostino, D., Clematis, A., Decherchi, S., Rocchia, W., Milanesi, L. and Merelli, I. (2014). "CUDA accelerated molecular surface generation". In: *Concurrency and Computation: Practice and Experience* 26.10: *Special Issue: Special issue on COMPBIO 2013*, pp. 1819–1831. DOI: `10.1002/cpe.3120`.

Doi, A. and Koide, A. (1991). "An Efficient Method of Triangulating Equi-Valued Surfaces by Using Tetrahedral Cells". In: *IEICE Transactions of Information and Systems* E74-D (1), pp. 214–224. ISSN: 0916-8532.

European Molecular Biology Laboratory – European Bioinformatics Institute (2023). *Electron Microscopy Data Bank*. URL: https://www.ebi.ac.uk/emdb/ (visited on 19/04/2023).

Fenlason, J. and Stallman, R. (1998). *The GNU Profiler*. (Visited on 12/10/2023).

Gíslason, M. K. and Nash, D. H. (2012). "Finite Element Modelling of a Multi-Bone Joint: The Human Wrist". In: *Finite Element Analysis - New Trends and Developments*, pp. 77–98. ISBN: 978-953-51-0769-9. DOI: 10.5772/50560.

Gowers, R. J., Linke, M., Barnoud, J., Reddy, T. J. E., Melo, M. N., Seyler, S. L., Domański, J., Dotson, D. L., Buchoux, S., Kenney, I. M. and Beckstein, O. (2016). "MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations". In: Proceedings of the 15th Python in Science Conference, pp. 98–105. DOI: 10.25080/majora-629e541a-00e.

Gravett, M. S. C. (2022). "Understanding the mechanics of myosin 5a through computational modelling and cryo-EM". University of Leeds.

Gravett, M. S. C., Cocking, R. C., Curd, A. P., Harlen, O., Leng, J., Muench, S. P., Peckham, M., Read, D. J., Rogers, J. F., Welch, R. C. and Harris, S. A. (2021). "Moving in the mesoscale: Understanding the mechanics of cytoskeletal molecular motors by combining mesoscale simulations with imaging". In: *WIREs Computational Molecular Science* 12 (3). DOI: 10.1002/wcms.1570.

Gravett, M. S. C. and Rogers, J. (2023). *mdcl-tetrahedral-meshing*. URL: https://bitbucket.org/jfrogers/mdcl_tetrahedral_meshing/src/master/ (visited on 19/12/2023). Private repository due to potential licensing issues.

Gravett, M. S. C., Rogers, J., Leng, J. and Pickering, J. (2023). *ffea-meshing*. URL: https://bitbucket.org/FFEA/ffea-meshing/src/master/ (visited on 19/12/2023).

H., N. (2021). *Bede (NICE)*. URL: https://n8cir.org.uk/supporting-research/facilities/nice/ (visited on 19/04/2023).

Hafner, A. E., Krausser, J. and Šarić, A. (2019). "Minimal coarse-grained models for molecular self-organisation in biology". In: *Current Opinion in Structural Biology* 58: *Biophysical and*

Computational Methods. Ed. by A. R. Lowe and L. S. Itzhaki, pp. 43–52. DOI: 10.1016/j.sbi.2019.05.018.

Hanson, B. S. (2018). "Mesoscale Modelling of Cytoplasmic Dynein using Fluctuating Finite Element Analysis". PhD in Theoretical Physics. University of Leeds.

Hansson, S. O., ed. (2015). *The Role of Technology in Science: Philosophical Perspectives*. Springer Science+Business Media, Dordrecht. Chap. Computer Simulations: A New Mode of Scientific Inquiry? ISBN: 978-94-017-9761-0. DOI: 10.1007/978-94-017-9762-7.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwij, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C. and Oliphant, T. E. (2020). "Array programming with NumPy". In: *Nature* 585.7825, pp. 357–362. DOI: 10.1038/s41586-020-2649-2.

He, Z. K., Cai, J., Cui, S., Liu, B. and Wang, Y. D. (2017). "Static Structural Analysis of 2MW Wind Turbine Blade Based on ANSYS Workbench". In: *DEStech Transactions on Engineering and Technology Research*. DOI: 10.12783/dtetr/tmcm2017/12678.

Hegg, M. and Lin, L. Y. (2008). "Near-field photodetection with high spatial resolution by nanocrystal quantum dots". In: *Optics Express* 15.25, pp. 17163–17170. DOI: 10.1364/OE.15.017163.

Hettrick, S. (2016). *A not-so-brief history of Research Software Engineers*. URL: https://www.software.ac.uk/blog/not-so-brief-history-research-software-engineers (visited on 21/08/2024).

Homès, B. (2024). *Fundamentals of Software Testing*. 2nd ed. John Wiley & Sons, Inc., p. xxviii. ISBN: 978-1-78630-982-2.

Humphrey, William, Dalke, Andrew and Schulten, Klaus (1996). "VMD – Visual Molecular Dynamics". In: *Journal of Molecular Graphics* 14, pp. 33–38. DOI: 10.1016/0263-7855(96)00018-5.

Hunter, J. D. (2007). "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3, pp. 90–95. DOI: 10.1109/MCSE.2007.55.

Isenberg, T., Isenberg, P., Chen, J., Sedlmair, M. and Möller, T. (2013). "A Systematic Review on the Practice of Evaluating Visualization". In: *IEEE Transactions on Visualization and Computer Graphics* 19.12, pp. 2818–2827. DOI: 10.1109/TVCG.2013.126.

Islam, M., Dinh, A., Wahid, K. and Bhowmik, P. (2017). "Detection of potato diseases using image segmentation and multiclass support vector machine". In: *2017 IEEE 30th Canadian Conference on Electrical and Computer Engineering (CCECE)*. IEEE, pp. 1–4. DOI: 10.1109/CCECE.2017.7946594.

ISO (2021). *ISO/IEC 5055:2021*. International Organization for Standardization. URL: https://www.iso.org/standard/80623.html (visited on 26/11/2024).

Jackson, B. (2021). "Open Data Policies among Library and Information Science Journals". In: *Publications* 9.2 (25). DOI: 10.3390/publications9020025.

jenkinsci-docs (2023). *Jenkins Handbook*. URL: https://www.jenkins.io/doc/book/ (visited on 12/10/2023).

Jiang, Z., Yang, M., Tsirlin, M., Tang, R., Dai, Y. and Lin, J. (2023). ""Low-Resource" Text Classification: A Parameter-Free Classification Method with Compressors". In: *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 6810–6828. DOI: 10.18653/v1/2023.findings-acl.426.

Jiménez, R. C., Kuzak, M., Alhamdoosh, M., Barker, M., Batut, B., Borg, M., Capella-Gutierrez, S., Hong, N. C., Cook, M., Corpas, M., Flannery, M., Garcia, L., Gelpí, J. L., Gladman, S., Goble, C., Ferreiro, M. G., Gonzalez-Beltran, A., Griffin, P. C., Grüning, B., Hagberg, J., Holub, P., Hooft, R., Ison, J., Katz, D. S., Leskošek, B., Gómez, F. L., Oliveira, L. J., Mellor, D., Mosbergen, R., Mulder, N., Perez-Riverol, Y., Pergl, R., Pichler, H., Pope, B., Sanz, F., Schneider, M. V., Stodden, V., Suchecki, R., Vařeková, R. S., Talvik, H., Todorov, I., Treloar, A., Tyagi, S., Gompel, M. van, Vaughan, D., Via, A., Wang, X., Watson-Haigh, N. S. and Crouch, S. (2017). "Four simple recommendations to encourage best practices in research software". In: *F1000Research* 6.876. DOI: 10.12688/f1000research.11407.1.

Kenmotsu, K. (2003). *Surfaces with Constant Mean Curvature*. American Mathematical Society. ISBN: 9780821834794.

Keskin, C., Kıraç, F., Kara, Y. E. and Akarun, L. (2012). "Hand Pose Estimation and Hand Shape Classification Using Multi-layered Randomized Decision Forests". In: *Computer Vision – ECCV 2012*. Springer, Berlin, Heidelberg, pp. 852–863. ISBN: 978-3-642-33782-6. DOI: 10.1007/978-3-642-33783-3_61.

Kitware Inc. (2023). *VTK Documentation*. URL: https://vtk.org/doc/nightly/html/index.html (visited on 06/12/2023).

Klaf, A. A. (1956). *Calculus Refresher*. Courier Corporation. Chap. Curvature of Curves. ISBN: 9780486203706.

Kmiecik, S., Gront, D., Kolinski, M., Wieteska, L., Dawid, A. E. and Kolinski, A. (2016). "Coarse-Grained Protein Models and Their Applications". In: *Chemical Reviews* 116 (14), pp. 7898–7936. DOI: 10.1021/acs.chemrev.6b00163.

Knüpfer, A., Rössel, C., Mey, D. an, Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A., Nagel, W. E., Oleynik, Y., Philippen, P., Saviankou, P., Schmidl, D., Shende, S., Tschüter, R., Wagner, M., Wesarg, B. and Wolf, F. (2011). "Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir". In: *Tools for High Performance Computing 2011*. Springer, pp. 79–91. DOI: 10.1007/978-3-642-31476-6_7.

Kollman, P., Case, D., Cheatham, T., Merz, K., Roitberg, A., Simmerling, C., Luo, R., Wang, J. and Walker, R. (2023). *AMBER*. URL: https://ambermd.org/ (visited on 18/04/2023).

Krivov, G. G., Shapovalov, M. V. and Dunbrack Jr., R. L. (2009). "Improved prediction of protein side-chain conformations with SCWRL4". In: *PROTEINS: Structure, Function, and Bioinformatics* 77 (4), pp. 778–795. DOI: 10.1002/prot.22488.

Lee, C. T., Laughlin, J. G., Moody, J. B., Amaro, R. E., McCammon, J. A., Holst, M. and Rangamani, P. (2020). "An Open-Source Mesh Generation Platform for Biophysical Modeling Using Realistic Cellular Geometries". In: *Biophysical Journal* 118 (5), pp. 1003–1008. DOI: 10.1016/j.bpj.2019.11.3400.

Li, Y., Bu, R., Sun, M., Wu, W., Di, X. and Chen, B. (2018). "PointCNN: convolution on X-transformed points". In: *NIPS'18: Proceedings of the 32nd International Conference on Neural Information Processing Systems*. Curran Associates Inc., pp. 828–838. DOI: 10.5555/3326943.3327020.

Liao, X., Xiao, L., Yang, C. and Lu, Y. (2014). "MilkyWay-2 supercomputer: system and application". In: *Frontiers of Computer Science* 8, pp. 345–356. DOI: 10.1007/s11704-014-3501-3.

Liljas, A., Liljas, L., Ash, M. -R., Lindblom, G., Nissen, P. and Kjeldgaard, M. (2016). *Textbook Of Structural Biology*. 2nd ed. World Scientific Publishing Co. Chap. Introduction. ISBN: 9789813142497.

Liu, S., Duan, K., Li, L., Wang, X. and Hu, Y. (2021). "A multilayer coarse-grained molecular dynamics model for mechanical analysis of mesoscale graphene structures". In: *Carbon* 178, pp. 528–539. DOI: 10.1016/j.carbon.2021.03.025.

Logan, D. L. (2022). *A First Course in the Finite Element Method*. 6th ed. Cengage Learning. ISBN: 978-0-357-67643-1.

Logilab (2014). *Pylint User Manual*. URL: https://docs.pylint.org/ (visited on 17/04/2023).

Mari, J. -L., Hétroy-Wheeler, F. and Subsol, G. (2019). *Geometric and Topological Mesh Feature Extraction for 3D Shape Analysis*. John Wiley & Sons, Inc. ISBN: 9781786300416.

Marsland, S. (2015). *Machine Learning: An Algorithmic Perspective*. 2nd ed. CRC Press. ISBN: 978-1-4665-8333-7.

Meyer, M., Desbrun, M., Schröder, P. and Barr, A. H. (2003). "Discrete Differential-Geometry Operators for Triangulated 2-Manifolds". In: *Visualization and Mathematics III*. Springer, Berlin, Heidelberg, pp. 35–57. ISBN: 978-3-642-05682-6. DOI: 10.1007/978-3-662-05105-4_2.

Microsoft (2023). *Documentation for Visual Studio Code*. URL: https://code.visualstudio.com/Docs (visited on 12/10/2023).

Musa, S. M., Kulkarni, A. V. and Havanur, V. K. (2014). *Finite Element Analysis: A Primer*. Dulles, VA: Mercury Learning and Information. ISBN: 978-93-81159-10-1.

Netgen/NGSolve Team (2023). *Netgen/NGSolve*. URL: `https://ngsolve.org/` (visited on 05/07/2023).

O'Neill, B. (2006). *Elementary Differential Geometry*. 2nd ed. Elsevier Science. ISBN: 9780080505428.

Oliver, R. C. (2013). "A Stochastic Finite Element Model for the Dynamics of Globular Proteins". University of Leeds, p. 82.

Pal, S. (2019). *Fundamentals of Molecular Structural Biology*. London, UK: Academic Press. ISBN: 978-0-12-814855-6.

Pan, X., Jin, H., Ku, X. and Fan, J. (2023). "The atom sampling method for mesoscale molecular dynamics and its application to graphene assemblies". In: *Physical Chemistry Chemical Physics* 25 (29), pp. 19727–19739. DOI: `10.1039/d3cp02219f`.

Park, Y., Yoon, J., Kang, B. and Kim, G. (2020). "A Tactile Device Generating Repulsive Forces of Various Human Tissues Fabricated from Magnetic-Responsive Fluid in Porous Polyurethane". In: *Materials (Basel)* 13 (5), pp. 1062–1076. DOI: `10.3390/ma13051062`.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M. and Duchesnay, E. (2011). "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12, pp. 2825–2830. DOI: `10.5555/1953048.2078195`.

Peng, S., Zhang, X., Su, W., Dong, D., Lu, Y., Liao, X., Lu, K., Yang, C., Liu, J., Zhu, W. and Wei, D. (2020). "High-Scalable Collaborated Parallel Framework for Large-Scale Molecular Dynamic Simulation on Tianhe-2 Supercomputer". In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 17.3, pp. 804–816. DOI: `10.1109/TCBB.2018.2805709`.

Pettersen, E. F., Goddard, T. D., Huang, C. C., Meng, E. C., Couch, G. S., Croll, T. I., Morris, J. H. and Ferrin, T. E. (2021). "UCSF ChimeraX: Structure visualization for researchers, educators, and developers". In: *Protein Sci.* 30.1, pp. 70–82. DOI: `10.1002/pro.3943`.

Python Software Foundation (2020). *Sunsetting Python 2*. URL: https://www.python.org/doc/sunset-python-2/ (visited on 17/04/2023).

Qi, C. R., Su, H., Mo, K. and Guibas, L. J. (2017). "PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, pp. 77–85. ISBN: 978-1-5386-0458-8. DOI: 10.1109/CVPR.2017.16.

Rahman, R. (2013). *Intel® Xeon Phi™ Coprocessor Architecture and Tools*. Apress, Berkeley, CA. Chap. Introduction to Xeon Phi Architecture, pp. 3–14. DOI: 10.1007/978-1-4302-5927-5_1.

Rasmussen, B. (2014). *High-Performance Windows Store Apps*. Redmond, WA: Microsoft Press, p. 82. ISBN: 9780133967173.

RCSB (2023). *PDB Molecule of the Month*. URL: https://pdb101.rcsb.org/motm/motm-by-date (visited on 29/05/2023).

Reiser, P., Neubert, M., Eberhard, A., Torresi, L., Zhou, C., Shao, C., Metni, H., van Hoesel, C., Schopmans, H., Sommer, T. and Friederich, P. (2022). "Graph neural networks for materials science and chemistry". In: *Communications Materials* 3. DOI: 10.1038/s43246-022-00315-6.

Richardson, R. A., Papachristos, K., Read, D. J., Harlen, O. G., Harrison, M., Paci, E., Muench, S. P. and Harris, S. A. (2014). "Understanding the apparent stator-rotor connections in the rotary ATPase family using coarse-grained computer modeling". In: *PROTEINS: Structure, Function, and Bioinformatics* 82 (12), pp. 3298–3311. DOI: 10.1002/prot.24680.

Rogers, J. (2021a). *FFEA 2.7.2 Evaluation Report*. Tech. rep. University of Leeds. Unpublished internal developer report.

– (2021b). *ffea-dev-tools*. URL: https://bitbucket.org/FFEA/ffea-dev-tools/src/master/ (visited on 19/12/2023).

Schrödinger (2023). *PyMOL*. URL: https://pymol.org/ (visited on 20/09/2023).

Schroeder, W., Martin, K. and Lorensen, B. (2006). *The Visualization Toolkit*. 4th ed. Kitware Inc. ISBN: 978-1-930934-19-1.

Science and Technology Facilities Council (2017). *Documentation for mrcfile.py*. URL: https://mrcfile.readthedocs.io/en/stable/ (visited on 28/08/2023).

scikit-learn Developers (2023). *scikit-learn User Guide*. Chap. Support Vector Machines. URL: https://scikit-learn.org/stable/user_guide.html (visited on 04/12/2023).

Shalev-Shwartz, S. and Ben-David, S. (2014). *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press. ISBN: 978-1-107-05713-5.

Shaw, M. (2003). "Writing Good Software Engineering Research Papers". In: 25th International Conference on Software Engineering, IEEE Computer Society, 2003, pp. 726–736. DOI: 10.1109/ICSE.2003.1201262.

Shivakumar, S. K. (2014). *Architecting High Performing, Scalable and Available Enterprise Web Applications*. Philadelphia, PA: Elsevier Science, p. 143. ISBN: 9780128025284.

Software Sustainability Institute (2024). *Resource Hub*. URL: https://www.software.ac.uk/resource-hub (visited on 01/12/2024).

Solernou, A., Hanson, B. S., Richardson, R. A., Welch, R., Read, D. J., Harlen, O. G. and Harris, S. A. (2018). "Fluctuating Finite Element Analysis (FFEA): A continuum mechanics software tool for mesoscale simulation of biomolecules". In: *PLOS Computational Biology*. Ed. by T. Poisot. DOI: 10.1371/journal.pcbi.1005897.

Souza, M. R. de, Haines, R., Vigo, M. and Jay, C. (2019). "What Makes Research Software Sustainable? An Interview Study With Research Software Engineers." In: 12th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE), pp. 135–138. DOI: 10.1109/CHASE.2019.00039.

Spyder Doc Contributors (2023). *Welcome to Spyder's Documentation*. URL: https://docs.spyder-ide.org/current/index.html (visited on 12/10/2023).

Staden, R. (1979). "A strategy of DNA sequencing employing computer programs". In: *Nucleic Acids Research* 6 (7), pp. 2601–2610. DOI: 10.1093/nar/6.7.2601.

Sternberg, S. (2012). *Curvature in mathematics and in physics*. Dover Publications. ISBN: 9780486292717.

Stone, J. E., Vandivort, K. L. and Schulten, K. (2013). "GPU-accelerated molecular visualization on petascale supercomputing platforms". In: 8th International Workshop on Ultrascale Visualization, pp. 1–8. DOI: 10.1145/2535571.2535595.

Supercomputing Applications, National Center for (2023). *Blue Waters*. URL: https://bluewaters.ncsa.illinois.edu/ (visited on 18/04/2023).

Tersoff, J. (1988). "Empirical Interatomic Potential for Carbon, with Applications to Amorphous Carbon". In: *Physical Review Letters* 61.25, pp. 2879–2882. DOI: 10.1103/PhysRevLett.61.2879.

The Clang Team (2023). *Clang-Tidy*. URL: https://clang.llvm.org/extra/clang-tidy/ (visited on 17/04/2023).

The GDB Developers (2023). *GDB Documentation*. URL: https://sourceware.org/gdb/documentation/ (visited on 12/10/2023).

The Python Standard Library (2023). *2to3 — Automated Python 2 to 3 code translation*. URL: https://docs.python.org/3/library/2to3.html (visited on 17/04/2023).

The SciPy Community (2023). *SciPy Documentation*. Chap. Multidimensional image processing (scipy.ndimage). URL: https://docs.scipy.org/doc/scipy/reference/ndimage.html (visited on 18/09/2023).

Theisen, C., Dunaiski, M., Williams, L. and Visser, W. (2017). "Writing Good Software Engineering Research Papers: Revisited". In: 39th IEEE International Conference on Software Engineering Companion, p. 402. DOI: 10.1109/ICSE-C.2017.51.

tkruse (2019). *CMakeLint*. URL: https://github.com/cmake-lint/cmake-lint (visited on 17/04/2023).

Tomar, S. (2006). "Converting video formats with FFmpeg". In: *Linux Journal* 2006.146, p. 10.

UNESCO (2023). *Education for sustainable development*. URL: https://www.unesco.org/en/sustainable-developmen/education (visited on 15/12/2023).

Valgrind Developers (2022a). *Valgrind*. URL: https://www.valgrind.org/ (visited on 17/04/2023).

– (2022b). *Valgrind User Manual*. Chap. Memcheck: a memory error detector. URL: https://valgrind.org/docs/manual/mc-manual.html (visited on 19/04/2023).

– (2022c). *Valgrind User Manual*. Chap. Helgrind: a thread error detector. URL: https://valgrind.org/docs/manual/hg-manual.html (visited on 19/04/2023).

Venters, C. C., Capilla, R., Nakagawa, E. Y., Betz, S., Penzenstadler, B., Crick, T. and Brooks, I. (2022). "Sustainable software engineering: Reflections on advances in research and practice". In: *Information and Software Technology* 164. DOI: 10.1016/j.infsof.2023.107316.

Venters, C. C., Kocak, S. A., Betz, S., Brooks, I., Capilla, R., Chitchyan, R., Duboc, L., Heldal, R., Moreira, A., Oyedeji, S., Penzenstadler, B., Porras, J. and Seyff, N. (2021). "Software Sustainability: Beyond the Tower of Babel". In: *2021 IEEE/ACM International Workshop on Body of Knowledge for Software Sustainability (BoKSS)*. IEEE, pp. 3–4. DOI: 10.1109/BoKSS52540.2021.00009.

Wagner, T., Merino, F., Stabrin, M., Moriya, T., Antoni, C., Apelbaum, A., Hagel, P., Sitsel, O., Raisch, T., Prumbaum, D., Quentin, D., Roderer, D., Tacke, S., Siebolds, B., Schubert, E., Shaikh, T. R., Lill, P., Gatsogiannis, C. and Raunser, S. (2019). "SPHIRE-crYOLO is a fast and accurate fully automated particle picker for cryo-EM". In: *Communications Biology* 2.218. DOI: 10.1038/s42003-019-0437-z.

Wang, Y., Li, Z. and Farimani, A. B. (2022). *Machine Learning in Molecular Sciences*. Springer International Publishing. Chap. Graph Neural Networks for Molecules. DOI: 10.1007/978-3-031-37196-7_2.

Weierstrass Institute for Applied Analysis and Stochastics (2013). *TetGen, Version 1.5, User's Manual*. URL: http://wias-berlin.de/software/tetgen/1.5/doc/manual (visited on 31/08/2023).

Welch, R. (2020). "KOBRA: A Fluctuating Elastic Rod Model for Slender Biological Macromolecules". Doctor of Philosophy. University of Leeds.

Workshop on Sustainable Software for Science: Practice and Experiences (2017). *Proceedings*. URL: https://wssspe.researchcomputing.org.uk/proceedings/ (visited on 01/12/2024).

Worldwide Protein Data Bank (2012). *Atomic Coordinate Entry Format Version 3.3*. URL: http://www.wwpdb.org/documentation/file-format-content/format33/v3.3.html (visited on 29/08/2023).

Xu, D. and Zhang, Y. (2012). "Ab initio protein structure assembly using continuous structure fragments and optimized knowledge-based force field". In: *PROTEINS: Structure, Function, and Bioinformatics* 80, pp. 1715–1735. DOI: 10.1002/prot.24065.

Xu, S., Liu, S., Wang, H., Chen, W., Zhang, F. and Xiao, Z. (2020). "A Hyperspectral Image Classification Approach Based on Feature Fusion and Multi-Layered Gradient Boosting Decision Trees". In: *Entropy (Basel)* 23.1 (20). DOI: 10.3390/e23010020.

Youree, R. K. (2023). *Software Reliability Techniques for Real-World Applications*. John Wiley & Sons, Inc., p. 177. ISBN: 9781119931829.

Zhang, M., You, H., Kadam, P., Liu, S. and Kuo, C. -C. J. (2020). "PointHop: An Explainable Machine Learning Method for Point Cloud Classification". In: *IEEE Transactions on Multimedia* 22 (7), pp. 1744–1755. DOI: 10.1109/TMM.2019.2963592.

Zhou, K. and Liu, B. (2022). *Molecular Dynamics Simulation: Fundamentals and Applications*. Amsterdam, Netherlands: Elsevier. ISBN: 978-0-12-816419-8. DOI: 10.1016/B978-0-12-816419-8.00006-4.

Appendix A

Qualitative Assessment of the Fluctuating Finite Element Analysis (FFEA) Software Questionnaire

Qualitative Assessment of the Fluctuating Finite Element Analysis (FFEA) Software Questionnaire

Please read through the following questions and type your answer underneath in the spaces provided. Some questions may also have multiple choice options in ***bold****. For these, highlight your answer either using a highlighting tool or with an underline.*

As a reminder, you are allowed to refuse to answer any question given without stating a reason. Also, while this appears to be a large amount of questions, it is highly likely that not all questions will be relevant to each participant. For example, this may be due to you not being a developer of FFEA, or not having experience with a particular part of FFEA. Simply leave any questions blank if you wish to not answer them or they are not relevant to you.

Please do not feel as if you must answer all questions in one session. You may pause and come back to this at a later time. When completed, please email this document to the lead researcher at j.f.rogers1@leeds.ac.uk. Please return this document within 2 weeks of obtaining.

This questionnaire is split into 3 sections. All participants should answer the questions in Sections A and C. If you answer "Yes" to Question 1, please also complete Section B.

Section A

All participants should answer these questions.

1. Do you have at least 2 weeks full time effort of developing the FFEA source code in the last 12 months?

Yes | **No**

Part 1 - FFEA Runner and Tools Questions

The following questions are in reference to the FFEA runner and FFEA tools. You will be asked about the FFEA visualiser in the next part.

2. Can you give a brief description of what functionality and workflow you have used in FFEA? You do not have to give specific details about any research you have done.

3. How much time does it generally take you to set up your commonly performed workflow and how much time does it generally take the FFEA runner to generate results?

4. What parts of these workflows do you generally find easy to do and why?

5. What parts of these workflows do you generally find satisfying to do and why?

6. What parts of these workflows do you generally find difficult to do and why?

7. What parts of these workflows do you generally find unsatisfying to do and why?

8. Are there any workarounds you've created to assist with the difficult or unsatisfying tasks that you encounter?

9. Are there any bugs you have encountered while doing your commonly performed tasks?

    a. If so, roughly how many?

    b. What was it and how often do you encounter each of them?

    c. How did you solve or work around these bugs? Do you know if the development team is aware of these bugs?

10. Do you have experience using FFEA with multiple people? If so, how well do you think FFEA accommodates for collaborative work?

11. Are there any features you can think of that if added to FFEA would reduce how difficult and unsatisfying you find certain tasks?

Part 2 - FFEA Visualiser Questions

The following questions are in reference to the FFEA visualiser plugin for PyMOL.

12. Do you have experience with using the FFEA visualiser since November in 2017?

Yes | **No**

If the answer to the previous question was "No" then the rest of the questions in this part can be skipped.

13. Can you describe how you use the FFEA visualiser as part of your workflow/workflows?

14. How much time does it generally take for the FFEA visualiser to load the data of your each of your workflow?

15. How well-suited do you find the FFEA visualiser for your workflow and how well does it present the results?

16. How effective do you find the outputs of the visualizer (e.g. images and videos) for communicating structural biology to the public and why?

17. What workflows in the FFEA visualiser do you find easy to do and why?

18. What workflows in the FFEA visualiser do you find satisfying to do and why?

19. What workflows in the FFEA visualiser do you find difficult to do and why?

20. What workflows in the FFEA visualiser do you find unsatisfying to do and why?

21. Are there any workarounds you've created to assist with the difficult or unsatisfying tasks you encounter?

22. Are there any bugs you have encountered while doing your commonly performed tasks?

a. If so, roughly how many?

b. What was it and how often do you encounter each of them?

c. How did you solve or work around these bugs? Do you know if the development team is aware of these bugs?

23. Do you come across any accessibility issues with the FFEA visualiser (such as layout issues, issues with colour usage, incompatibility with accessibility software etc.)?

    a. If so, how does this impact your ability to use the FFEA visualiser? Do you know if the development team is aware of these issues?

24. If you have experience of using the FFEA visualizer with multiple people? If so, how well do you think the FFEA visualiser accommodates for collaborative work?

25. Are there any features you can think of that if added to the FFEA visualiser would reduce how difficult and unsatisfying you find certain tasks?

Part 3 - Statements

In this section you will be shown a list of statements. Under each you will be given multiple options on how much you agree or disagree with the statement. For each, please highlight your answer and explain why you gave that answer underneath the multiple-choice options.

26. FFEA tools is easy to use.

Extremely Disagree | **Highly Disagree** | **Disagree** | **Agree** | **Highly Agree** | **Extremely Agree**

27. FFEA tools is easy to learn.

Extremely Disagree | **Highly Disagree** | **Disagree** | **Agree** | **Highly Agree** | **Extremely Agree**

28. FFEA tools is satisfying to use.

Extremely Disagree | **Highly Disagree** | **Disagree** | **Agree** | **Highly Agree** | **Extremely Agree**

29. FFEA tools runs at an appropriate speed.

Extremely Disagree | **Highly Disagree** | **Disagree** | **Agree** | **Highly Agree** | **Extremely Agree**

30. FFEA runner is easy to use.

Extremely Disagree | **Highly Disagree** | **Disagree** | **Agree** | **Highly Agree** | **Extremely Agree**

31. FFEA runner is easy to learn.

Extremely Disagree | Highly Disagree | Disagree | Agree | Highly Agree | Extremely Agree

32. FFEA runner is satisfying to use.

Extremely Disagree | Highly Disagree | Disagree | Agree | Highly Agree | Extremely Agree

33. FFEA runner runs at an appropriate speed.

Extremely Disagree | Highly Disagree | Disagree | Agree | Highly Agree | Extremely Agree

34. The FFEA visualiser is easy to use.

Extremely Disagree | Highly Disagree | Disagree | Agree | Highly Agree | Extremely Agree

35. The FFEA visualiser is easy to learn.

Extremely Disagree | Highly Disagree | Disagree | Agree | Highly Agree | Extremely Agree

36. The FFEA visualiser is satisfying to use.

Extremely Disagree | Highly Disagree | Disagree | Agree | Highly Agree | Extremely Agree

37. The FFEA visualiser runs at an appropriate speed.

Extremely Disagree | Highly Disagree | Disagree | Agree | Highly Agree | Extremely Agree

Section B

Please complete this section you answered "Yes" to Question 1.

Part 1 - General Developer Questions

38. Which of the following do you have experience with developing?

    FFEA Tools | FFEA Runner | FFEA Visualiser

39. Can you give a brief description of the general types of tasks you have done when developing FFEA (for example developing a new algorithm, refactoring, developing new tests, documentation, debugging etc.)? You do not have to give specific details.

40. How much time or effort did it take to develop the areas of FFEA you worked on?

41. What parts of development of FFEA did you generally find easy to do and why?

42. What parts of development of FFEA did you generally find unsatisfying to do and why?

43. What parts of development of FFEA did you generally find difficult to do and why?

44. What parts of development of FFEA did you generally find unsatisfying to do and why?

45. What tools have you used during your development work of FFEA and how did they help you?

46. What do you see as FFEA's current biggest issue that requires attention? How do you think FFEA's future would be affected if this issue is not solved?

47. Are there any changes you think could be made to FFEA or its development cycle that would improve its ease of development?

Part 2 - Ranked Questions

In this section you will be shown a list of statements. Under each you will be given multiple options on how much you agree or disagree with the statement. For each, please highlight your answer and explain why you gave that answer underneath the multiple-choice options.

48. FFEA tools is easy to develop for.

    Extremely Disagree | Highly Disagree | Disagree | Agree | Highly Agree | Extremely Agree

49. The source code for FFEA tools is easy to understand.
Extremely Disagree | Highly Disagree | Disagree | Agree | Highly Agree | Extremely Agree

50. I enjoy developing FFEA tools.
Extremely Disagree | Highly Disagree | Disagree | Agree | Highly Agree | Extremely Agree

51. FFEA runner is easy to develop for.
Extremely Disagree | Highly Disagree | Disagree | Agree | Highly Agree | Extremely Agree

52. The source code for FFEA runner is easy to understand.
Extremely Disagree | Highly Disagree | Disagree | Agree | Highly Agree | Extremely Agree

53. I enjoy developing FFEA runner.
Extremely Disagree | Highly Disagree | Disagree | Agree | Highly Agree | Extremely Agree

54. The FFEA visualiser is easy to develop for.
Extremely Disagree | Highly Disagree | Disagree | Agree | Highly Agree | Extremely Agree

55. The source code for the FFEA visualiser is easy to understand.
Extremely Disagree | Highly Disagree | Disagree | Agree | Highly Agree | Extremely Agree

56. I enjoy developing the FFEA visualiser.
Extremely Disagree | Highly Disagree | Disagree | Agree | Highly Agree | Extremely Agree

Section C

All participants should answer these questions.

57. Is there anything about your experience with using or developing FFEA that you have not discussed in the prior questions but think should be stated?

IMPORTANT: Now that you have finished providing your responses, please read through your answers and redact any that you wish to not be used in this research. Upon submitting your responses this will be considered as giving permission for direct quotes from your provided answers to appear in reports and publications, as stated in the information sheet and consent form. Remember, however, that even after submission of this questionnaire you maintain the right to withdraw until 01/10/22.

Appendix B

Datasets for rod classification

This appendix contains lists of biomolecule electron density maps obtained from the Electron Microscopy Data Bank (European Molecular Biology Laboratory – European Bioinformatics Institute 2023) used as the datasets for the experiments in Chapter 8 on classifying biomolecules based on whether or not they contain rod structures. The objects used are listed by their ID in EMDB, and the isolevel (or contour level) used to generate an isosurface is given alongside. More information about a particular object can be found by using the url `https://www.ebi.ac.uk/emdb/EMD-[ID]` where "[ID]" is replaced with the object's EMD-ID number.

All experiments in Chapter 8 use the data in Table B.1, which contains 30 biomolecules containing rod structures and 33 biomolecules containing no rod structures, for a total of 63 biomolecules. The second series of experiments in §8.3.3, the results of which are noted in Table 8.4, use the data in both Tables B.1 and B.2. The data in Table B.2 contains 77 biomolecules containing rod structures and 74 biomolecules containing no rod structures, for a total of 151 biomolecules. The combined datasets contain a total of 214 biomolecules, with an even split of 107 biomolecules containing rod structures and 107 biomolecules containing no rod structures.

Table B.1: Dataset of biomolecule electron density maps used for all experiments in Chapter 8.

Contains rods		**Contains no rods**	
EMD-ID	**Isolevel**	**EMD-ID**	**Isolevel**
1909	0.33	0524	0.00668
1995	15.5	0782	0.00738
2324	399.35	1858	0.5
2549	9.84	1985	0.5
2801	0.0688	2594	0.74
2946	0.0073	3103	3.5
3170	0.13	3580	0.019
3578	1.8	5043	1.15
3705	0.06	5120	0.14
3982	0.0279	5186	1.3
7067	0.156	5189	0.03
8702	10	5235	0.8
8724	0.04	5520	0.4
10827	0.03	5565	0.015
11070	0.022	5779	0.027
11681	0.5	6343	0.0155
11826	0.21	6344	6.59
11832	0.287	6775	0.0354
12887	0.03	6903	2.5
13174	3.24	8118	3.67
13895	11	8187	0.1
13986	0.031	11219	0.085
15666	0.00614	20265	3
15988	3	21487	0.15
26214	2.38	21820	0.17
30072	0.015	22288	0.15
30708	0.006	25687	0.8
30854	0.00997	27523	0.0654
32252	0.032	27539	0.01
34121	0.104	27812	0.614
		28896	0.54
		30888	0.3
		34173	0.264

Table B.2: Dataset of biomolecule electron density maps used for the second series of experiments in §8.3.3, the results of which are noted in Table 8.4.

Contains rods		**Contains no rods**	
EMD-ID	**Isolevel**	**EMD-ID**	**Isolevel**
13630	0.4	14379	0.129
14343	0.79	14706	0.005
14790	0.025	14863	0.0039
14878	1.3	15403	0.45
14879	2.2	15412	0.01
14881	0.18	15480	0.255
14882	0.15	15579	350
15343	0.0051	15853	0.033
15382	0.0233	15985	0.11
15395	0.04	16049	2.74
15398	0.06	16089	1.5
15473	0.017	16213	0.00505
15633	0.0245	16218	0.0276
15901	1.1	16223	0.027
15977	0.466	16238	0.08
16036	0.04	16303	0.202
16221	0.102	16736	0.32
16244	0.124	16804	0.2
16332	0.121	16827	0.236
16431	0.0175	16841	0.17
16540	3.63	16914	0.27
16618	0.158	16979	0.144
16683	0.03	16998	0.5
16686	0.025	17015	0.6
16847	0.0145	17156	0.044
16865	0.01	17162	0.373
16987	0.45	17163	0.086
17542	1.9	17464	0.5
17674	0.004	17533	0.125
17676	0.00238	17600	0.058
18146	4.11	17648	0.2
26571	4.06	17650	0.001
26572	4.29	17711	0.044
26887	0.39	17840	0.01
26904	0.63	17880	0.16
27495	0.4	18478	0.0255
27496	0.008	25427	0.00115
27790	0.01	26859	0.4
27887	0.1	26889	0.5
27912	0.367	27185	2.5
28126	0.2	27350	0.017
28736	0.0266	27378	0.019
28739	0.014	27684	0.00938
28762	0.12	27738	4.2
28834	0.00125	28074	0.026832

Contains rods		**Contains no rods**	
EMD-ID	**Isolevel**	**EMD-ID**	**Isolevel**
28839	0.00154	28555	0.72
28844	0.000712	28815	0.67
29016	0.15	28963	0.0112
29329	0.544	29667	11.1
29509	0.25	33235	0.05
29886	0.2	33511	1.49
32034	0.027	33582	4.482
32992	0.00557	33631	0.09
33833	4	34057	0.0361
34051	0.76	34117	0.11
34140	0.00379	34605	0.0165
34338	0.0437	34662	0.00757
35201	0.4	34689	0.5
35213	0.5	35274	0.111
35324	0.04	35527	7.14
35344	0.116	35828	0.657
35363	0.279	36140	0.098
35805	0.0044	36265	0.0141
36623	0.005	36464	9
37716	0.5	36866	0.13
40202	0.02	37316	0.115
40263	0.363	40162	0.15
40553	0.01	40242	0.03
41098	0.1	40691	0.002
41298	0.75	40983	0.6
41308	0.25	41114	0.0061
41679	0.0728	41120	0.0086
41791	0.349	41183	0.6
41983	0.028	42246	2.5
41999	0.004		
42142	0.016		
42149	0.679		