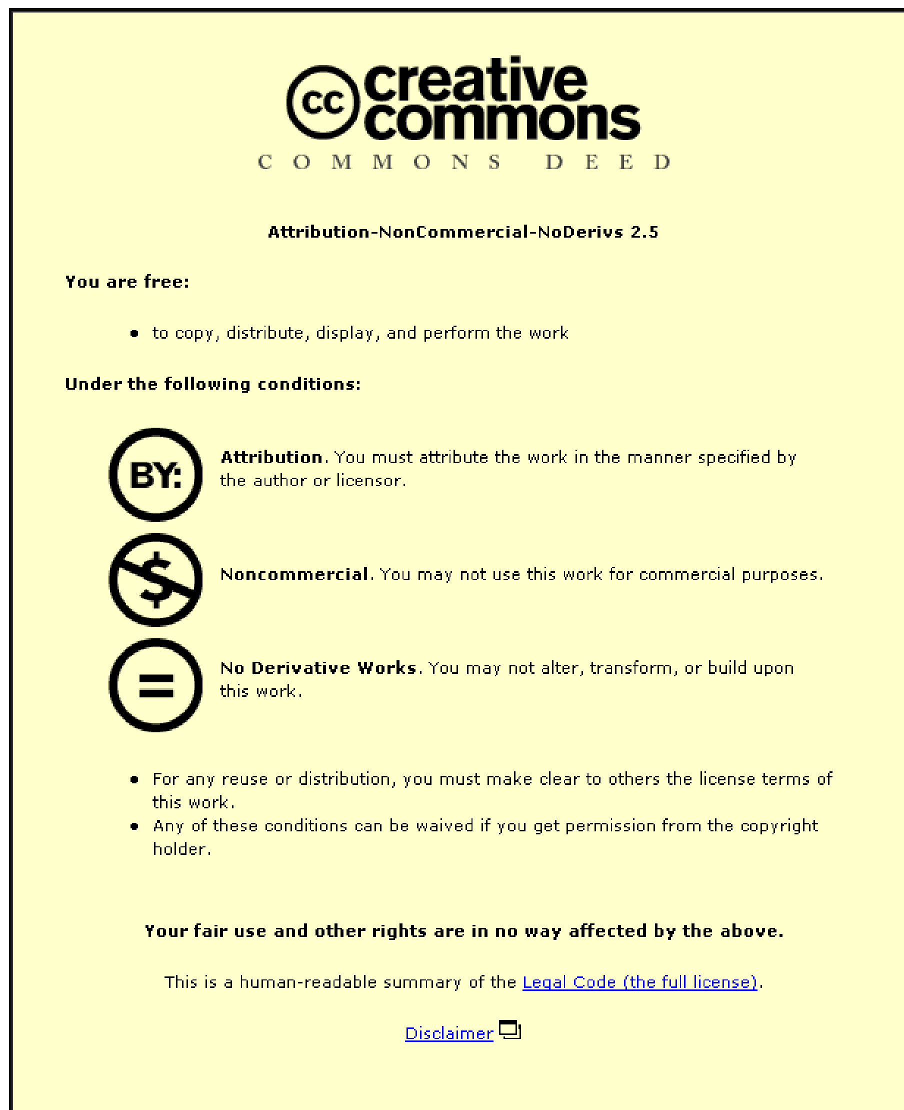
Loughborough University

This item was submitted to Loughborough University as a PhD thesis by the author and is made available in the Institutional Repository (https://dspace.lboro.ac.uk/) under the following Creative Commons Licence conditions.



For the full text of this licence, please go to:
http://creativecommons.org/licenses/by-nc-nd/2.5/

Application of Multi-core and Cluster Computing to the Transmission Line Matrix Method

by

Daniel Ryan Browne, M. Eng. (Hons), DIS

A Doctoral Thesis submitted in partial fulfilment of the requirements
for the award of Doctor of Philosophy of Loughborough University.

Submitted April 2014 - © 2014 Daniel R. Browne

To my parents Linda and Gareth, for their unconditional and never-ending love, support and encouragement.

Abstract

The Transmission Line Matrix (TLM) method is an existing and established mathematical method for conducting computational electromagnetic (CEM) simulations. TLM models Maxwell's equations by discretising the contiguous nature of an environment and its contents into individual small-scale elements and it is a computationally intensive process. This thesis focusses on parallel processing optimisations to the TLM method when considering the opposing ends of the contemporary computing hardware spectrum, namely large-scale computing systems versus small-scale mobile computing devices.

Theoretical aspects covered in this thesis are:

- The historical development and derivation of the TLM method.
- A discrete random variable (DRV) for raindrop diameter, allowing generation of a rain-field with raindrops adhering to a Gaussian size distribution, as a case study for a 3-D TLM implementation.
- Investigations into parallel computing strategies for accelerating TLM on large and small-scale computing platforms.

Implementation aspects covered in this thesis are:

- A script for modelling rain-fields using free-to-use modelling software.
- The first known implementation of 2-D TLM on mobile computing devices.
- A 3-D TLM implementation designed for simulating the effects of rain-fields on extremely high frequency (EHF) band signals.

By optimising both TLM solver implementations for their respective platforms, new opportunities present themselves. Rain-field simulations containing individual raindrop geometry can be simulated, which was previously impractical due to the lengthy computation times required. Also, computationally time-intensive methods such as TLM were previously impractical on mobile computing devices. Contemporary hardware features on these devices now provide the opportunity for CEM simulations at speeds that are acceptable to end users, as well as providing a new avenue for educating relevant user cohorts via dynamic presentations of EM phenomena.

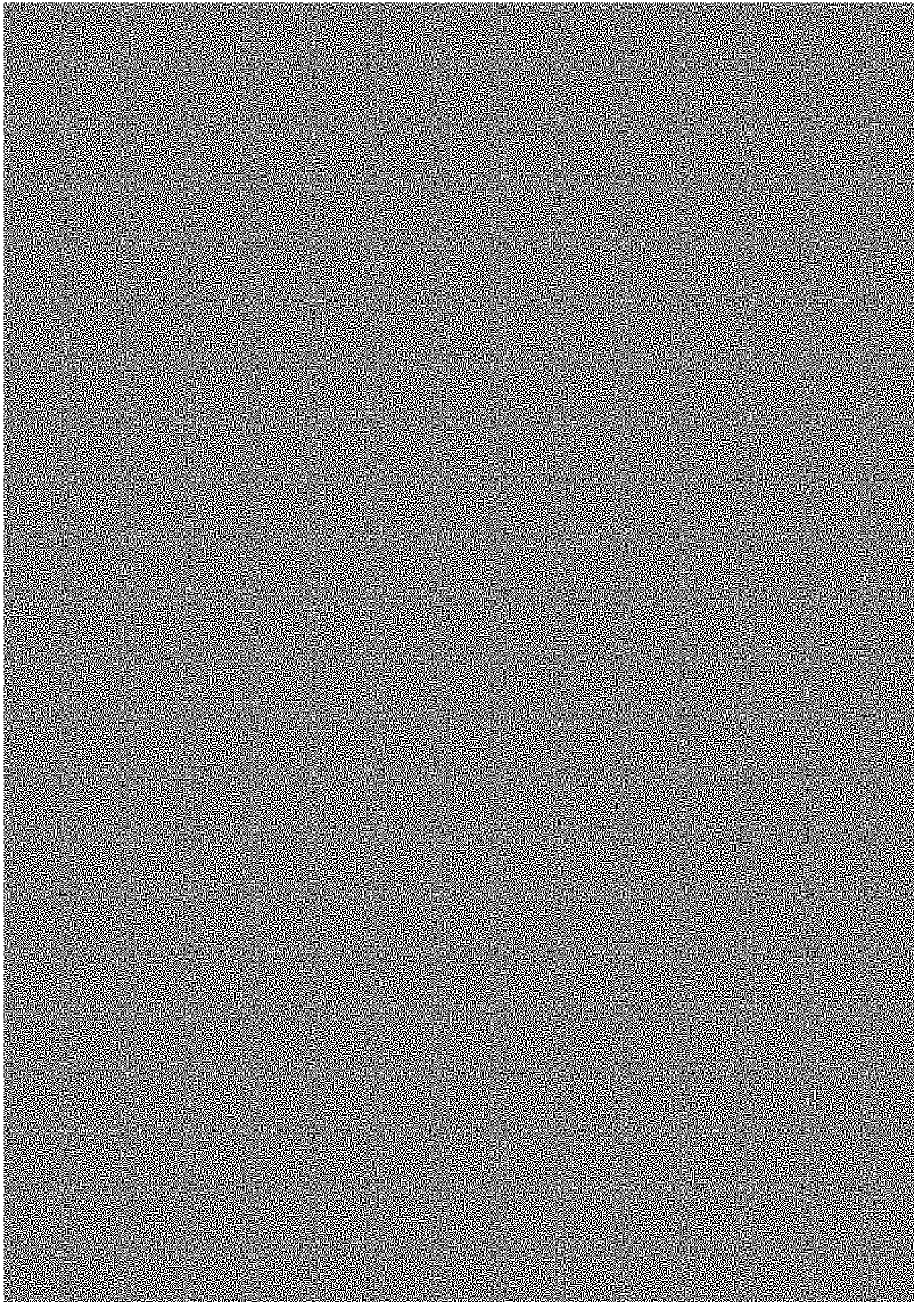
Acknowledgements

First and foremost I would like to express my thanks to my two supervisors, Dr. Simon Pomeroy and Dr. James Flint for being a constant source of support and ideas, and for tactfully pointing out when I'd done something stupid in my code. I am sure that the last three years would have been a lot harder without being able to meet with them both every week to bounce ideas around on squared paper a whiteboard!

Secondly I'd like to thank my family for being there for me every step of the way, both during my PhD and throughout my life so far. I love you all and cherish the support you've shown me in the last three years. Special mentions as follows: To my mum Linda; you have offered me help at every turn, shared in our mad conversations, and provided me with home-cooked meals every night for 9 months! Special thanks as well for volunteering for the task of proof-reading the majority of this Thesis, despite not having a clue what it was on about when you started reading! To my dad Gareth; We became season ticket holders at Leicester City during my PhD, and despite The Foxes' mixed fortunes during that time I can say that I've definitely loved the time we've spent bonding together. WE ARE PREMIER LEAGUE! My brother Jon; We are thick as thieves, and as always you have been there to share in lively debates about all things nerdy and football-related! In the last three years you (along with your fiancée Esme) have been an absolute inspiration to me as you've started your family together with your lovely daughter Emily and new-born son. Any time I've been lacking motivation I've looked to your dedication to your family as inspiration.

Friends are what keep you sane during the day-to-day slog of a PhD, you all know who you are, but I will take this opportunity to thank a few by name. Childhood friends Tommy, Zack, Kimberly, Billy, Trini and Jason for showing me how to battle (my) demons. Mike Lord, band-mate and fellow First Contact aficionado! You introduced me to the Battlefield franchise in 2011, and we've spent an unhealthy amount of time playing it since. My best and oldest friend Trish Standen - this Thesis has been fuelled in part by our B-movie and gin nights which I am going to miss as I become a honorary Londoner! Who'd have thought when we met at 11 years old, we'd both end up doctors?!

Reddit - without you, this Thesis would have been finished on time!



Contents

1 Introduction **1**

1.1 Novel aspects of the research 2

1.2 Development of the research methodology 3

1.3 Research objectives 4

1.4 Case study introductions 4

1.4.1 Desktop/HPC case study 4

1.4.2 Mobile device case study 5

1.5 Thesis summary 6

References 8

2 The TLM method **9**

2.1 Introduction to the TLM method 9

2.2 Two-dimensional nodes 10

2.2.1 2-D shunt nodes 13

2.2.2 2-D series nodes 16

2.2.3 Wave generation and propagation 19

2.2.4 Thévenin equivalent circuits 21

2.3 Three-dimensional nodes 24

2.3.1 The SCN 25

2.3.2 The stub-loaded SCN 30

2.4 Boundary conditions 36

2.4.1 Reflective boundaries 37

2.4.2 Absorbing boundaries 37

2.4.3 Periodic boundaries 39

2.5 Optimisations to the 3-D TLM algorithm 40
2.5.1 SCN computational optimisations 40
2.5.2 Stub-loaded SCN computational optimisations 43
2.5.3 Computational optimisations summary 45
2.6 Conclusion 46
References 47

3 Parallel computing strategies for TLM **49**
3.1 Review of current parallel computing strategies 49
3.1.1 Message passing interface (MPI) 49
3.1.2 OpenMP 50
3.1.3 Hybrid parallelisation 51
3.1.4 Open computing language (OpenCL) 52
3.2 Parallel computing & the TLM method 52
3.2.1 Graphics processing unit (GPU) strategies 53
3.2.2 Multicore & symmetric multiprocessor strategies 53
3.2.3 High performance computing (HPC) cluster strategies 54
3.3 Conclusions & impact on research strategy 56
References 58

4 Geometric modelling of raindrops **60**
4.1 Review of raindrop geometry modelling techniques 61
4.1.1 Review of available models 61
4.1.2 Previous investigations to date 63
4.1.3 Summary of the Beard & Chuang model 64
4.2 A rain-field modelling strategy 64
4.2.1 Calculating raindrop size occurence probabilities 65
4.2.2 Generating a discrete random variable for raindrop diameter . . . 67
4.2.3 Discretising the raindrop geometry 71
4.2.4 Collision detection strategies 76
4.2.5 Implemented collision detection algorithm 77
4.2.6 Summary of raindrop modelling method 79
4.3 Conclusion 82

References 84

5 TLM on large-scale computing platforms 86

5.1 Introduction 86

5.2 Large-scale computing limitations & considerations 87

5.2.1 General purpose computing on graphics processing units 87

5.2.2 Single instruction, multiple data (SIMD) 88

5.2.3 High-performance computing (HPC) clusters 88

5.2.4 Symmetric multiprocessing (SMP) hardware 89

5.2.5 Symmetric multiprocessing (SMP) software 89

5.2.6 Message passing interface (MPI) 90

5.3 Methodology 91

5.3.1 Theory of operation 91

5.3.2 Compute-node boundary data exchange 93

5.3.3 SMP acceleration strategy 97

5.3.4 Extracting simulation data 98

5.3.5 Summary 102

5.4 Results 103

5.4.1 Validation 103

5.4.2 Benchmarking 106

5.5 Performance analysis 108

5.6 Summary 112

References 113

6 TLM on mobile computing platforms 114

6.1 Introduction 114

6.2 Methodology 115

6.2.1 ARM NEON overview 116

6.2.2 Two-dimensional TLM with ARM NEON 117

6.2.3 A SIMD approach to the TLM method 118

6.2.4 Multi-threaded optimisations 121

6.2.5 Summary 122

6.3 Motivations for education as a use case 122

6.3.1 Limitations and considerations 122
6.3.2 Education as a use case 124
6.4 Benchmarking results 125
6.5 Performance analysis 126
6.6 Summary 128
References 130

7 Conclusions 132
7.1 Contributions to the field 132
7.2 Further work 133
7.2.1 Rain-field modelling performance 133
7.2.2 Modelling performance study using large-scale platforms 134
7.2.3 3-D simulations on mobile devices 134
7.2.4 Mobile solver usage study 135
7.2.5 Ad-hoc cluster computing using mobile devices 135
7.3 Overall conclusion 135

Appendices 137

Appendix A SketchUp raindrop meshing script 137

Appendix B 3-D SCN TLM solver documentation 156

Appendix C Mobile TLM solver documentation 205

Appendix D iOS mobile solver analytics data (28/01/2013 - 24/02/2014) 281

List of Abbreviations and Symbols

Abbreviations

Abbreviation	Expansion
ABC	Absorbing Boundary Condition
ACN	Asymmetrical Condensed Node
CAD	Computer-Aided Design
EM	Electromagnetic
EMC	Electromagnetic Compatibility
FDFEM	Frequency Domain Finite Element Method
FDTD	Finite Difference Time Domain method
HSCN	Hybrid Symmetrical Condensed Node
MoM	Method of Moments
MTBC	Matched Termination Boundary Condition
o/c	Open circuit
PEC	Perfect Electrically Conducting
RF	Radio Frequency
RFI	Radio Frequency Interference
s/c	Short circuit
SCN	Symmetrical Condensed Node
SSCN	Symmetrical Supercondensed Node
TDFEM	Time Domain Finite Element Method
TE	Transverse Electric
TEM	Transverse Electromagnetic
TLM	Transmission Line Modelling Method
TM	Transverse Magnetic
2-D	Two-dimensional
3-D	Three-dimensional

Circuit theory and TLM notation

Symbol	Denotes
C	Capacitance (F)
D_i	Node density (node.s^3.m^{-3})
f_c	TLM mesh cutoff frequency (Hz)
G	Conductance (S)
I	Total current (A)
$\Delta\ell$	Node dimension for a cubic cell (m)
L	Inductance (H)
R	Resistance (Ω)
$\mathbf{S}$	Scattering matrix
Δt	Time step (s)
T	TLM transmission coefficient
u_{TL}	Pulse propagation velocity (m s^{-1})
u_{TLM}	Wave velocity in the TLM mesh (m s^{-1})
V	Total voltage (V)
$\Delta x, \Delta y, \Delta z$	Node dimensions for a general rectangular cell (m)
Y	Transmission line characteristic admittance (S)
$\hat{Y}$	Normalised admittance = $\frac{Y}{Y_0}$
Z, Z_{TL}	Transmission line characteristic impedance (Ω)
Γ	TLM reflection coefficient
ζ	Stub coefficient

Electromagnetics, physical properties of materials, and waves

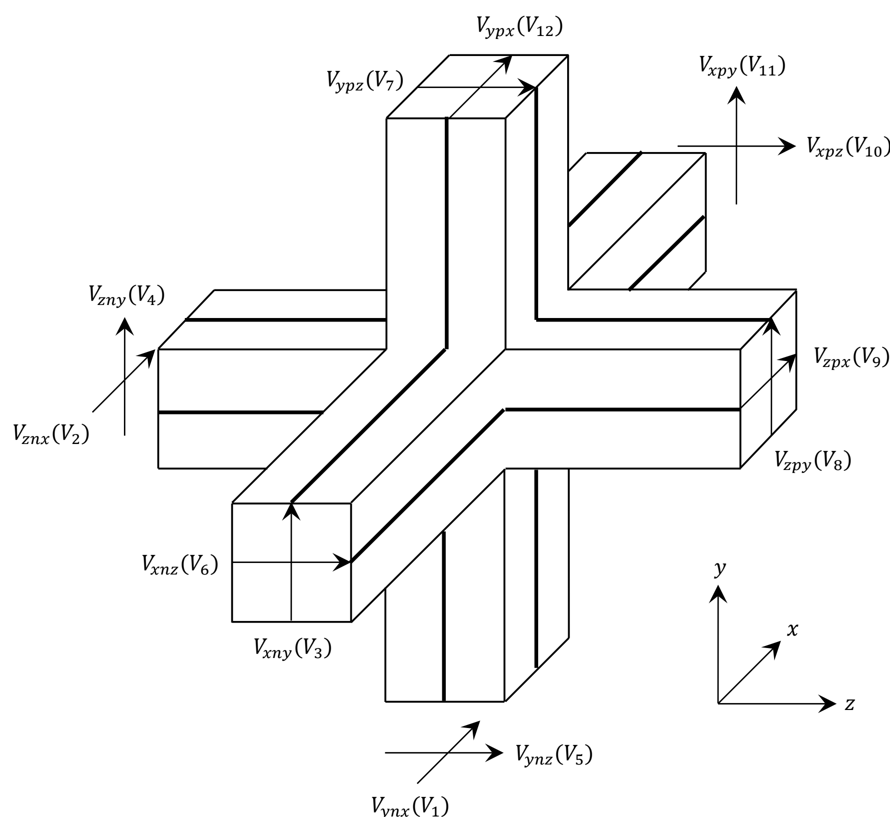
Symbol	Denotes
c	Speed of light in vacuo = 2.99792458×10^8 (m s^{-1})
$\mathbf{E}, E_x, E_y, E_z$	Electric field strength (V m^{-1})
f	Frequency (Hz)
$\mathbf{H}, H_x, H_y, H_z$	Magnetic field strength (A m^{-1})
$\mathbf{J}$	Current density (A m^{-2})
t	Time (s)
u	Wave velocity (m s^{-1})
Y_0	Intrinsic admittance of free space = $\frac{1}{Z_0}$ (S)
Z_0	Intrinsic impedance of free space ≈ 376.7303 (Ω)
ϵ	Permittivity (dielectric constant) (F m^{-1})
ϵ_r	Relative permittivity
ϵ_0	Permittivity of free space $\approx 8.854187 \times 10^{-12}$ (F m^{-1})
λ	Wavelength (m)
μ	Permeability (H m^{-1})
μ_r	Relative permeability
μ_0	Permeability of free space = $4\pi \times 10^{-7} \approx 1.256637 \times 10^{-6}$ (H m^{-1})
ν	Volume (m^3)
ρ	Reflection coefficient
ρ_T	Theoretical reflection coefficient
σ_e	Electric conductivity (S m^{-1})
σ_m	Magnetic conductivity (S m^{-1})
ϕ	Wave incident angle
ϕ_e	Estimate of wave incident angle
ω	Angular frequency (rad s^{-1})

Mathematics

Symbol	**Denotes**
a, b	Finite difference parameters
$a_n(x, y, z)$	Binary array at time step n
B_m	Boundary operator of order m
D	Spatial shift operator
I	Identity shift operator
i, j, k	Dummy indices
j	$\sqrt{-1}$
K	Time shift operator
$d\mathbf{l}$	Element of length
$\hat{\boldsymbol{n}}$	Unit length normal vector
$\hat{\boldsymbol{x}}, \hat{\boldsymbol{y}}, \hat{\boldsymbol{z}}$	Unit vectors in x-, y- and z-directions
$\left.\begin{matrix}\alpha_1, \ldots, \alpha_p \\ \beta_1, \ldots, \beta_p \\ \gamma_1, \ldots, \gamma_p\end{matrix}\right\}$	Boundary coefficients
π	Ratio of circle circumference to diameter ≈ 3.141593
$\nabla \times \mathbf{A}$	Curl of vector $\mathbf{A}$
∞	Infinity

Port labelling convention

Port labels used in this text comply with the following convention (Johns' original notation in parentheses):



Other indices denote the time step and the pulse propagation direction, e.g.:

$$ {}_{n+1}^{\;\;i}V_{zpx} $$

"The incident voltage on the z-positive directed link-line, polarised in the x-direction at time step $n + 1$."
A similar scheme is used for parameters associated with link and stub lines. Symmetry is implied if 'n' or 'p' is omitted from the label subscript, e.g. $Z_{xpy} = Z_{xny} = Z_{xy}$.

Superscript or subscript	**Denotes**
e	Electric
i	Incident
m	Magnetic
o	Open circuit
r	Reflected
s	Short circuit

List of Figures

2.1 An incident pulse V_i, arriving at the centre of a 2-D TLM node. 11

2.2 An incident pulse of 1 V, followed by two time-steps and the associated scattered port pulse values. 11

2.3 A lumped element model of a section of transmission line of length $\Delta\ell$. . 12

2.4 A 2-D shunt node assembled from coaxial tranmission line sections. [2.6] 13

2.5 A 2-D series node, assembled from coaxial tranmission line sections. [2.6] 17

2.6 A comparison of the mesh propagation velocity at 0° relative to a single axis, with respect to actual medium velocity. 21

2.7 A transmission link-line terminated by arbitrary network N. [2.6] 22

2.8 The Thévenin equivalent circuit for the link-line shown in Fig. (2.7). [2.6] 23

2.9 Thévenin equivalent circuit for the 2-D shunt node. [2.6] 23

2.10 Thévenin equivalent circuit for the 2-D series node. [2.6] 24

2.11 A non stub-loaded SCN, with Trenkić port notation (Johns' notation in parentheses). 25

2.12 An arbitrary, non-cubic block of space of dimensions Δx, Δy, Δz. [2.6] . . 33

2.13 An illustration of periodic boundaries in operation on all six faces of a TLM mesh volume. 39

4.1 Normal distribution for the approximated Beard & Chuang raindrop geometry model. The x-axis shows the raindrop diameter (mm), and the y-axis shows the probability of occurrence. 66

4.2 Comparison of the implemented discrete random variable (executed 10 000 times) with expected occurrence probability values calculated from section (4.2.1). 71

4.3 An illustration of the test used for drawing a circle of water-based nodes in the XY plane within the 3-D solver, showing center-point C, test-point P and radius r. 73

4.4 An illustration of the Voxelise process in operation, showing the voxel volume ($\Delta\ell^3$), as well as the probing vector V_p, used to test the centre-point of the voxel, C. 74

4.5 A raindrop model, shown with a bounding box and one row of the virtual mesh used during discretisation for an arbitrary value of $\Delta\ell$. 75

4.6 Raindrop geometry, shown with one row of the virtual mesh used during discretisation for an arbitrary value of $\Delta\ell$. Voxels set to behave as water are highlighted in dark red. 76

4.7 The same raindrop from Fig. (4.5) with its equivalent TLM mesh overlayed, meshed for a F_c of 10.053GHz. 77

4.8 The raindrop TLM mesh data for the raindrop in Fig. (4.5) as it will appear in the simulation, meshed for a F_c of 10.053GHz. 78

4.9 Collision detection between raindrops in close proximity using vectors. . 79

4.10 An example of several raindrops in close proximity, modelled using SketchUp. 80

4.11 A flow diagram of the steps involved in meshing a randomised rain-field from within SketchUp. 81

5.1 A flow diagram of the steps involved in importing the randomised raindrop field mesh information, ready for the main application to execute the simulation. 92

5.2 An overview of the mesh visualisation scheme, where individual slices through a 3-D mesh ***(a)*** are converted into a multi-page image ***(b)*** in order to represent the mesh at each time-step ***(c)***. The mesh dimensions are given by x, y and z and the time-step number is given by n. 99

5.3 A visualisation of the colour map that results from the calculations shown in (5.2). 99

5.4 An illustration of how MPI is used to improve performance via parallel file I/O, in an example system with four processing units. X_{DIM} and Y_{DIM} are sub-mesh dimensions and r is the processing unit number. The black squares represent file pointers which move concurrently as each processing unit writes to the file in parallel. 101

5.5 Frequency domain results showing resonant frequencies in a simulated 1 m^3 cavity. 105

5.6 The raw performance metrics of the 3-D SCN TLM solver when running on varying numbers of processing cores with no write operations to disk. (An average of three simulation runs). 107

5.7 The wall-clock execution times of the 3-D SCN TLM solver when running on varying numbers of processing cores with no write operations to disk. (An average of three simulation runs). 108

5.8 The wall-clock execution times of the 3-D SCN TLM solver when running on varying numbers of processing cores and with different stride values for the generated PPM image outputs. (An average of three simulation runs for each number of cores under test). 109

5.9 The performance figures of the 3-D SCN TLM solver when running on varying numbers of processing cores and with different stride values for the generated PPM image outputs, normalised to the raw performance figures shown in Fig. (5.7). 111

6.1 Illustration of the storage method within a 64-bit NEON register for two single precision floats. 117

6.2 An array of NEON registers holding neighbouring port values for two adjacent nodes to be processed. 118

6.3 An example of vector addition with single precision floats using NEON registers. 118

6.4 A visualisation of the simulated 1 m^2 cavity resonator used for benchmarking. 125

6.5 Normalised processing speed comparision between the ARM NEON SIMD TLM solver and the serial TLM solver. 127

D.1 Weekly user sessions figures for the iOS version of the mobile solver application. 282
D.2 User sessions observed over the course of any given 24 hour period for the iOS version of the solver application. 282
D.3 Weekly new user sessions figures for the iOS version of the mobile solver application. 283
D.4 New user sessions observed over the course of any given 24 hour period for the iOS version of the solver application. 283
D.5 Weekly active user sessions figures for the iOS version of the mobile solver application. 284
D.6 Active user sessions observed over the course of any given 24 hour period for the iOS version of the solver application. 284
D.7 Session length usage data across all users for the iOS version of the solver application. 285
D.8 Session per day usage data across all users for the iOS version of the solver application. 285
D.9 Session per week usage data across all users for the iOS version of the solver application. 286
D.10 Session per month usage data across all users for the iOS version of the solver application. 286
D.11 Usage retention percentages data for the first 30 days after installing the application. 287
D.12 User age distribution (based on the 20.8% of users where age is known) for the iOS version of the solver application. 287
D.13 iOS device family usage percentages across all sessions. 288

List of Tables

2.1 Operations per node for different optimisation techniques of the scatter process within TLM. [2.13] 43

4.1 Cosinusoidal distortion coefficients for the Approximated Beard & Chuang model. 64

4.2 Occurence probability $P(d)$ for raindrops of diameter d. 67

4.3 A representation of the array holding the discrete random variable V_d using a naive construction approach. 68

4.4 Discrete random variable verification results comparing observed and expected occurrence probability values, and showing relative error values. 71

5.1 The absolute and relative normalised performance figures for the raw 3-D SCN TLM solver based on the results as shown in Fig. (5.6). 109

6.1 Typical ARM NEON register configurations for 64-bit & 128-bit registers. 117

6.2 Benchmarking results comparing ARM NEON SIMD & serial-processing TLM solvers on mobile devices. 126

Author's publications

1. Browne, D.R.; Pomeroy, S.C.; Flint, J.A.; , "A mobile virtual electromagnetics laboratory for iPhone," *8th Intl. Conf. on Computation in Electromagnetics, CEM 2011* , pp.116-117, 11-14 Apr. 2011.

2. Browne, D.R.; Pomeroy, S.C.; Flint, J.A.; , "A framework for teaching electromagnetics concepts using mobile devices," *IEEE Intl. Symposium on Antennas and Propagation* , pp.1401-1404, 3-8 Jul. 2011.

3. Browne, D.R.; Pomeroy, S.C.; Flint, J.A.; , "Benchmarking of a TLM solver on mobile devices sharing a common hardware architecture," *Loughborough Antennas and Propagation Conf. (LAPC 2011)* , 14-15 Nov. 2011.

4. Browne, D.R.; Flint, J.A.; Pomeroy, S.C.; , "Raindrop modelling for high frequency propagation studies," *Loughborough Antennas and Propagation Conf. (LAPC 2011)* , 14-15 Nov. 2011.

5. Browne, D.R.; Flint, J.A.; Pomeroy, S.C.; , "Implementing a mobile electromagnetics laboratory," *Science, Measurement & Technology, IET* , vol.6, no.5, pp.398-402, Sept. 2012.

6. Browne, D.R.; Chouliaras, V.A.; Flint, J.A.; Pomeroy, S.C.; , "SIMD acceleration of a TLM solver using ARM processors," *Loughborough Antennas and Propagation Conf. (LAPC 2012)* , 12-13 Nov. 2012.

7. Browne, D.R.; Pomeroy, S.C.; Flint, J.A.; , "A discrete random variable for raindrop wave propagation modelling," *Antennas, Wireless and Electromagnetics, 1st IET Colloquium on* , pp.1-20, 29 May 2013.

8. Browne, D.R.; Pomeroy, S.C.; Flint, J.A.; , "Computational electromagnetics for

mobile devices on multiple platforms," *IEEE Intl. Symposium on Antennas and Propagation* , pp.2167-2168, 7-12 Jul. 2013.

9. Browne, D.R.; Pomeroy, S.C.; Flint, J.A.; , "Geometric modelling of raindrops with free-to-use tools," *ACES Journal* , submitted for review Feb. 2014.

Chapter 1

Introduction

In recent years, much of the investigations into accelerating the performance of computational electromagnetic (CEM) solver software applications has focussed on strategies specific to commodity desktop computer systems. Many of these recent implementations utilise graphics processing units (GPUs) in order to improve the solver performance past that which is achievable from the central processing unit (CPU) alone. Such implementations are however relatively difficult to port to different hardware architectures, and also have limited performance expandability. This is due to the physical space limitations inherent with the use of several GPUs in a single system.

Traditional parallel computing strategies for accelerating the performance of CEM solver applications therefore present an opportunity to investigate a software implementation that is flexible, and easily portable between desktop and larger-scale distributed computing systems. Developing a CEM solver application with such expandability allows users access to much greater processing power for simulation execution, and therefore simulations may be modelled with a much higher resolution and maximum frequency features. Utilising parallel processing in this way allows simulation execution time frames to remain practical for end users.

Modern mobile computing devices available in recent years, such as *smartphones* and *tablets* are increasing in performance and features in much a similar fashion to desktop computing in throughout the 1990s - 2000s. Similar parallel processing strategies to those employed for a desktop CEM solver implementation can be carried over to these

small-scale computing devices in order to execute relatively complex simulations in a timely fashion.

This thesis explores TLM solver implementations acclerated by varying parallel processing techniques, selected based on the selected desktop and mobile hardware platforms in use. These investigations were carried out in the context of these two extremes of the modern computing *spectrum*, and relevant case study applications were selected in each case.

1.1 Novel aspects of the research

There is a need for a new perspective concerning TLM implementations that are optimised for parallel-computing architectures in both the large-scale and small-scale computing cases. It is therefore desirable for the research to work towards the implementation of parallelised TLM solver applications that are specialised for a simulating a particular class of CEM problem, dependent on whether the solver is to be run on a large-scale or small-scale computing platform. Optimising performance and functionality for a specific class of problem also presents the possibility to produce results more efficiently than the commercial alternatives, which are generally designed to operate on much wider types of scenarios.

There are a number of simulation applications that would benefit greatly from the performance increases inherent in the implementation of a parallelised CEM solver. One such class of problems that has been identified for a problem-specific TLM solver is millimetre wave attenuation and depolarisation due to rain fields. EM transmissions using frequencies in the Super High Frequency (SHF) band and Extremely High Frequency (EHF) band (corresponding to frequencies of 3 – 30 GHz, and 30 – 300 GHz respectively) have extremely short wavelengths. Increasing the frequencies of EM transmission through these two bands results in an increasing dominance of the size of individual raindrops with respect to the signal transmission wavelength. This in turn produces increasingly adverse effects on signals as transmission wavelengths approach the same order of magnitude as the dimensions of raindrops, and even more so as the drop dimensions become significantly larger than the transmission wavelengths in the

EHF band.

A highly-optimised parallel TLM solver with specific configuration options for generating and testing EM wave depolarisation effects within different rain field distributions or due to individual drops is therefore an appropriate research goal. In order to model the small-scale geometries of raindrops, a finely discretised mesh is required for the TLM simulation. The increased amount of nodes in fine-mesh simulations increases overall simulation time in traditional solvers. Implementing a new TLM solver using a parallel computing architecture will allow increased simulation speed with respect to general purpose solvers, despite the increase in the complexity of the simulations to be carried out. The resulting software should be a high-performance, parallelised, problem-specific TLM solver offering comprehensive configuration options tailored towards rain field and individual drop simulations, and other problems which necessitate a brute-force approach.

1.2 Development of the research methodology

The project aims to develop and investigate strategies for obtaining the best performance when conducting CEM-based simulations using TLM on parallel computing platforms. The project will analyse results to quantitatively demonstrate that the solver software to be implemented for use on HPC cluster architectures is significantly faster for a given simulation set-up than the same simulation running on a commercial solver in a normal desktop environment. By gathering results using varied parallelisation techniques, the optimum performance set up will be identified, and its characteristic will be analysed. In order to verify the final 3-D TLM solver software, there is a need for a suitable benchmark in order to compare the simulation results of the software with a commercial software package. During the testing phase of the software development, equivalent simulations will be undertaken in CST MicroStripes, a popular commercial CEM solver. This will produce a set of results that can be treated as a benchmark for both result accuracy and simulation time for each simulation to be completed.

1.3 Research objectives

The research goals which have been referred to in the above section can be summarised as follows:

1. Produce two implementations for TLM solvers that are optimised for desktop/HPC systems and mobile computing devices respectively.
2. Verify both TLM implementations for their mathematical accuracy utilising accepted empirical techniques for time and frequency-domain verification.
3. Benchmark the performance gains observed when using different parallel processing strategies to optimise the TLM solver implementations in both cases.
4. Based on benchmark observations, utilise suitable parallel processing paradigms that will offer the best performance when using typical cluster computer architectures (in the large-scale case) or typical mobile computing processors (in the mobile device case).
5. Study the depolarisation effects on single raindrops at various frequencies, comparing the implemented solver with an equivalent simulation using CST MicroStripes or similar software to validate the results produced by the implemented solver.
6. Identify an area of the EM spectrum between 6 – 300 GHz that warrants further investigation in terms of the depolarisation effects due to rain, and use the implemented solver to conduct the investigation.

1.4 Case study introductions

1.4.1 Desktop/HPC case study

In the Extremely High Frequency (EHF) range of 30 – 300 GHz, EM radiation has wavelengths between 10–1 mm, giving rise to the name 'millimetre waves'. At these frequencies, signals are very susceptible to attenuation due to atmospheric conditions, limiting applications by distance and line of sight. Attenuation due to rain and humidity

levels has a dramatic effect even over relatively short distances (when compared to radio signals operating in the lower frequency bands). There is also an added complication in the band between 57 – 64 GHz, where the resonance of oxygen molecules present in the atmosphere attenuates signals to an even greater extent.

To simulate the attenuation and depolarisation effects on radio signals that are caused by rain fields, the simulation must recreate the characteristics of rain as accurately as possible. To achieve this, firstly the geometry of individual drops must be recreated based on previous investigations [1.1, 1.2, 1.3]. To complete the modelling process, a representative size distribution must also be implemented within the simulating environment. The final consideration that must be taken into account is the relative permittivity of water, which changes as a function of the EM signal frequency that it is subject to. Accurately recreating all of these factors should result in a model, which behaves in a very similar way to experimental observations.

1.4.2 Mobile device case study

Any TLM implementation for mobile computing devices will be limited in performance compared to a typical desktop implementation. However, it is also likely that the high resolution touch-sensitive screens present on the majority of such devices should also be utilised for user interactivity and dynamic visual feedback. For these reasons, the number and complexity of the TLM-related calculations should be minimised in order to maximise perceived software performance. A 2-D TLM implementation is an obvious solution to address these performance limitations. This however restricts the complexity of the environments that are to be modelled with a mobile device implementation.

An educational learning-aid application presents an ideal use case for such an implementation, as a range of simple to intermediate phenomena can be demonstrated to students at a high school or undergraduate level in a dynamic and engaging manner, whilst also providing user interactivity via the touch-screen on the mobile device.

1.5 Thesis summary

Chapter 2

An introduction to the TLM method, including the analogies to Maxwell's equations. Derivations of the various properties of 2-D and 3-D methods are given, as well as discussion concerning boundary condition implementations. Finally, discussion is included on optimisations that can be made to the TLM algorithm in order to reduce the number of mathematical operations that occur during a simulation.

Chapter 3

A review of the parallel computing strategies suitable for use with the TLM method, including discussion of previous work investigating the use of the TLM method with the different parallel computing methods. Based on these findings, a research strategy is discussed centring on investigations using mobile device and Desktop & HPC cluster platforms.

Chapter 4

A number of techniques for modelling raindrops on the individual, geometric level are reviewed based on previous work in the field. Discussion of previous investigations using the models is given, as well as an assessment of the overall accuracy of each model as compared to real-world observations. A novel method for generating randomised raindrop fields is presented, adhering to size distribution constraints whilst avoiding raindrop intersections during their random placement.

Chapter 5

A highly-optimised 3-D TLM solver utilising a hybrid parallel processing approach is presented. A discussion of its implementation is included, discussing limitations and considerations, benchmarking results and performance analysis. A summary of

the features of the solver is also included, focussing on optimisations allowing for the modelling of raindrop fields in the EHF band and above.

Chapter 6

The first implementation of the 2-D TLM algorithm on mobile computing devices is presented. A discussion of its implementation using the iOS platform is presented, including limitations and considerations, benchmarking results and performance analysis. A summary of the features of the solver is also included, focussing on its use as an educational learning aid. Finally, a summary is given discussing the potential for CEM solver applications on mobile devices in the future.

Chapter 7

A conclusion of the findings and work undertaken for this thesis, as well as proposals for areas that warrant further investigation. A set of final conclusions are also given.

References

[1.1] H. R. Pruppacher and R. L. Pitter, “A semi-empirical determination of the shape of cloud and raindrops,” *Journal of the Atmos*, vol. 28, no. 1, pp. 86–94, 1971.

[1.2] K. V. Beard and C. Chuang, “A new model for the equilibrium shape of raindrops,” *Journal of the Atmospheric Sciences*, vol. 44, no. 11, pp. 1509–1524, 1987.

[1.3] K. V. Beard, V. N. Bringi, and M. Thurai, “A new understanding of raindrop shape,” *Atmospheric Research*, vol. 97, no. 4, pp. 396–415, 2010.

Chapter 2

The TLM method

The TLM method was originally developed in the early 1970s [2.1], and since its introduction it has been the subject of much research. It is now a well-established numerical method for modelling wave propagation, and it is often used in research regarding computational electromagnetics, as well as acoustics. This chapter presents an overview of the TLM method, including derivation of 2-D and 3-D node types; wave propagation and generation; modelling lossy and inhomogeneous materials; discussion of boundary conditions; and computational optimisations to the TLM algorithm that can be made before applying parallel processing techniques to it.

2.1 Introduction to the TLM method

Initial attempts to map Maxwell's equations to a set of equivalent electrical circuits were first published in the 1940s [2.2], however further investigations at the time were limited due to the nascent nature of the digital computer industry. By the early 1970s, the performance of commercially-available computers had risen to a point where it was practical to use them to conduct the calculations required to model Maxwell's equations. Peter Johns was the originator of the TLM method, for the study of wave scattering [2.1]. A proposal for a formal definition of a TLM node, first came in the form of 2-D shunt and series nodes [2.3].

The TLM method is a time-domain technique for numerical modelling of wave propaga-

tion, and associated effects such as diffraction, dispersion, attenuation and depolarisation. Wave propagation is simulated in agreement with Huygen's Principle [2.4]. TLM models a given environment discretely in both space and time, in which the contents of the model can be solved precisely, whilst presenting an algorithm that is inherently stable. Electromagnetic field components are modelled by taking advantage of the analogous behaviour of materials with a mesh of intersecting simulated transmission lines.

Each point in the mesh where sectional lengths of transmission lines, or link-lines intersect, is called a node [2.5]. Modelling wave propagation is achieved in a two-step process, firstly by monitoring incident voltage pulses towards each node from each link-line. Based on the link-line characteristics, these voltage pulses are then scattered back outwards from each node towards its neighbouring nodes.

Modelling environments using TLM is possible in both 2-D and 3-D, where 3-D nodes build upon the derivation of 2-D nodes, in order to model more complex environments in three-dimensional space. It should be noted that whilst some 3-D node types do not have a simple circuit analogue, they instead exploit the behaviour of basic physical laws such as the conservation of energy, in their derivations.

2.2 Two-dimensional nodes

If we assume that link-lines in a 2-D mesh all have an equal characteristic impedance Z_{TL}, then all voltage pulses incident on a node via a link-line will experience an impedance mismatch. In the case of 2-D TLM, a voltage pulse incident on any one of the four node link-lines will see the three remaining link-lines in parallel, with a total impedance of $\frac{Z_{TL}}{3}$, as shown in Fig. (2.1). The transmission and reflection coefficients for 2-D TLM for a pulse incident on a single link-line are obtained via the methods shown in (2.1) and (2.2).

$$\rho = \frac{\frac{Z_{TL}}{3} - Z}{\frac{Z_{TL}}{3} + Z} = -0.5 \tag{2.1}$$

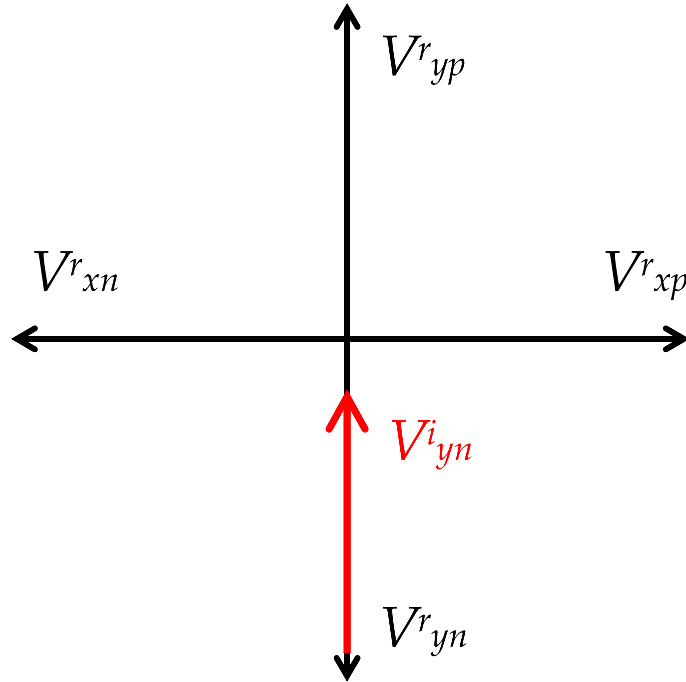


Figure 2.1: An incident pulse V_i, arriving at the centre of a 2-D TLM node.

$$\tau = \frac{2(\frac{Z_{TL}}{3})}{\frac{Z_{TL}}{3} + Z} = 0.5 \tag{2.2}$$

The scatter and connect process of a 2-D simulation is illustrated in Fig. (2.2), where a 1 V pulse is injected into the "south-facing" link-line of the centre node. This behaviour is in agreement with Huygens' principle, as it is shown that each node is a secondary, isotropic radiator for energy within the mesh [2.4]. The net effect is circular, symmetrical wave-front propagation across the mesh from the original point source.

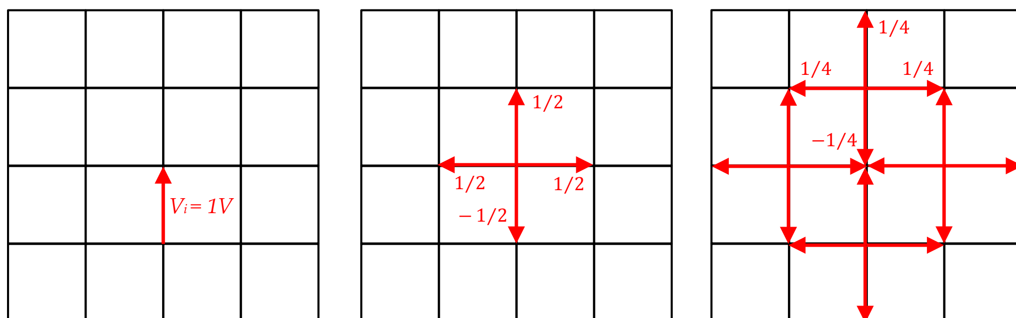


Figure 2.2: An incident pulse of 1 V, followed by two time-steps and the associated scattered port pulse values.

When a 2-D TLM node is taken in isolation, removing it from its surrounding neigh-

bours, it is a trivial task to derive the fundamental properties of the node. In order to do this, a lumped element model of a length of transmission line must first be considered. Fig. (2.3) shows an illustration of such a model, where C and L denote a capacitance per unit length and inductance per unit length respectively.

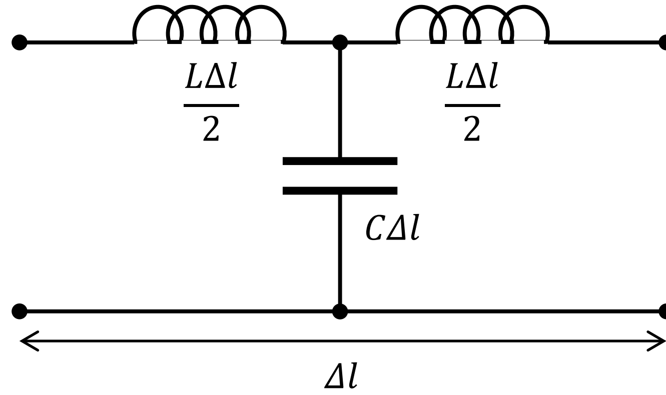


Figure 2.3: A lumped element model of a section of transmission line of length $\Delta\ell$.

As shown above, the length of link-lines within the node is equal to the node size $\Delta\ell$. If a voltage pulse is injected into one of these lines, it will take some small amount of time, Δt to propagate from one end of the line to the other. This will also induce a current in the line, which can be expressed as in (2.3).

$$i = \frac{\Delta Q}{\Delta t} = CV_p \frac{\Delta \ell}{\Delta t} \tag{2.3}$$

From this the propagation velocity of the line, u can be derived. The magnetic flux, Φ relates to the line inductance and current as shown in (2.4). Using Faraday's law, the rate of change of flux and line voltage are equivalent, as shown in (2.5). This can then be rearranged to solve for the pulse propagation velocity, u_{TL} as shown in (2.6).

$$\Phi = L\Delta\ell i \tag{2.4}$$

$$V_p = \frac{\Delta \Phi}{\Delta t} = LCV_p u^2 \tag{2.5}$$

$$u_{TL} = \frac{1}{\sqrt{LC}} \tag{2.6}$$

Related to this is the characteristic impedance of the link-lines in the mesh, which is as

shown in (2.7).

$$Z_{TL} = \sqrt{\frac{L}{C}} \tag{2.7}$$

2.2.1 2-D shunt nodes

When using the shunt configuration, the link-lines that form the node at their central intersection are connected in parallel. An illustration of this is shown in Fig. (2.4). For simulations that are executed on a shunt node mesh, TM-mode propagation is modelled. That is, the only non-zero components of the waves are H_x, H_y and E_z. However, it should be noted that this is based on the convention that $\frac{V_z}{\Delta z} \rightarrow E_z$, and mapping this quantity to different field components allows for modelling of TE modes. Using a Cartesian coordinate system, the partial differential equations governing the 2-D shunt node mesh can be expressed as shown in (2.8), with the relevant Maxwell equivalents (2.9).

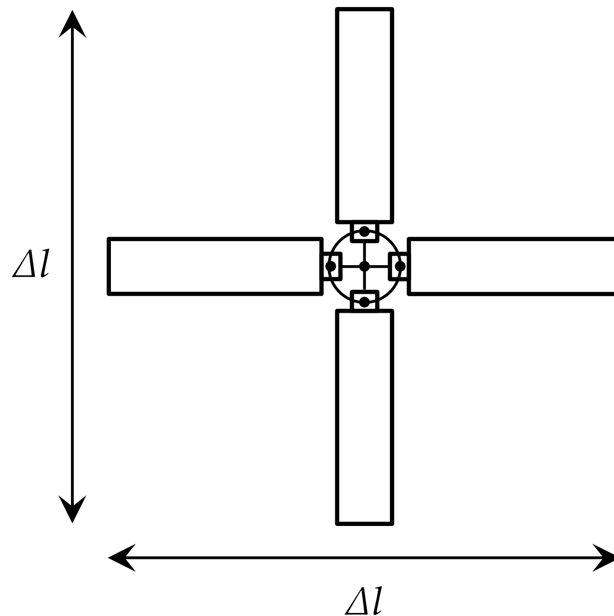


Figure 2.4: A 2-D shunt node assembled from coaxial tranmission line sections. [2.6]

$$\frac{\partial E_z}{\partial y} = -\mu \frac{\partial H_x}{\partial t}$$

$$\frac{\partial E_z}{\partial x} = -\mu \frac{\partial H_y}{\partial t} \tag{2.8}$$

$$\frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y} = \epsilon \frac{\partial E_z}{\partial t}$$

$$\frac{\partial V_z}{\partial y} = -L \frac{\partial I_x}{\partial t}$$

$$\frac{\partial V_z}{\partial x} = -L \frac{\partial I_y}{\partial t} \tag{2.9}$$

$$\frac{\partial I_x}{\partial x} + \frac{\partial I_y}{\partial y} = -2C \frac{\partial E_z}{\partial t}$$

Differentiating each equation along with the Maxwell equivalent, with respect to x, y and t in turn then removing magnetic field components, results in a pair of scalar wave equations that are equivalent, as shown in (2.10).

$$\begin{aligned} \frac{\partial^2 E_z}{\partial x^2} + \frac{\partial^2 E_z}{\partial y^2} &= \mu\epsilon \frac{\partial^2 E_z}{\partial t^2} \\ \frac{\partial^2 V_z}{\partial x^2} + \frac{\partial^2 V_z}{\partial y^2} &= 2LC \frac{\partial^2 V_z}{\partial t^2} \end{aligned} \tag{2.10}$$

This pair of equivalent equations can be used to derive an expression for the propagation velocity, both in the medium that is being modelled, and the 2-D shunt node TLM mesh, as shown in (2.11).

$$u = \frac{1}{\sqrt{\mu\epsilon}}$$

$$u_{TLM} = \frac{1}{\sqrt{2LC}} = \frac{1}{\sqrt{2}} \cdot \frac{1}{\sqrt{LC}} \tag{2.11}$$

It is therefore apparent that the propagation velocity of the 2-D TLM mesh is $\frac{1}{\sqrt{2}}$ times greater than that of the actual medium that is to be modelled. Providing the node size $\Delta\ell$ is sufficiently small, a network will behave in agreement with Maxwell's equations, and model wave interactions accurately. In the mesh, C is equivalent to ϵ and L is equivalent to μ. The relationships between the field components E_z, H_x and H_y, and the port voltages within a particular node are shown in (2.12).

$$E_z \leftrightarrow -\frac{V_z}{\Delta\ell} = -\frac{1}{2\Delta\ell} \cdot (V^i_{xn} + V^i_{xp} + V^i_{yn} + V^i_{yp})$$

$$H_x \leftrightarrow -\frac{I_y}{\Delta\ell} = -\frac{1}{Z_{TL}\Delta\ell} \cdot (V^i_{yp} - V^i_{yn}) \tag{2.12}$$

$$H_y \leftrightarrow -\frac{I_x}{\Delta\ell} = -\frac{1}{Z_{TL}\Delta\ell} \cdot (V^i_{xp} - V^i_{xn})$$

These equivalences between the port voltages and field components are central to the scattering procedure, as shown earlier in (2.8) and (2.9). In the 2-D shunt node case, the scattering procedure that produces a vector of reflected voltages $\mathbf{V}^r$, can be expressed as a multiplication of a scattering matrix $\mathbf{S}$ with a vector of incident voltage pulses $\mathbf{V}^i$, as shown in (2.13). For a 2-D shunt node mesh, the scattering matrix, $\mathbf{S}$ is defined as shown in (2.14). The incident voltages vector, $\mathbf{V}^i$ is defined by the voltages present at the relevant ports immediately neighbouring the current node to the north, south, east and west, as shown in (2.15).

$$\mathbf{V^r} = \mathbf{S} \cdot \mathbf{V^i} \tag{2.13}$$

$$\mathbf{S} = \frac{1}{2}\begin{bmatrix} -1 & 1 & 1 & 1 \\ 1 & -1 & 1 & 1 \\ 1 & 1 & -1 & 1 \\ 1 & 1 & 1 & -1 \end{bmatrix} \tag{2.14}$$

$$\mathbf{V}^i = \begin{bmatrix} V^i_{xn} \\ V^i_{xp} \\ V^i_{yn} \\ V^i_{yp} \end{bmatrix} \tag{2.15}$$

2.2.2 2-D series nodes

It is possible to implement a similar Cartesian 2-D mesh using transmission lines that are connected in series instead of parallel, as has been shown previously. An illustration of this arrangement, as well as the Thévenin equivalent circuit, is shown in Fig. (2.5). The mesh partial differential equations and their Maxwell equivalents are derived in a similar fashion as in the 2-D shunt node case, as shown in (2.16) and (2.17). However, it is noted in the series node case that $\Sigma V = -L\frac{dI}{dt}$ and $I = -C\frac{dV}{dt}$ in each direction of propagation. In the series node case, a TE field is presented, (i.e. non-zero field components E_x, E_z and H_y are modelled).

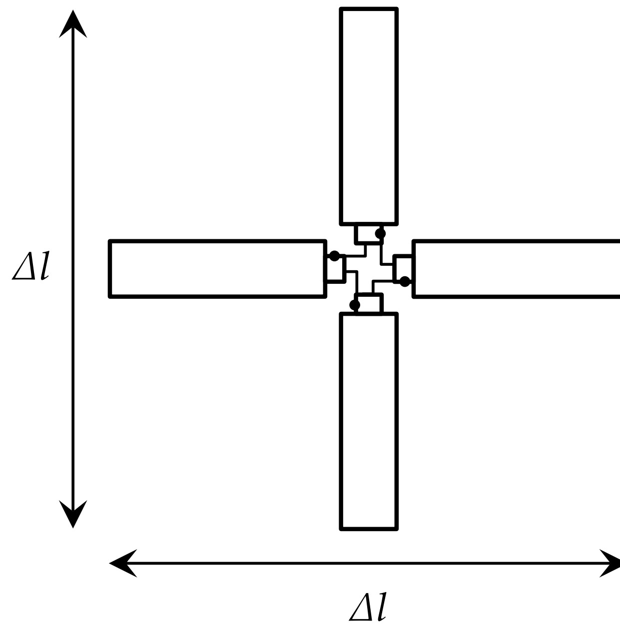


Figure 2.5: A 2-D series node, assembled from coaxial tranmission line sections. [2.6]

$$\frac{\partial E_x}{\partial z} - \frac{\partial E_z}{\partial x} = -\mu \frac{\partial H_y}{\partial t}$$
$$-\frac{\partial H_y}{\partial z} = \epsilon \frac{\partial E_x}{\partial t} \qquad (2.16)$$
$$\frac{\partial H_y}{\partial x} = \epsilon \frac{\partial E_z}{\partial t}$$

$$\frac{\partial V_x}{\partial z} + \frac{\partial V_z}{\partial x} = -2L \frac{\partial I_y}{\partial t}$$
$$\frac{\partial I_y}{\partial z} = -C \frac{\partial V_x}{\partial t} \qquad (2.17)$$
$$\frac{\partial I_y}{\partial x} = -C \frac{\partial V_z}{\partial t}$$

A pair of scalar wave equations are derived in a similar fashion to that used for 2-D shunt node networks. For series node networks, the electric field components are eliminated, as shown in (2.18).

$$\frac{\partial^2 H_y}{\partial x^2} + \frac{\partial^2 H_y}{\partial z^2} = \mu\epsilon \frac{\partial^2 H_y}{\partial t^2}$$
$$\frac{\partial^2 I_y}{\partial x^2} + \frac{\partial^2 I_y}{\partial y^2} = 2LC \frac{\partial^2 I_y}{\partial t^2} \tag{2.18}$$

The propagation velocity equation is identical to that shown for 2-D shunt node networks. The equivalence between the port voltages associated with each node and the modelled field components is also derived similarly to those for 2-D shunt node networks, as shown in (2.19).

$$H_y \leftrightarrow -\frac{I_y}{\Delta\ell} = \frac{1}{4Z_{TL}\Delta\ell} \cdot (V^i_{znx} + V^i_{xpz} - V^i_{zpx} - V^i_{xnz})$$
$$E_x \leftrightarrow -\frac{V_x}{\Delta\ell} = \frac{1}{\Delta\ell} \cdot (V^i_{znx} + V^i_{zpx}) \tag{2.19}$$
$$E_z \leftrightarrow -\frac{V_z}{\Delta\ell} = -\frac{1}{\Delta\ell} \cdot (V^i_{xpz} + V^i_{xnz})$$

The scattering matrix, **S** for series node networks, is defined as shown in (2.20). Similarly, the incident voltage vector $\mathbf{V}^i$ contains different values, due to the altered arrangement of the equivalent circuit, as shown in (2.22).

$$\mathbf{S} = \frac{1}{2} \begin{bmatrix} 1 & 1 & 1 & -1 \\ 1 & 1 & -1 & 1 \\ 1 & -1 & 1 & 1 \\ -1 & 1 & 1 & 1 \end{bmatrix} \tag{2.20}$$

$$\mathbf{V}^i = \begin{bmatrix} V^i_{xnz} \\ V^i_{xpz} \\ V^i_{zn} \\ V^i_{zpx} \end{bmatrix} \tag{2.21}$$

2.2.3 Wave generation and propagation

Wave generation

Waveforms can be generated in a TLM mesh by adding, or *injecting* voltage pulses into the mesh at the desired node locations. These voltage impulses are added to those already present within the mesh at the location of interest. Non-trivial waveform behaviours of specific frequencies can be simulated by appropriately varying the amplitude of the injected voltage pulses over a number of subsequent time-steps.

It should be noted that care must be taken in the process of attempting to drive waveform propagation in a TLM mesh. Undesirable (and as a result, inaccurate) results are easily obtained if attention is not paid to the range of potential frequency components of specific waveforms that are injected into a mesh.

As explained previously in Section (2.2.1), there are limits on the wave propagation velocity within a TLM mesh that are directly dependent on the frequencies of any propagating wave-fronts within the mesh. As an example, injecting a step-change impulsive waveform moving between 0 V to 1 V and back to 0 V over subsequent time-steps (i.e. giving the waveform a period of Δt) would result in a broadband waveform with many high-frequency components. Many of these frequencies would likely be above the threshold of acceptability in order to guarantee accurate results with few dispersive artefacts present during propagation.

When generating waveforms for TLM simulations, band-limited energy sources are used to minimise the excitation of spurious modes within the simulation. The following section discusses the derivation of the mesh frequency limits which are typically adhered to during TLM simulations.

Wave propagation

Due to the Cartesian nature of the TLM mesh, the distance required to travel along link-lines in order to reach the equivalent point in space represented by a node location is dependent on the angle of propagation away from a wave-front source. This gives rise to dispersive effects which vary in severity with the propagation angle.

In the case of wave propagation at 45° relative to the x and y axes, the TLM mesh propagation velocity can be shown to be:

$$u_{45^\circ} = \frac{distance}{time} = \frac{\sqrt{2}\Delta\ell}{2\Delta t} = \frac{u}{\sqrt{2}} \tag{2.22}$$

This agrees with the expressions for mesh propagation velocity shown previously in Section (2.2.1).

It was shown previously that in the case of fine-mesh simulations, a relationship between the mesh velocity, u_{TLM} and the propagation velocity of the actual medium, u can be expressed as in the form shown in (2.23).

$$u_{TLM} = \frac{1}{\sqrt{2}}u \tag{2.23}$$

Comparing these two results illustrates that wave-fronts propagating at 45° travel at the same velocity as the mesh propagation velocity.

However, waves propagating at 0° relative to the x or y axis behave much differently. When analysing the propagation of waves parallel to a single axis, an association between u_{TLM} and u can be derived as shown below in (2.24) [2.7].

$$\frac{u_{TLM}}{u} = \frac{\pi\left(\frac{\Delta\ell}{\lambda_0}\right)}{asin\left[\sqrt{2}sin\left(\pi\frac{\Delta\ell}{\lambda_0}\right)\right]} \tag{2.24}$$

The ratio between u_{TLM} and u is shown to be dependent upon $\Delta\ell$ and λ_0. A dramatic difference in the ratio between mesh velocity and medium velocity is observed when comparing the 45° result with a 0° result across a range of values of $\frac{\Delta\ell}{\lambda_0}$. This is illustrated below in Fig. (2.6).

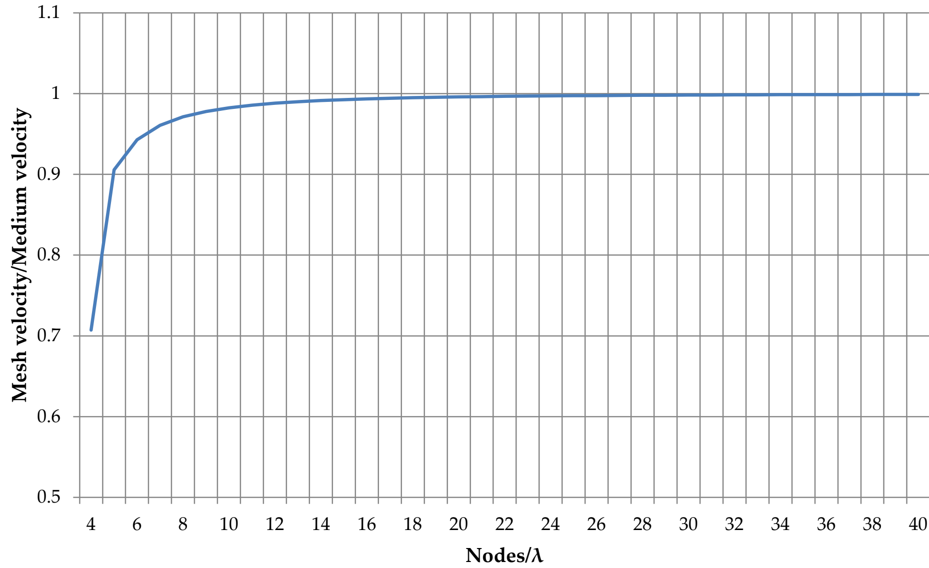


Figure 2.6: A comparison of the mesh propagation velocity at 0° relative to a single axis, with respect to actual medium velocity.

The results show that as expected, a finer mesh (i.e. a higher value of nodes/λ) allows for a closer approximation to the actual medium propagation velocity in directions other than 45°, of which the 0° case is the worst case scenario. At 10 nodes/λ, the mesh propagation velocity anomaly is less than 1.8 percent, and this reduces further to less than 0.5 percent at 20 nodes/λ.

In typical simulations, the intended nodes/λ value of the mesh therefore dictates the maximum frequency component that can be modelled within the simulation to the desired level of accuracy. A value of 10 nodes/λ limit is often used, as the average velocity anomaly across all frequencies in a simulation will be lower than the quoted maximum of 1.8 percent, which only occurs at the cut-off frequency and a 0° propagation direction. The 10 nodes/λ limit also represents a sensible compromise when considering the nodes/λ value of a simulation mesh directly impacts the equivalent real-world volume that can be represented for a given amount of RAM available for use.

2.2.4 Thévenin equivalent circuits

In order to aid the analysis of the scattering process, both internally within the node and in-between nodes, a simplified version of the node circuit is required. One such method is the construction of a *Thévenin equivalent circuit*.

Fig. (2.7) shows a length of transmission line connected to an arbitrary network, N. In this context, the transmission line will be replaced by a Thévenin equivalent circuit of a voltage source coupled with a series-connected impedance which allows the voltage and current solutions within network N to be determined.

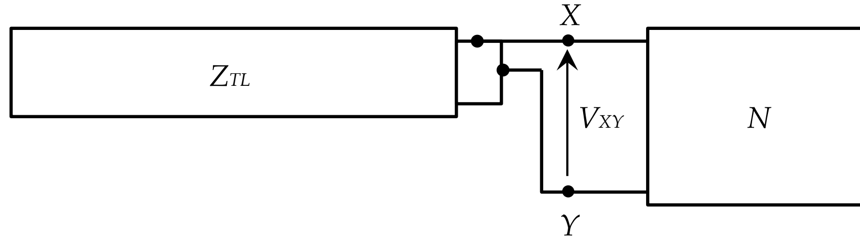


Figure 2.7: A transmission link-line terminated by arbitrary network N. [2.6]

A reference point is chosen (terminals X and Y in Fig. (2.7)) as the observation point. Firstly, the open circuit voltage V_{oc} is found across these terminals. This is the Thévenin voltage source. In this case when the transmission line is un-terminated, the current flowing at XY is equal to zero. Therefore, the magnitude of the reflected current must be equal to the incident current, as shown in (2.25).

$$\frac{V^i}{Z_{TL}} = \frac{V^r}{Z_{TL}} \tag{2.25}$$

At this stage, applying Kirchoff's Voltage Law gives the relationship between V_{oc}, V^i and V^r, as shown in (2.26).

$$V^r = V_{oc} - V^i \tag{2.26}$$

The open circuit voltage, V_{oc} can now be determined by substituting for V^r in terms of V^i into (2.26). Re-arranging this expression results in a value for V_{oc} as shown in (2.27).

$$V_{oc} = 2V^i \tag{2.27}$$

The value for the Thévenin impedance is determined by observing the impedance at terminals XY, which in this case is equal to the characteristic impedance of the

transmission line, Z_{TL}.

The resulting Thévenin equivalent circuit is shown in Fig. (2.8). The same procedure can be applied to the four link-lines present in both shunt and series 2-D nodes in order to derive an equivalent circuit for both nodes, as shown in Fig. (2.9) and (2.10). It should be noted that arbitrary values are stated for the link-line admittances (Y_{xz} and Y_{yz}) and the characteristic impedances (Z_{xz} and Z_{zx}).

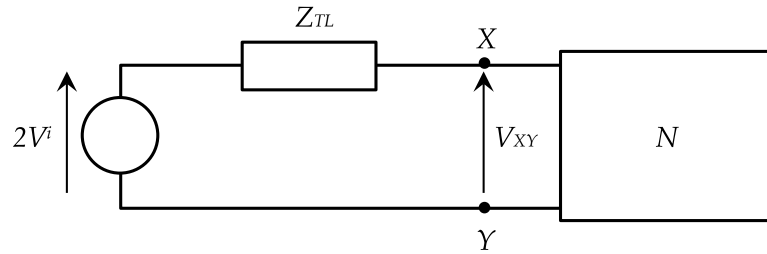


Figure 2.8: The Thévenin equivalent circuit for the link-line shown in Fig. (2.7). [2.6]

During the scattering procedure, reflected voltages can be calculated by assuming the Thévenin equivalent circuit model and using the expression shown previously in (2.26).

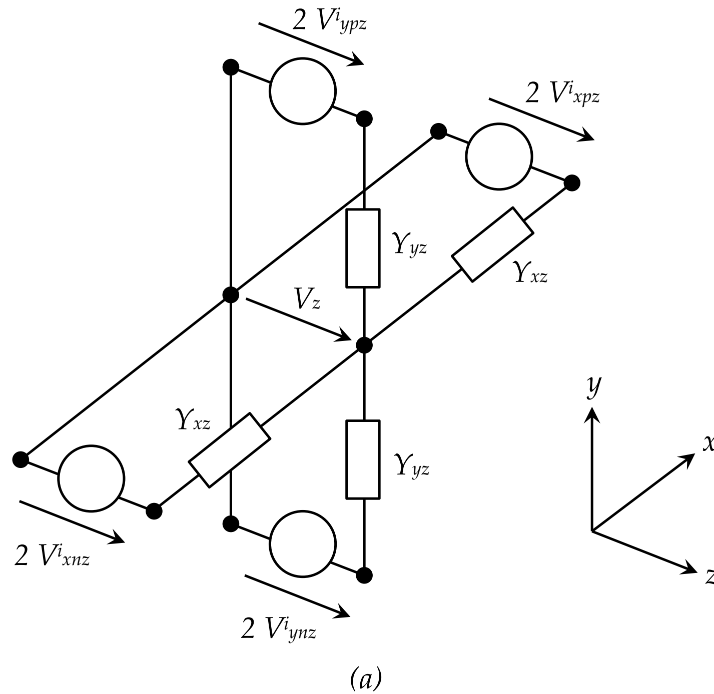


Figure 2.9: Thévenin equivalent circuit for the 2-D shunt node. [2.6]

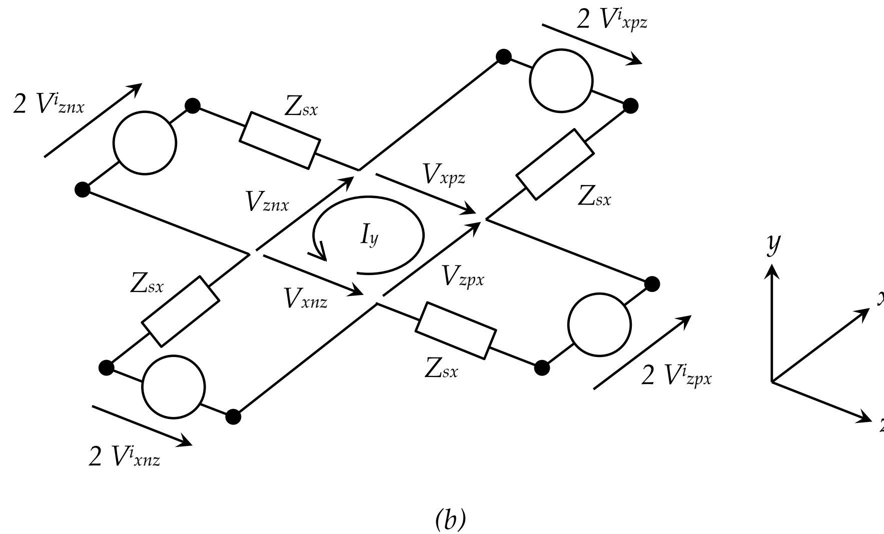


Figure 2.10: Thévenin equivalent circuit for the 2-D series node. [2.6]

2.3 Three-dimensional nodes

Following the early developments of the TLM method (which concentrated on 2-D problem spaces), efforts turned to a suitable 3-D TLM implementation. An expanded node was proposed [2.8] [2.9], with its disadvantage being that the field components were spatially separated, despite being trivial to calculate. Development of a condensed node followed [2.10], moving all scattering and field component values to one and the same point in space. The disadvantage of the condensed node was that depending on the direction of observation, the first connection of each node is either shunt or series. This presents small discrepancies in calculated values, which for example become apparent when observing boundary behaviour at high frequencies. In a significant step forwards, Johns proposed a symmetrical condensed node (SCN) [2.11], which eliminated the asymmetric nature of the original condensed node, optimised the number of arithmetic operations involved, and retained the single spatial point as the node centre. Since its development, the SCN is the most commonly implemented node for 3-D TLM simulations, and both it and its variations are used exclusively throughout this thesis for the 3-D desktop and cluster simulations. This section describes two different forms of SCNs, and gives a concise derivation of their scattering matrices.

2.3.1 The SCN

Previously, TLM node behaviour such as the node scatter matrix has been derived by constructing an equivalent Thévenin circuit. This methodology is not possible with the SCN, since its structure can no longer be represented in such a way. Fig. (2.11) shows the structure of the basic SCN without stubs[1]. In each direction of propagation into and out of the node, there are a pair of link-lines which carry the two relevant polarisations of any waves present in the mesh. Each of the 12 link-lines have the same characteristic impedance, which is equal to free-space impedance Z_0. This node structure allows the modelling of homogeneous meshes, where altering other mesh properties (such as the dielectric properties of nodes) is unnecessary.

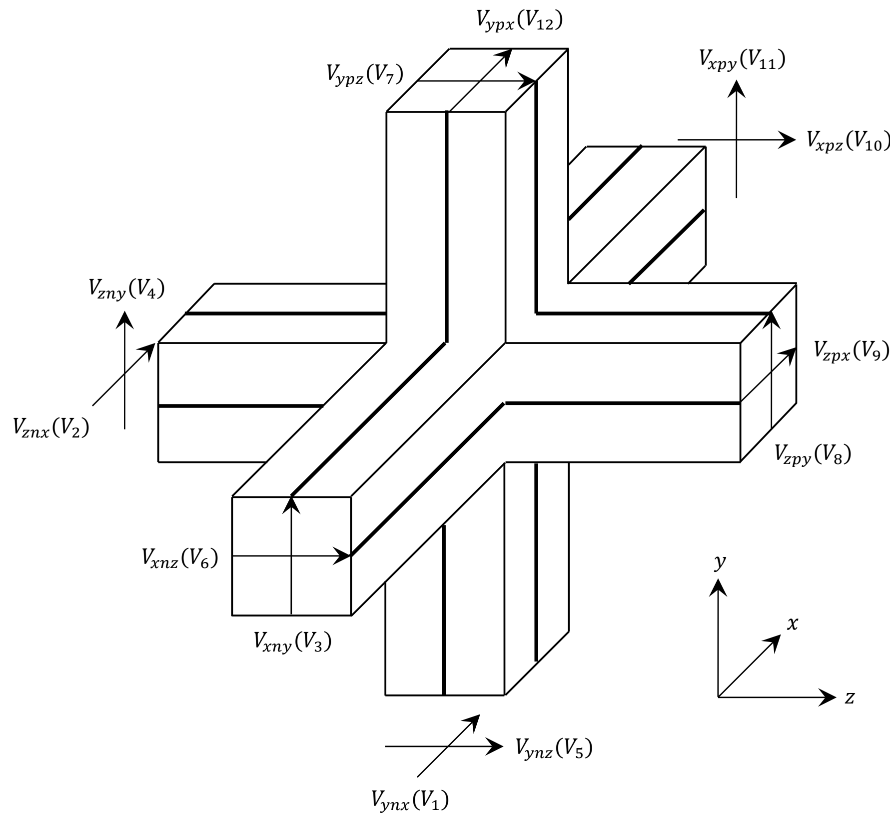


Figure 2.11: A non stub-loaded SCN, with Trenkić port notation (Johns' notation in parentheses).

The 12×12 scattering matrix of the SCN, **S** can be derived by first examining which port values are associated with the calculation of each field component. As an example,

[1]The use of stubs allows the modelling of materials with differing dielectric properties within the same mesh volume

a voltage pulse $V^i = 1$ is incident upon port V_{ynx} of the SCN. This pulse has field components E_x and H_z associated with it. One of the corresponding Maxwell equations involving these two field components is shown in (2.28).

$$\frac{\partial H_z}{\partial y} - \frac{\partial H_y}{\partial z} = \epsilon \frac{\partial E_x}{\partial t} \tag{2.28}$$

The equation in (2.28) shows that the incident pulse on V_{ynx} scatters into ports V_{ynx}, V_{znx}, V_{zpx} and V_{ypx}, as all ports share x-axis polarised components. Due to the symmetry of the SCN, the amplitude of the pulses scattered to ports V_{ynx} and V_{ypx} shall both be equal, and set to a and c respectively. Similarly, the pulses scattered to V_{znx} and V_{zpx} shall also be equal, and set to b. The second Maxwell's equation that concerns field components E_x and H_z is shown in (2.29).

$$\frac{\partial E_y}{\partial x} - \frac{\partial E_x}{\partial y} = -\mu \frac{\partial H_z}{\partial t} \tag{2.29}$$

This accounts for the remaining pulses that should be scattered into ports V_{xny} and V_{xpy}, which both have x-axis directed components. These quantities will be equal and opposite in sign, and shall be assigned the value d and $-d$ respectively. The same procedure can be applied to each port in turn, resulting in the scattering matrix, as shown in (2.30) [2.11].

$$\mathbf{S} = \begin{bmatrix} a & b & d & 0 & 0 & 0 & 0 & 0 & b & 0 & -d & c \\ b & a & 0 & 0 & 0 & d & 0 & 0 & c & -d & 0 & b \\ d & 0 & a & b & 0 & 0 & 0 & b & 0 & 0 & c & -d \\ 0 & 0 & b & a & d & 0 & -d & c & 0 & -0 & b & 0 \\ 0 & 0 & 0 & d & a & b & c & -d & 0 & b & 0 & 0 \\ 0 & d & 0 & 0 & b & a & b & 0 & -d & c & 0 & 0 \\ 0 & 0 & 0 & -d & c & b & a & d & 0 & b & 0 & 0 \\ 0 & 0 & b & c & -d & 0 & d & a & 0 & 0 & b & 0 \\ b & c & 0 & 0 & 0 & -d & 0 & 0 & a & d & 0 & b \\ 0 & -d & 0 & 0 & b & c & b & 0 & d & a & 0 & 0 \\ -d & 0 & c & b & 0 & 0 & 0 & b & 0 & 0 & a & d \\ c & b & -d & 0 & 0 & 0 & 0 & 0 & b & 0 & d & a \end{bmatrix} \tag{2.30}$$

Scattering properties

In order to derive the true scattering matrix, the values of a, b, c and d must firstly be found. The total capacitance associated with the link-lines governing ports V_{ynx}, V_{znx}, V_{zpx} and V_{ypx}, C_x is defined as shown in (2.31).

$$C_x = \epsilon \frac{wv}{u} \tag{2.31}$$

The field components associated with these link-lines are defined as shown in (2.32).

$$E_x = \frac{V_x}{\Delta x}$$

$$H_y = \frac{I_z}{\Delta y} \tag{2.32}$$

$$H_z = -\frac{I_y}{\Delta z}$$

Where V_x is the voltage drop across the link-lines in the x-axis direction, I_y and I_z are

circulating currents in the y and z-axis directions respectively and Δx, Δy and Δz denote the x, y, z dimensions of a given node. Let I_{ynx}, I_{znx}, I_{zpx} and I_{ypx} be the currents entering their respective ports. The discrete form of (2.28) is then shown as in (2.33), which can be simplified to (2.34).

$$\frac{I_{ypx} + I_{ynx}}{wv} + \frac{I_{zpx} + I_{znx}}{wv} = \frac{C_x u}{wv} \frac{\partial}{\partial t} \frac{V_x}{u} \tag{2.33}$$

$$I_{ypx} + I_{ynx} + I_{zpx} + I_{znx} = C_x \frac{\partial V_x}{\partial t} \tag{2.34}$$

This demonstrates the fact that there will be no loss of current within the node, and that the only current loss in the system will be due to the rate of change of voltage across the capacitance of the link-lines outside the node centre. The scattering matrix is expressed in terms of voltages present on link-lines. The conservation of current at the node centre can be expressed in terms of port link-line voltages, as shown in (2.35).

$$1 - a = 2b + c \tag{2.35}$$

The z-axis directed inductance, L_z is associated with ports V_{ynx}, V_{xny}, V_{xpy} and V_{ypx}. L_z is defined as shown in (2.36), with the associated field components shown in (2.37).

$$L_z = \mu \frac{uv}{w} \tag{2.36}$$

$$E_x = \frac{V_x}{u}$$

$$E_y = \frac{V_y}{v} \tag{2.37}$$

$$H_z = -\frac{I_z}{w}$$

The discrete form of the Maxwell equation (2.29), governing the relationship between

z-axis directed inductance and the relevant port voltages is given as (2.38), which can be simplified to (2.39).

$$\frac{V_{xny} - V_{xpy}}{uv} - \frac{V_{ypx} - V_{ynx}}{uv} = -L_z \frac{w}{uv} \frac{\partial}{\partial t} \frac{I_A}{w} \tag{2.38}$$

$$V_{xny} + V_{ypx} - V_{xpy} - V_{ynx} = L_z \frac{\partial I_A}{\partial t} \tag{2.39}$$

This demonstrates the fact that there will be no loss of voltage within the node, and that the only voltage loss in the system will be due to the rate of change of current in the inductance of the link-lines outside the node centre. This results in the port voltage equation (2.40).

$$1 + a = 2d + c \tag{2.40}$$

A SCN must conserve energy in any incident pulse configuration present within the mesh. This shows that the scattering matrix, $\mathbf{S}$ must be a unitary matrix satisfying $\mathbf{S^T S} = \mathbf{I}$. This gives rise to the fact that $\Sigma S_{nr} S_{rs} = 1$ where $r = s$, and that $\Sigma S_{nr} S_{rs} = 0$ where $r \neq s$. These properties of the scattering matrix result in the following equations relating the port voltage quantities $a - d$, as shown in (2.41).

$$\begin{gathered} a^2 + 2b^2 + c^2 + 2d^2 = 1 \\ 2ab + 2bc = 0 \\ 2ad - 2cd = 0 \\ 2ac + 2b^2 - 2d^2 = 0 \end{gathered} \tag{2.41}$$

Solving the simultaneous equations (2.35), (2.40) and those included in (2.41) results in

the following result: $a = 0, b = \frac{1}{2}, c = 0$ and $d = \frac{1}{2}$. The scattering matrix, **S** can then be redefined as shown in (2.42), appearing in a similar format to the scattering matrices constructed for both 2-D node schemes.

$$\mathbf{S} = \frac{1}{2}\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & -1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & -1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \tag{2.42}$$

2.3.2 The stub-loaded SCN

In order to model lossy or inhomogeneous materials (for example modelling a material with a dielectric constant different to free-space), it is necessary to construct a version of the SCN which can accommodate these features. Johns proposed the addition of 6 additional stub ports to the 12-port SCN. As before, ports 1-12 are the same and interface with the corresponding ports in neighbouring nodes of the mesh. The stub ports 13-18 couple with the field components E_x, E_y, E_z, H_x, H_y and H_z. The E-field ports are open-circuit stubs that allow for additional node capacitance, while the H-field ports are short-circuit stubs adding additional inductance to the node. As before, the propagation time from the outer edge of each port to the centre of the node is defined as $\frac{\Delta t}{2}$, where Δt is the time-step of the mesh.

Open-circuit stub parameters

The capacitance value of the region of space defined by the stub-loaded SCN in the x-axis direction, C_x is defined in terms of a capacitance present on the corresponding x-directed link-lines using an open-circuit stub value of C_x^s. The definition of C_x is shown in (2.43). We can rearrange this equation, noting the relationship between the impedance of free-space, Z_0 and ϵ_0, as shown in (2.44). This results in a definition of the stub value C_x^s, as shown in (2.45).

$$C_x = \epsilon\Delta\ell = 2Y_0\Delta t + C_x^s \tag{2.43}$$

$$Z_0 = \frac{|E|}{|H|} = \mu_0 c = \sqrt{\frac{\mu_0}{\epsilon_0}} = \frac{1}{\epsilon_0 c} \Rightarrow \epsilon_0 = \frac{Y_0}{c} \tag{2.44}$$

$$C_x^s = Y_0\left(\frac{\epsilon_r}{c}\Delta\ell - 2\Delta t\right) \tag{2.45}$$

The time for an impulse to be scattered into, and then back out of a stub, is set to be the same as the time-step of the mesh Δt, and the stub length is therefore $\frac{1}{2}\Delta\ell$. The stub admittance value can be calculated as shown in (2.46), normalised to the admittance of free-space, Y_0.

$$\hat{Y}_x^s = \frac{2C_x^s}{Y_0\Delta t} = 2\left(\frac{\epsilon_r}{c}\frac{\Delta\ell}{\Delta t} - 2\right) \tag{2.46}$$

The stub value is modelled as a passive component within the SCN. Therefore, to ensure the stability of the simulation, the stub must always have a positive value. This is governed by ensuring the time-step Δt is calculated based on the lowest value of ϵ_r of any node in the mesh. Equation (2.46) can be simplified further by noting that $\Delta t = \frac{\Delta\ell}{2c}$, and is shown in (2.47).

$$\hat{Y}_x^s = \begin{cases} 4(\epsilon_r - 1) & \text{if } \epsilon_0 = 1, \\ 4(\frac{\epsilon_r}{\epsilon_0} - 1) & \text{otherwise.} \end{cases} \tag{2.47}$$

The same procedure is applied for calculating the y and z-axis directed stub values, $\hat{Y}_y^s$ and $\hat{Y}_z^s$.

Short-circuit stub parameters

The inductance value of the region of space defined by the stub-loaded SCN in the x-axis direction, L_x is defined in terms of an inductance present on the corresponding x-directed link-lines using a short-circuit stub value of L_x^s, which is derived as shown in (2.48).

$$\begin{aligned} L_x &= \mu\Delta\ell = 2Z_0\Delta t + L_x^s \\ \Rightarrow L_x^s &= Z_0\left(\frac{\mu_r}{c}\Delta\ell - 2\Delta t\right) \end{aligned} \tag{2.48}$$

The stub impedance value, normalised to the free-space impedance Z_0 for a x-axis directed stub, is derived as shown in (2.49).

$$\begin{aligned} \hat{Z}_x^s &= \frac{2L_x^s}{Z_0\Delta t} \\ &= \begin{cases} 4(\mu_r - 1) & \text{if } \mu_0 = 1, \\ 4(\frac{\mu_r}{\mu_0} - 1) & \text{otherwise.} \end{cases} \end{aligned} \tag{2.49}$$

It should be noted that since this stub is short-circuited, the reflected pulse observed after time Δt will be opposite in sign.

Modelling inhomogeneous materials

In order to model materials of varying electrical and magnetic properties, as well as anisotropic materials, it is necessary to first generalise our mesh node definition to take account of non-cubic mesh nodes. An example of an anisotropic, non-cubic block of

space is shown in Fig. (2.12).

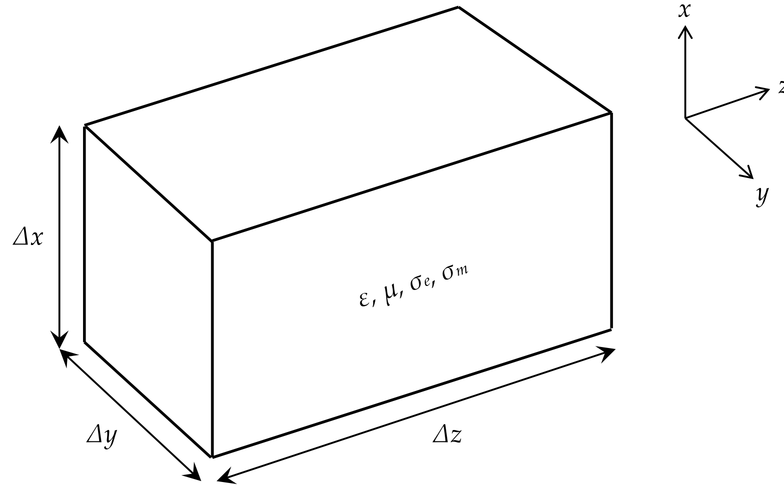


Figure 2.12: An arbitrary, non-cubic block of space of dimensions Δx, Δy, Δz. [2.6]

Using this generalised node definition, it is possible to determine expressions for the capacitance, inductance, electric conductivity and magnetic conductivity. These direction-dependent expressions are as follows:

$$C_i = \epsilon_i \frac{\Delta j \Delta k}{\Delta i} \tag{2.50}$$

$$L_i = \mu_i \frac{\Delta j \Delta k}{\Delta i} \tag{2.51}$$

$$G_i = \sigma_{ei} \frac{\Delta j \Delta k}{\Delta i} \tag{2.52}$$

$$R_i = \sigma_{mi} \frac{\Delta j \Delta k}{\Delta i} \tag{2.53}$$

Where $i,\ j,\ k$ are interchangeably equivalent to x, y, and z directions. C_i, L_i, G_i and R_i are the respective total node quantities in the i direction. Now that a set of generalised expressions have been presented, it is possible to consider the case of a mesh of cubic nodes. In this case, $\Delta x = \Delta y = \Delta z = \Delta \ell$ and therefore the above expressions can be simplified as follows:

$$C_i = \epsilon_i \Delta \ell \tag{2.54}$$

$$L_i = \mu_i \Delta \ell \tag{2.55}$$

$$G_i = \sigma_{ei}\Delta\ell \tag{2.56}$$

$$R_i = \sigma_{mi}\Delta\ell \tag{2.57}$$

Electric losses can be modelled with no additional storage at the individual node level. This is due to the fact that as an open-circuit stub, any pulses incident on the electric loss stubs during the scatter process will be absorbed, with no reflected pulses being emitted. The loss tangent for a material of complex permittivity ϵ and conduction conductivity σ, is given in (2.58).

$$tan\delta_e = \frac{\sigma_e}{\omega\epsilon_r\epsilon_0} \tag{2.58}$$

The stub admittance, normalised with respect to free-space impedance Z_0 for a i-axis directed stub, is derived as shown in (2.59).

$$\hat{G}_i = Z_0 G_i \tag{2.59}$$

Magnetic losses are modelled in a similar fashion; however the energy removed from the node is proportional to the magnetic field instead of the electric field. The stub impedance, normalised to the admittance of free-space Y_0 is shown in (2.60). Another method for modelling magnetic losses, that more accurately approximates the behaviour of real ferrite materials, has previously been described [2.12].

$$\hat{R}_i = Y_0 R_i \tag{2.60}$$

Scattering Properties

The scattering matrix, **S** is derived in a similar fashion to the non stub-loaded SCN described previously. With consideration to the x-axis directed component of the electric field, the relationship between incident and reflected pulses across the relevant link-lines is shown in (2.61). V_{ox} and V_{ex} denote the open-circuit and electric loss stubs, respectively.

$$V^i_{ynx} + V^i_{ypx} + V^i_{znx} + V^i_{zpx} + \hat{Y}^s_x V^i_{ox} = V^r_{ynx} + V^r_{ypx} + V^r_{znx} + V^r_{zpx} + \hat{Y}^s_x V^r_{ox} + \hat{G}_x V^r_{ex} \quad (2.61)$$

With consideration to the x-axis directed component of the magnetic field, the relationship between incident and reflected flux across the relevant link-lines is shown in (2.62). V_{sx} and V_{mx} denote the short-circuit and magnetic loss stubs, respectively.

$$\begin{aligned} I^i_{ynz} + I^i_{zpy} - I^i_{ypz} - I^i_{zny} + \hat{Z}^s_x I^i_{sx} &= I^r_{ynz} + I^r_{zpy} - I^r_{ypz} - I^r_{zny} + \hat{Z}^s_x I^r_{sx} + \hat{R}_x I^r_{mx} \\ \Rightarrow V^i_{ynz} + V^i_{zpy} - V^i_{ypz} - V^i_{zny} + V^i_{sx} &= -(V^i_{ynz} + V^i_{zpy} - V^i_{ypz} - V^i_{zny} + V^i_{sx} + V^r_{mx}) \end{aligned} \quad (2.62)$$

The electric and magnetic field continuity is ensured in an identical fashion as described for the non stub-loaded SCN. In the stub-loaded case, the stubs themselves do not form part of the calculations, as they do not have an associated direction.

For the electric field stubs, the derivation of the pulses reflected out of the open-circuit and electric loss stubs is shown in (2.63) and (2.64).

$$V^r_{ox} = V_x - V^i_{ox} \quad (2.63)$$

$$V^r_{ex} = V_x \quad (2.64)$$

Note here that V_x is the total voltage across the node. Therefore, V^r_{ex} is a redundant quantity, and does not require calculation unless its magnitude is necessary. V_x can be defined as the voltage across the total node capacitance, C_x. This gives rise to the definition, as shown in (2.65).

$$V_x = \frac{V_{ynx} + V_{ypx} + V_{znx} + V_{zpx} + \hat{Y}^s_x V_{ox}}{4 + \hat{Y}^s_x} \quad (2.65)$$

V_x can also be re-defined in terms of the relevant incident pulses on the node. This

is convenient, since they are known quantities during the scatter and connect process. This re-definition is shown in (2.66).

$$V_x = \frac{2}{4 + \hat{Y}_x^s + \hat{G}_x}(V_{ynx}^i + V_{ypx}^i + V_{znx}^i + V_{zpx}^i + \hat{Y}_x^s V_{ox}^i) \tag{2.66}$$

For the magnetic field stubs, the derivation of the pulses reflected out of the short-circuit and magnetic loss stubs is shown in (2.67) and (2.68).

$$V_{sx}^r = Z_0 \hat{Z}_x^s I_x + V^i sx \tag{2.67}$$

$$V_{mx}^r = Z_0 \hat{R}_x I_x \tag{2.68}$$

I_x is the total current circulating around the node, the sign of which is set to be consistent with that of the magnetic field. I_x can be defined as shown in (2.69), relating the flux linkage on the total node inductance, L_x with the sum of the flux linkage on the relevant link-lines.

$$I_x = \frac{I_{ypz} + I_{zny} - I_{ynz} - I_{zpy} - \hat{Z}_x^s I_{sx}}{4 + \hat{Z}_x^s} \tag{2.69}$$

I_x may also be re-defined in terms of the relevant incident pulses on the node. As before, this is convenient since these are known quantities. This re-definition is shown in (2.70).

$$I_x = \frac{2}{Z_0(4 + \hat{Z}_x^s + \hat{R}_x)}(V_{ypz}^i + V_{zny}^i - V_{ynz}^i - V_{zpy}^i - V_{sx}^i) \tag{2.70}$$

2.4 Boundary conditions

Boundary conditions are a fundamental component of any numerical modelling technique. They allow for various different materials to be modelled within the mesh, as well as ensuring the mesh is terminated at its edge extents. This is often necessary

since a simulation always represents a finite volume of space, which is often used to represent a problem in an infinite volume. This section discusses the various boundary conditions utilised throughout this thesis.

2.4.1 Reflective boundaries

A reflective boundary, or one that is a perfect electric conductor (PEC), reflects incident electromagnetic waves perfectly with no loss and a phase inversion. PEC boundaries can be used to represent conducting metal objects within the mesh. This is clearly a useful component to any numerical modelling technique, as it can be used to model antennas, cavities and other metallic structures of interest for RF engineers.

From a conceptual viewpoint, the basic requirement for a boundary node of this type is that it must return any pulses incident on it back into the mesh in the exact opposite direction. As it is also a *perfect* reflective boundary, these reflected pulses will be of the same magnitude as the original incident pulses, without any loss factor. This is electrically equivalent to a short-circuit, and as such the reflection coefficient for the waves reflected from a PEC boundary is -1.0. An example of the comparison between the scatter equation for a port in a normal non stub-loaded SCN, and a PEC boundary node, is shown in (2.71).

$$\begin{aligned} V^r_{ynx} &= \frac{1}{2}\left(V^i_{znx} + V^i_{zpx} + V^i_{xny} - V^i_{xpy}\right) \\ V^r_{ynx} &= -V^i_{ynx} \end{aligned} \tag{2.71}$$

2.4.2 Absorbing boundaries

Implementations of absorbing boundaries are commonplace within TLM software applications, and indeed most numerical methods. This is due to the fact that they allow the approximation of an infinite space outside of the model boundaries, which is useful in many cases. A popular approach, named *matched termination*, is based on the assumption of connecting an infinitely long transmission line to each node of a boundary face of the mesh. In practice this is impossible; however within the model the electrical

properties of these terminating lines are selected so as to eliminate reflections back into the mesh, as much as possible. The result is a face of nodes that will largely absorb any energy observed at their location. In order to derive an expression for the reflection coefficient of a MTBC, the characteristic impedance of the link-lines is first expressed in terms of quantities determined previously in Section (2.3.2). For example, the characteristic impedance of a z-directed line in x-polarisation can be expressed as shown in (2.72).

$$Z_{znx} = \sqrt{\frac{L_y}{C_x}} = \sqrt{\frac{\mu_{ry}\Delta x\Delta z}{\Delta y}\frac{\Delta x}{\epsilon_{rx}\Delta z\Delta y}} = Z_0\sqrt{\frac{\mu_{ry}}{\epsilon_{rx}}}\frac{\Delta x}{\Delta y} \tag{2.72}$$

As implied by the name, the terminating resistance for a node that is to possess a MTBC should be matched to that of the link-line it is connected to, as far as possible. Therefore, as with the previous example for calculating the characteristic impedance, the terminating resistance for the link-line is as shown in (2.73).

$$R_{MTBC} = Z_0\sqrt{\frac{\mu_{ry}}{\epsilon_{rx}}}\frac{\Delta x}{\Delta y} \tag{2.73}$$

This expression can be generalised using the same methods as when generalising node quantities in Section (2.3.2). This gives a definition for the termination resistance on an i-oriented link-line, as shown in (2.74).

$$R_{MTBC} = Z_0\sqrt{\frac{\mu_{rk}}{\epsilon_{rj}}}\frac{\Delta j}{\Delta k} \tag{2.74}$$

The reflection coefficient for the MTBC to be applied during the TLM simulation can therefore be expressed as shown in (2.75).

$$\Gamma_{ij} = \frac{R_{MTBC} - Z_{ij}}{R_{MTBC} + Z_{ij}} \tag{2.75}$$

It should be noted, that in the case of all 3-D SCN nodes that share a link-line impedance equal to Z_0, the above expression will result in a value of *0* for the MTBC reflection coefficient.

2.4.3 Periodic boundaries

The periodic boundary condition is a special case, where the pulses incident on the node are not modified *per se* by a coefficient (as shown previously), but are transmitted to be incident on non-adjacent nodes within the mesh volume. In the typical implementation, periodic boundaries are placed on the edge extents of a mesh to allow periodic, symmetrical modelling of the geometry within a mesh. In this arrangement, pulses arriving at a boundary node will be transmitted to become incident on the equivalent node at the opposite face of the mesh. By placing wrap-around boundaries on all external faces of the mesh, the effect will be that any wave pulses will effectively observe a periodic array of the objects present within the mesh. This is shown in Fig. (2.13).

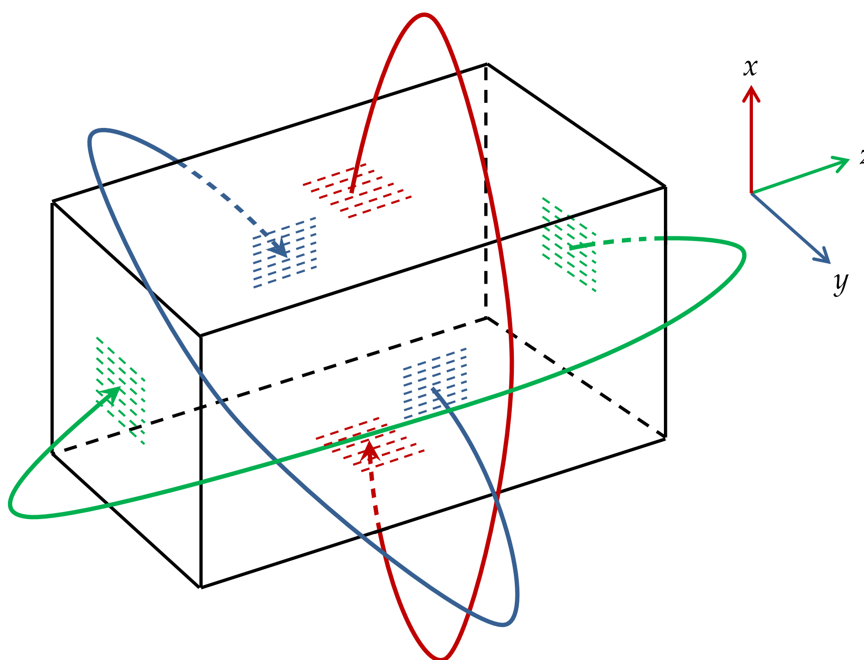


Figure 2.13: An illustration of periodic boundaries in operation on all six faces of a TLM mesh volume.

Periodic boundaries arranged as in the figure above, are useful for the modelling of periodic and quasi-periodic structures, without resorting to increasing the overall mesh volume. This ensures that simulation times are kept to a minimum, whilst analysing periodic effects.

2.5 Optimisations to the 3-D TLM algorithm

As discussed previously in this chapter, both the SCN and stub-loaded SCN have a scattering matrix as the foundation of the TLM algorithm for distributing wave-front pulse components throughout the mesh for the subsequent time step. Whilst this is useful as part of a convenient representation of the scatter and connect process, it is not optimised to reduce computation time when implementing TLM as part of a software application. Before considering strategies for implementing a parallelised version of the TLM algorithm, it is pertinent to ensure that the algorithm itself is optimised to reduce computational operations. It has been shown previously [2.13] that it is possible to significantly reduce computations with respect to the scattering matrix proposed by Johns [2.11], as well as other alternative methods [2.14]. This section discusses these differences in depth, and presents expressions for computational time per node in each case.

2.5.1 SCN computational optimisations

A first approach to implementing the 3-D TLM algorithm within a software application would be to replicate the original scattering matrix, **S** that is presented in (2.3.1). The scattering matrix can be re-expressed using Herring node port notation, in the form shown in (2.76).

$$
\begin{aligned}
V^r_{ynx} &= \frac{1}{2}\left(V^i_{znx} + V^i_{zpx} + V^i_{xny} - V^i_{xpy}\right) \\
V^r_{ypx} &= \frac{1}{2}\left(V^i_{znx} + V^i_{zpx} + V^i_{xpy} - V^i_{xny}\right) \\
V^r_{znx} &= \frac{1}{2}\left(V^i_{ynx} + V^i_{ypx} + V^i_{xnz} - V^i_{xpz}\right) \\
V^r_{zpx} &= \frac{1}{2}\left(V^i_{ynx} + V^i_{ypx} + V^i_{xpz} - V^i_{xnz}\right) \\
V^r_{zny} &= \frac{1}{2}\left(V^i_{xny} + V^i_{xpy} + V^i_{ynz} - V^i_{ypz}\right) \\
V^r_{zpy} &= \frac{1}{2}\left(V^i_{xny} + V^i_{xpy} + V^i_{ypz} - V^i_{ynz}\right) \\
V^r_{xny} &= \frac{1}{2}\left(V^i_{zny} + V^i_{zpy} + V^i_{ynx} - V^i_{ypx}\right) \\
V^r_{xpy} &= \frac{1}{2}\left(V^i_{zny} + V^i_{zpy} + V^i_{ypx} - V^i_{ynx}\right) \\
V^r_{xnz} &= \frac{1}{2}\left(V^i_{ynz} + V^i_{ypz} + V^i_{znx} - V^i_{zpx}\right) \\
V^r_{xpz} &= \frac{1}{2}\left(V^i_{ynz} + V^i_{ypz} + V^i_{zpx} - V^i_{znx}\right) \\
V^r_{ynz} &= \frac{1}{2}\left(V^i_{xnz} + V^i_{xpz} + V^i_{zny} - V^i_{zpy}\right) \\
V^r_{ypz} &= \frac{1}{2}\left(V^i_{xnz} + V^i_{xpz} + V^i_{zpy} - V^i_{zny}\right)
\end{aligned}
\tag{2.76}
$$

The total mathematical operations for this implementation are 36 additions/subtractions and 12 multiplications. By examining the group of equations presented above, it becomes apparent that portions of each equation share calculation components when pulses that arrive at the node share a common direction and polarisation (e.g. the equations for V^r_{ynx} and V^r_{ypx}). When implementing this as a piece of software, it is sensible to cache these partial sums and differences in temporary variables, an example implementation is shown below (in pseudo-code):

```
float vSum, vDiff;
//Scatter (using cached partial sum and differences for each port)
vSum = Vznx + Vzpx;
vDiff = Vxny - Vxpy;
Vynx = 0.5 * (vSum + vDiff);
Vypx = 0.5 * (vSum - vDiff);
vSum = Vynx + Vypx;
vDiff = Vxnz - Vxpz;
Vznx = 0.5 * (vSum + vDiff);
Vzpx = 0.5 * (vSum - vDiff);
```

```
vSum = Vxny + Vxpy;
vDiff = Vynz - Vypz;
Vzny = 0.5 * (vSum + vDiff);
Vzpy = 0.5 * (vSum - vDiff);
vSum = Vzny + Vzpy;
vDiff = Vynx - Vypx;
Vxny = 0.5 * (vSum + vDiff);
Vxpy = 0.5 * (vSum - vDiff);
vSum = Vynz + Vypz;
vDiff = Vznx - Vzpx;
Vxnz = 0.5 * (vSum + vDiff);
Vxpz = 0.5 * (vSum - vDiff);
vSum = Vxnz + Vxpz;
vDiff = Vzny - Vzpy;
Vynz = 0.5 * (vSum + vDiff);
Vypz = 0.5 * (vSum - vDiff);
```

This optimisation reduces the amount of additions/subtractions from 36 to 24 per node, and the multiplications remain constant at 12 per node. The algorithm can be optimised further by utilising the identity shown in (2.77), to express the second port of each direction/polarisation pair in terms of the first. This reduces the multiplications from 12 to 6 per node, as follows:

```
float vDiff;
//Scatter (using cached partial differences for each port and equivalence
    identity)
vDiff = Vxny - Vxpy;
Vynx = 0.5 * (Vznx + Vzpx + vDiff);
Vypx = Vynx - vDiff;
vDiff = Vxnz - Vxpz;
Vznx = 0.5 * (Vynx + Vypx + vDiff);
Vzpx = Vznx - vDiff;
vDiff = Vynz - Vypz;
Vzny = 0.5 * (Vxny + Vxpy + vDiff);
Vzpy = Vzny - vDiff;
vDiff = Vynx - Vypx;
Vxny = 0.5 * (Vzny + Vzpy + vDiff);
Vxpy = Vxny - vDiff;
vDiff = Vznx - Vzpx;
```

```
Vxnz = 0.5 * (Vynz + Vypz + vDiff);
Vxpz = Vxnz - vDiff;
vDiff = Vzny - Vzpy;
Vynz = 0.5 * (Vxnz + Vxpz + vDiff);
Vypz = Vynz - vDiff;
```

$$\frac{1}{2}(a-b) = \frac{1}{2}(a+b) - b \qquad (2.77)$$

The optimisation methods that have been discussed are summarised in Table (2.1). It is possible to use these figures to construct expressions that allow developers to produce estimates of the amount of real-world time that the scatter process will take to operate on a single node. Using the Trenkić method as an example, we assign the computational time for a single floating point addition, subtraction and multiplication the values ℓ, m and n respectively. It is then trivial to construct an expression representing the total time to operate on a single node, as shown in (2.78), assuming constant time complexity on read and write operations to variables in memory.

Component	**Johns [2.11]**	**Naylor [2.14]**	**Trenkić [2.13]**
Add/Sub	36	42	24
Multiply	12	6	6
Total	48	48	30

Table 2.1: Operations per node for different optimisation techniques of the scatter process within TLM. [2.13]

$$t_{sc} = 12\ell + 12m + 6n \qquad (2.78)$$

2.5.2 Stub-loaded SCN computational optimisations

As discussed earlier in this chapter, in order to model lossy materials of different dieletric properties, it is necessary to use stub-load SCNs. The scattering method discussed earlier presented a 18×18 scattering matrix, **S**. Determining this matrix is a complex task, requiring the solution of non-linear simultaneous equations. As previously discussed, Naylor and Ait-Sadi [2.14] have demonstrated that it is possible to execute the scattering procedure *without* the need to first determine a scattering

matrix. Instead, the scattering process is reduced to three equations for node voltages, and three equations for node loop currents; these are summarised in (2.79).

$$\begin{aligned}
V_x &= kV_x\left(V_{ynx} + V_{ypx} + V_{znx} + V_{zpx} + V_{ox}\right)\\
V_y &= kV_y\left(V_{zny} + V_{zpy} + V_{xny} + V_{xpy} + V_{oy}\right)\\
V_z &= kV_z\left(V_{xnz} + V_{xpz} + V_{ynz} + V_{ynz} + V_{oz}\right)\\
I_x &= kI_x\left(V_{ynz} - V_{ypz} + V_{zpy} - V_{zny} + V_{sx}\right)\\
I_y &= kI_y\left(V_{znx} - V_{zpx} + V_{xpz} - V_{xnz} + V_{sy}\right)\\
I_z &= kI_z\left(V_{xny} - V_{xpy} + V_{ypx} - V_{ynx} + V_{sz}\right)
\end{aligned} \tag{2.79}$$

The constants $kV_x - kV_z$ and $kI_x - kI_z$ are calculated once, and stored for future reference at the start of the simulation. An example of this is demonstrated as follows (in pseudo-code), where G is the normalised electric loss stub admittance and Z is the normalised magnetic loss stub impedance:

```
kVx = 2.0 / (4.0 + Yox + Gx);
kVy = 2.0 / (4.0 + Yoy + Gy);
kVz = 2.0 / (4.0 + Yoz + Gz);
kIx = 2.0 / (4.0 + Ysx + Zx);
kIy = 2.0 / (4.0 + Ysy + Zy);
kIz = 2.0 / (4.0 + Ysz + Zz);
```

Using the equations in (2.79), the scatter process for the stub-loaded SCN is now significantly simpler than in the case of using the scattering matrix directly. Firstly, each non-stub port is scattered into, as in the non stub-loaded SCN case. An example of this is as follows (in pseudo-code):

```
//Scatter into non-stub ports of the SCN
Vxny = Vy - ZIz - Vxpy;
Vxnz = Vz + ZIy - Vxpz;
Vxpy = Vy + ZIz - Vxny;
Vxpz = Vz - ZIy - Vxnz;
Vynz = Vz - ZIx - Vypz;
Vynx = Vx + ZIz - Vypx;
Vypz = Vz + ZIx - Vynz;
```

```
Vypx = Vx - ZIz - Vynx;
Vznx = Vx - ZIy - Vzpx;
Vzny = Vy + ZIx - Vzpy;
Vzpx = Vx + ZIy - Vznx;
Vzpy = Vy - ZIx - Vzny;
```

Finally, the open and short-circuit stubs are updated at the end of the scatter process for each node. An example of this is as follows (in pseudo-code):

```
//Update the node stub values
VoxYox = (Yox * Vx) - VoxYox;
VoyYoy = (Yox * Vy) - VoyYoy;
VozYoz = (Yox * Vz) - VozYoz;
Vsx = (Zsx * ZIx) - Vsx;
Vsy = (Zsy * ZIy) - Vsy;
Vsz = (Zsz * ZIz) - Vsz;
```

It should be noted that for the open-circuit stub values, the product of the voltage and the normalised admittance is stored for convenience, and to avoid duplicate calculations during the program execution as this quantity is required more than once.

2.5.3 Computational optimisations summary

It has been demonstrated that there is significant potential for performance gains when using the TLM method without first adapting it for parallel computation. Previous investigations [2.13] have demonstrated that optimisations are indeed possible which significantly reduce the number of numerical operations present during the scattering process of the TLM algorithm. This is achieved by re-thinking the canonical derivation [2.11] of the process in terms of a software function, and identifying repeated partial sums and calculated quantities which are stored for access later.

These optimisations to the TLM method whilst the algorithm is still in a serially-processed format are critically important in order to achieve the optimum processing performance when a solver has been implemented using a fully-developed parallel computing strategy. The importance of considering the algorithm from a software developer's perspective has been identified, and allows additional performance gains from the beginning of the implementation of the solver application. A method for

easily quantifying these initial performance gains based on the number of mathematical operations present has also been presented.

2.6 Conclusion

This chapter has presented an overview of the TLM method; including a brief history, its analogies to Maxwell's equations and its limitations. The relationship between 3-D and 2-D node types has also been highlighted, and the 3-D SCN has been identified as the favoured node type for the project.

The main features of the method have been discussed, including derivations and implementation details for the modelling of excitation sources, boundaries and inhomogeneous materials.

Periodic boundaries were highlighted as having particular usefulness for the project by allowing for the modelling of larger quasi-periodic raindrop fields, without needing to expand the mesh volume.

Optimisations to the 3-D TLM algorithm to improve solver performance were discussed based on previous investigations. The importance of algorithm optimisation prior to considering parallel processing was highlighted.

References

[2.1] P. B. Johns and R. L. Beurle, "Numerical solution of 2-dimensional scattering problems using a transmission-line matrix," *Proc. of the Inst. of Electrical Engineers*, vol. 118, no. 9, pp. 1203–1208, 1971.

[2.2] G. Kron, "Equivalent circuit of the field equations of Maxwell-I," *Proc. of the IRE*, vol. 32, no. 5, pp. 289–299, 1944.

[2.3] S. Akhtarzad and P. B. Johns, "Generalised elements for t.l.m. method of numerical analysis," *Proc. of the Inst. of Electrical Engineers*, vol. 122, no. 12, pp. 1349–1352, 1975.

[2.4] C. Huygens, *Traité de la Lumière*. Leiden, 1690.

[2.5] C. Christopoulos, *The transmission-line modelling method: TLM*. IEEE/OUP Series on Electromagnetic Wave Theory, New York, USA; Oxford, UK: IEEE Press; Oxford University Press, 1995.

[2.6] J. A. Flint, *Efficient automotive electromagnetic modelling*. PhD thesis, Loughborough University, 2000.

[2.7] C. Christopoulos, *The transmission-line modelling method: TLM*, ch. 5.3, p. 101. IEEE/OUP Series on Electromagnetic Wave Theory, New York, USA; Oxford, UK: IEEE Press; Oxford University Press, 1995.

[2.8] S. Akhtarzad and P. B. Johns, "Solution of 6-component electromagnetic fields in three space dimensions and time by the t.l.m. method," *Electronic Letters*, vol. 10, no. 25/26, pp. 535–537, 1974.

[2.9] S. Akhtarzad and P. B. Johns, "Solution of maxwell's equations in three space dimensional and time by the TLM method of numerical analysis," *Proc. of the Inst. of Electrical Engineers*, vol. 122, pp. 1344–1348, 1975.

[2.10] P. Saguet and E. Pic, "Utilization d'un nouveau type de noeud dans la méthode TLM en trois dimensions," *Electronic Letters*, vol. 18, pp. 478–480, 1982.

[2.11] P. B. Johns, "A symmetrical condensed node for the TLM method," *IEEE Trans. on Microwave Theory and Techniques*, vol. 35, no. 4, pp. 370–377, 1987.

[2.12] J. F. Dawson, "Improved magnetic loss for TLM," *Electronic Letters*, vol. 29, no. 5, pp. 467–468, 1993.

[2.13] V. Trenkić, C. Christopoulos, and T. M. Benson, "New developments in the numerical simulation of RF and microwave circuits using the TLM method," (Nis), 1st Conf. in Modern Satellite and Cable Systems (TELSIKS), 7-9 October 1993.

[2.14] P. Naylor and R. Ait-Sadi, "Simple method for determining 3-D nodal scattering in nonscalar problems," *Electronic Letters*, vol. 28, no. 25, pp. 2353–2354, 1992.

CHAPTER 3

Parallel computing strategies for TLM

3.1 Review of current parallel computing strategies

For several decades, parallel computing has been utilised to distribute the computation of complex and time-consuming calculations that would otherwise require vast amounts of time to be completed, even with the cutting edge technology on hand at the time. An example of this is the rendering of Computer Generated Imagery (CGI) used in films and television to add special effects, or in the case of computer animated media, the entire film. With the use of distributed parallel computing systems, during the production of computed animated films, typically several hours are required to render each frame, equating to several hundred-thousand computing hours for a complete film to be completed.

This chapter analyses several different schemes for implementing parallel processing (both with local and distributed systems) within software applications in order to improve the performance of the application in question.

3.1.1 Message passing interface (MPI)

MPI is not a programming language in itself, but rather it is a specification for how parallel processing libraries adhering to the MPI standard should behave. MPI computing libraries are available for the vast majority of programming languages that are

in use today, and despite not being endorsed as by any standards organisation, MPI has become the de facto choice for many parallel processing applications for distributed-memory systems [3.1][3.2].

Applications using MPI take advantage of parallel computing by executing a copy of the application on each computing node of a parallel computing system. Because each node is executing a copy of the program, local calculation results at the node level are stored in local RAM. Globally-required data is then broadcast as needed to the co-ordinating, or *master* process over the network, which then collates the received data as required. This is a major advantage of the MPI specification, since it provides a straightforward method of passing data between computing nodes on a distributed memory system, such as a HPC cluster.

3.1.2 OpenMP

As with MPI, OpenMP [3.3] is not a language, but an Application Programming Interface (API) that is freely available for use with the C/C++ and FORTRAN programming languages. OpenMP support is included with many popular compilers, such as GCC and within Microsoft Visual Studio C++ 2005, 2008 and 2010. Unlike MPI, where MPI calls are made in-line, OpenMP calls are in the form of `#pragma` pre-processor directives which are then dealt with at compile time depending on the specific compiler implementation of the OpenMP specification.

OpenMP is useful in shared memory multiprocessor scenarios, where several processors are able to address the same shared memory space. Modern desktop computers are often multi-core machines, where a single physical CPU in fact contains two or more physical processing cores on the same piece of silicon. Each of these cores is able to compute information independently and access the installed RAM, which is shared between the cores at the hardware level. Whilst increasing processor speeds was previously the driving force behind increasing computer performance, computers today and in the future use increasing numbers of processing cores sharing the load to improve performance further.

Utilising OpenMP calls allows the developer to divide any time intensive sections of an

application between the available processing cores on a machine, with each working on a portion of the problem at the same time. In the case of loops, each processor will work on an approximately equal number of iterations of the loop at the same time in order to complete calculations for that section of the application faster.

3.1.3 Hybrid parallelisation

It has been shown that MPI is very useful for a networked cluster of machines, since copies of an application are run on each node, with data being sent between nodes as required. However, this does not make sense when attempting to parallelise an application that will be run on a single multi-core machine. Firstly in this case, since all memory is local and shared, there is no need for each processor to be executing a copy of the application keeping redundant copies of local data. Secondly, using MPI to pass messages between instances of the application being executed introduces an overhead, which in this case is largely redundant since OpenMP could be used to address the shared memory directly with a lower overhead. Conversely, OpenMP is not suitable in the networked cluster scenario, since it is not designed for dealing with non-shared memory spaces. This restricts its use to local systems with a single shared block of RAM.

Many modern cluster implementations utilise multi-core CPUs at the computing node level. These extra cores would be made redundant in a MPI implementation that ran one instance of an application per node, on a single core. In the case of one instance of a program running per core thread, a pure MPI implementation would still have room for improvement as discussed above. The accepted strategy for modern clusters is to execute one application instance per node, using MPI to communicate between nodes, and to use OpenMP at the processor core thread level to further parallelise computations and maximise the performance gains observed overall [3.1][3.2]. In fact, this is the implementation method used with many of the most powerful supercomputers in the world.

3.1.4 Open computing language (OpenCL)

OpenCL is a low-level API for parallel processing using heterogeneous combinations of processors such as CPUs, GPUs or other processors. It is an open standard maintained by the Khronos Group [3.4], however commercial implementations have also been made available by several technology companies such as Apple [3.5] and NVIDIA [3.6].

The OpenCL API consists of a language (based on C99 [3.7]) that is used to write *kernels*, or functions designed to run in parallel on OpenCL-compatible devices. These kernels can accelerate software performance by focussing on either task or data-based parallelism, or a combination of both approaches. One advantage of OpenCL is the portability of the implemented kernel programs across a number of different hardware platforms and architectures without modification to the code. This is achieved by platform-specific parallel computing technologies being abstracted away from the developer. Another advantage of the API is that applications are able to utilise the GPU for parallel processing that is unrelated to graphics operations, in a similar fashion to CUDA on NVIDIA devices. However unlike CUDA, the OpenCL abstracted memory and execution model simplifies software development to an extent as well as ensuring cross-platform compatibility.

On computing systems with a combination of relatively performant CPU and GPU processors, utilising OpenCL if possible offers a method of maximising processing performance when compared to the performance of either processor in isolation.

3.2 Parallel computing & the TLM method

Over the years, there has been an increasing interest in taking advantage of parallel processing strategies in order to improve the performance of the TLM algorithm. Traditionally the scatter and connect process takes place sequentially across all nodes in a mesh, but the nature of the TLM algorithm means that computations may be carried out simultaneously on all node elements to process the results for the subsequent time step. It is possible to implement an approach where the simulation mesh is divided into smaller portions during parallel computation, with each sub-mesh being worked by a

discrete processor. Calculations for these sub-meshes can therefore be worked upon for subsequent time steps simultaneously, and information at the sub-mesh boundaries exchanged with neighbouring sub-meshes at the relevant time during the *connect* phase of the TLM algorithm.

3.2.1 Graphics processing unit (GPU) strategies

GPUs are specialised microprocessors that are traditionally highly-optimised for the parallel computation of graphics data for use in applications such as rendering frames in video games and video encoding/decoding. Floating-point support was introduced in DirectX 9.0 onwards, and the Nvidia CUDA API was released in 2006 to allow code written in C to execute in parallel on supported GPUs. This allowed 'traditional' computations that were intended for CPU processing to be executed in parallel on the GPU for improved performance. The typical internal GPU architecture is optimised to execute calculations on large datasets in parallel with low interdependency, as seen in typical graphics rendering scenarios. Since TLM meshes for complex problems often contain large numbers of nodes, and the scatter and connect process is easily parallelised, TLM is an ideal candidate for execution on a GPU.

One of the first investigations of the use of CUDA to parallelise a 2D-TLM solver showed that by utilising one of the most powerful commercially-available GPUs at the time of publication (2008) a 570% improvement was demonstrated over the solver when running on the desktop machine using only one core for processing [3.8]. The CUDA implementation had a peak performance of 200 million nodes.s^{-1} on a mesh containing approximately 15 000 000 2-D nodes. This equates to a single time step taking 0.075s with the CUDA version of the solver, and nearly 2.0s using the non-parallel implementation [3.8].

3.2.2 Multicore & symmetric multiprocessor strategies

Rossi has given an example of a SMP TLM system, implemented using OpenMP [3.9]. As an extension of the work described above in section 3.2.1 [3.8], the 2-D TLM solver that had been implemented was also modified to implement OpenMP parallelisation

strategies, and full benchmarking was conducted between the 'serial' version running using just one core for processing, along with the OpenMP version using all 4 available cores for processing in parallel. Finally these results were also compared with the same 2-D solver being run using the GPU method described previously.

As expected, both parallelised versions of the solver showed better performance than the serial version. The results demonstrated that the parallelised version of the solver using OpenMP showed on average a 165% performance improvement when compared to the serial version of the solver. Furthermore, the GPU version was 150% faster still when compared to the OpenMP version [3.9], or approximately 413% faster when compared to the serial version.

3.2.3 High performance computing (HPC) cluster strategies

Lorenz et al. gives an example of a fully implemented cluster-based TLM solver [3.10]. This implementation demonstrates a working TLM solver optimised for microwave frequencies operating on both homogeneous and heterogeneous clusters of computers connected via a LAN. Analysis of the solver provides verification of results by comparing the observed simulated resonant frequency of a cavity resonator with theoretical values. In the paper, Lorenz proved that different structures measuring many wavelengths in size could be modelled using different configurations of commodity level distributed computing systems by executing simulations on large-scale 3D models of vehicles and analysing the results.

TLM is inherently suited to parallelisation due to the nature of the underlying algorithm. Within the calculations for any given time step to be processed, the output port values for each node are completely independent from the output values of all other nodes present in the mesh. This allows the scatter and connect calculations to be processed simultaneously without introducing additional computational complexity or undesirable effects to the simulation. There has therefore been much scrutiny of contemporary efforts towards producing efficient parallel implementations of the TLM method. Stothard completed one such review of previous attempts at parallelising the TLM method in his Thesis [3.11]. In this review, a number of papers concerning implementations are discussed [3.12, 3.13, 3.14, 3.15, 3.16, 3.17, 3.18], and comparisons

between the different methods are also given. Each paper identifies and confirms the large potential for performance increases made possible by parallelising a TLM-based solver. Performance increases from 3.5 – 70 times normal were observed during testing of the various implementations by their respective authors. Despite this, it was also shown that communications with the mesh during simulation execution had a large bearing on the specific speed-up factor observed. So et al. [3.13, 3.14] demonstrated that by injecting a Gaussian-based pulse into their mesh over several time steps instead of a single impulse, the overall performance increase when compared to a single processor dropped approximately 50%.

In his review, Stothard identified the maximum mesh dimensions as a major limiting factor of the implementations that were critiqued (except for the implementation discussed by Parsons et al. in [3.17]). In every other implementation [3.12, 3.13, 3.14, 3.15, 3.16] and [3.18], the approach was to map a single node to each processor available in the specific cluster computer used. It is quite clear, especially with the advances in modern processors and the amount of RAM available on a typical system since these papers were published, that this is not a practical approach for a modern parallel TLM solver implementation. Modern cluster-based parallel computers are almost exclusively based on multi-processor and/or multi-core technology at the computing node level. It is apparent therefore, that a modern parallel implementation of a TLM-based solver can take advantage of the hybrid approach as discussed previously in Section (3.1.3). With this approach, the simulation mesh is divided into N sub-meshes (where N is the number of computing nodes available for use on a cluster system). This efficient use of computing resources by placing several nodes on each computing node allows for large, complex full-field problems to be simulated. As well as being parallelised at the computing node level via MPI or a similar message-passing scheme, further performance improvements through parallelisation can be achieved by simultaneous computation of sub-meshes at the processor core level via the use of the OpenMP library to divide the scatter and connect loop iterations for the sub-mesh between cores to be worked upon at the same time.

3.3 Conclusions & impact on research strategy

It has been the aim of this chapter to provide a review of the current literature relating to the research that is currently under way. The review of the literate has allowed several considerations concerning the specifics of the research strategy to be formed, with each being based upon previous work in the field. This approach should help result in a well-informed research strategy that bears direct relevance to the previous studies published, building upon them in a logical fashion. The following conclusions and observations have been drawn from conducting the literature review:

In considering the various options for the utilisation of a parallel computing architecture, it quickly became apparent that a hybrid approach that combined the use of both MPI and OpenMP would present the best potential for maximising performance increases when running the solver on HPC cluster systems. This conclusion came from the fact that OpenMP is designed only for use with SMP architectures and that a purely MPI-based implementation of parallelisation would be introducing unnecessary overheads at the computing node level. By design, a cluster is not a SMP system since each computing node executes a copy of the program, with no shared memory between nodes, and MPI communication between processor cores within a node would be slower than using OpenMP to parallelise a program and have each processor core thread address the on-board shared RAM.

Whilst also being supported by the literature [3.1][3.2], this hybrid approach is also justified by its use in many of the supercomputers in the TOP500 list of the five hundred most powerful supercomputers in the world. Using a GPU as an alternative method to produce performance increases was also explored within the review of studies concerning parallel implementations of the TLM algorithm [3.8]. Whilst the studies to date in this area have demonstrated significant performance improvements on a single computer basis when compared to MPI or MPI/OpenMP implementations, it must be said that computing clusters do not contain high-end GPUs, so the performance gains are most often limited to a single machine. Also, using CUDA or OpenCL in order to use the GPU for general purpose computing operations requires extra development effort with regards to memory management in order to maximise the use of the faster in-

ternal memory available within the GPU, which in turn maximises performance gains. Finally, using the GPU method goes against the original intention to produce a solver that is as platform independent as possible, with the option of making portions of the code open source for community modification and improvement.

References

[3.1] P. S. Pacheco, *Parallel programming with MPI.* Morgan KaufmaMcGraw-Hill, San Francisco, 1997.

[3.2] M. J. Quinn, *Parallel programming in C with MPI and OpenMP.* McGraw-Hill, New York, 2004.

[3.3] "OpenMP application programming interface." `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`.

[3.4] "OpenCL specification version: 2.0." `https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf`.

[3.5] "OpenCL for OS X." `https://developer.apple.com/opencl/`.

[3.6] "OpenCL NVIDIA implementation." `https://developer.nvidia.com/opencl`.

[3.7] "C99 specification - ISO/IEC 9899:TC3." `http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf`.

[3.8] F. V. Rossi, P. P. M. So, N. Fichtner, and P. Russer, "Massively parallel two-dimensional TLM algorithm on graphics processing units," in *IEEE MTT-S Intl. Microwave Symp. Digest*, pp. 153–156, 15-20 June 2008.

[3.9] F. V. Rossi, "Graphics hardware accelerated transmission line matrix procedures," Master's thesis, University of Victoria, 2008.

[3.10] P. Lorenz, J. V. Vital, B. Biscontini, and P. Russer, "High-throughput transmission line matrix (TLM) system in grid environment for microwave design, analysis and optimizations," *IEEE MTT-S Intl. Microwave Symp. Digest*, pp. 1115–1118, 2005.

[3.11] D. Stothard, *The development of an application specific processor for the transmission line matrix method*. PhD thesis, Loughborough University, 1999. pp. 17-27.

[3.12] J. L. Dubard, O. Benevello, D. Pompei, J. L. Roux, P. P. M. So, and W. J. R. Hoefer, "Acceleration of TLM through signal processing and parallel computing," in *Intl. Conf. on Computation in Electromagnetics*, pp. 71–74, 1991.

[3.13] P. P. M. So and W. J. R. Hoefer, "Optimization of microwave structures using a parallel TLM module," in *10th Annual Review of Progress in Applied Computational Electromagnetics*, pp. 546–553, 21-26 March 1994.

[3.14] P. P. M. So, C. Eswarappa, and W. J. R. Hoefer, "Transmission line matrix method on massively parallel processor computers," in *9th Annual Review of Progress in Applied Computational Electromagnetics*, pp. 467–474, 22-26 March 1993.

[3.15] P. O. Luthi, B. Chopard, and J. F. Wagen, "Wave propagation in urban micro-cells: a massively parallel approach using the TLM method," in *Applied Parallel Computing in Physics, 2nd Intl. Workshop*, pp. 408–418, 1995.

[3.16] C. C. Tan and V. F. Fusko, "TLM modelling using a SIMD computer," *Intl. Journal of Numerical Modelling: Electronic Networks, Devices and Fields*, vol. 6, no. 4, pp. 299–304, 2005.

[3.17] P. J. Parsons, S. R. Jaques, S. H. Pulko, and F. A. Rabni, "TLM modelling using distributed computing," *IEEE Microwave and Guided Wave Letters*, vol. 6, no. 3, pp. 141–142, 1996.

[3.18] H. Li and Q. F. Stout, "Reconfigurable SIMD massively parallel computers," *Proceedings of the IEEE*, vol. 79, no. 4, pp. 429–443, 1991.

Chapter 4

Geometric modelling of raindrops

The example case study of accurate modelling of rain-fields for use with a 3-D TLM implementation was introduced previously in section (1.4). As discussed earlier, accurately modelling the behaviour of EM signals propagating through rainfall is of interest for a number of applications, for example line-of-sight EHF transmission links. With this class of application, accurate predictions of the link performance are crucial to the viability of the end product. It is therefore clear that improvements to the accuracy of rain-field modelling are of interest both academically and commercially.

Previous studies into the depolarisation and attenuation due to rainfall have mainly focussed on approximation models, measured results or verification of a model against measured results [4.1, 4.2, 4.3, 4.4]. Any approximation model of raindrop behaviour will be based on one or more assumptions, and therefore the model will inherently possess limitations in certain cases when compared to real-world observations of depolarisation and attenuation due to rainfall. Conversely, any real-world data will have its own limitations due to its inflexibility and the difficulty in being able to tailor a dataset to a particular end user application.

In this case, the proposed compromise is to model the geometry of individual raindrops within a given rain-field. This provides flexibility in the parameters of the rain-field model, such as raindrop size range and drop density. Modelling individual raindrops whilst placing them randomly also allows the rain-field model to more closely approximate a rain-field that might be observed in nature.

Before a rain-field can be successfully simulated, a geometrical model for individual raindrops must first be researched, and a suitable discretisation method chosen. This forms the first part of the process of preparing discretised mesh information ready for interpretation by a 3-D solver.

To be successful, any method for generating rain-fields should attempt to adhere to the following criteria:

- The geometrical raindrop model should closely approximate real-world observations.
- The simulated raindrops within a rain-field should adhere to a suitable size distribution.
- The simulated raindrops should be randomly sized and placed within the rain-field.
- However, the simulated raindrops should also avoid intersecting with one-another.
- The discretisation method should be accurate and flexible.

This chapter presents solutions to the above criteria, demonstrating a comprehensive method for generating randomly-generated rain-fields ready for use with a 3-D solver.

4.1 Review of raindrop geometry modelling techniques

4.1.1 Review of available models

In order to produce accurate attenuation and depolarisation results when injecting a given signal into the simulated rain field environment, there must also be an accurate representation of both drop geometry and size distribution. There are several examples of papers that have been published over the last several decades [4.5, 4.6, 4.7] concerning the geometry and sizes of raindrops with subsequent papers demonstrating improvements and refinements to the models that have been presented previously.

In popular art and literature, raindrops are frequently depicted as teardrop shaped. The fundamental effects of surface tension on liquids counter this popular opinion,

and in fact studies into raindrop geometry have conclusively proven that this is not the case [4.5, 4.6, 4.7]. Experimental data from multiple studies using devices such as disdrometers to measure the size and shape of drops has demonstrated that in the case of small raindrops (<2.0 mm in diameter) the drops appear to be approximately spherical in shape. This shows that in the case of drops of this size and smaller, the surface tension and internal hydrostatic pressure are the dominant factors dictating the drop geometry, and not the air pressure on the bottom surface of the drop as it falls through the air.

Many different studies have attempted to produce a suitable geometrical model for drops larger than 2.0 mm in diameter. In these papers, the volume of the raindrop is the limiting factor, and as such, the diameter figures that are stated are the equivalent undistorted diameters of a drop if it were to retain its spherical in shape as it fell. Pruppacher & Pitter [4.5] having conducted one such study, produced a model for raindrops up to 8.0 mm in diameter. A later study by Beard & Chuang [4.6] demonstrated an improvement to this model, with verification of results based on photographic observations from a two-dimensional video disdrometer to validate the accuracy of the model against their observations. This model has also been utilised in several subsequent related studies, such as [4.8]. The main improvement of the Beard & Chuang Model over previous models was the fact it addressed the inverse curvature of the underside of the drop in the Pruppacher & Pitter Model, which increased as a function of drop size. The photographic observations by Beard & Chuang showed no evidence of such an indentation within photographed drops as they fell [4.6, 4.7]. In fact, even at very large drop sizes (8.0 mm+) despite significant flattening to the underside of the drop, the overall curvature of the drop surface at any given point remained positive. During computer modelling, this fact produced two effects when coupled with the symmetry of the drop in the vertical plane. Firstly, only half of the drop edge outline needs to be computed, before a rotational sweep is applied to produce a solid model of a given drop. Secondly, due to the overall positive curvature of the drop at any given point, the cross section of the drop through its centre can also be considered to be a profile image of the drop when viewed side on.

4.1.2 Previous investigations to date

Following the investigations into the shape of raindrops whilst in an equilibrium state, subsequent papers have attempted to gain an in depth understanding of the oscillation of raindrops during their fall to the earth in both laboratory and real-world conditions. Recent research has investigated how the possible causes for these oscillations, where wind and collisions factors have previously been ruled out as a cause for the oscillations observed in drops larger than 1.0 mm in diameter [4.7]. In this paper, the mechanics of raindrop geometry are discussed. The factors causing increasing deformation as drop diameter increases are identified as both the increasing weight and also an increased differential between air pressure above and beneath the drop as it falls. The paper then discusses observations of simulated rain in calm wind conditions using an 80m fall to allow drops to stabilise and achieve terminal velocity. Raindrop shape measurements were also obtained from rain events in several real-world locations, and comparisons were then made with the experimental results.

Regarding the intended application for the solver software, from reviewing the literature it is clear that extensive research has been undertaken over the last decades towards understanding raindrop geometry. Increasing use of the EHF portion of the EM spectrum has led to a need for a better understanding of the attenuation and depolarising effects of rain on EM radiation at these extremely high frequencies. Following a review of the different raindrop models available, the Beard and Chuang Model [4.6] was selected for use in the solver to generate raindrop geometries within the TLM mesh. This was based upon the accuracy of the model when compared to photographic observations of both simulated and real-world raindrops. This model also demonstrated improvements upon earlier models such as the Pruppacher and Pitter Model [4.5]. Finally, whilst recent papers have discussed the oscillations present in larger drops, the focus for the solver will be to implement steady-state drops in order to avoid overcomplicating the solver software. If time allows at a later date, there will be the potential to implement drops with oscillations present during their fall based upon the observations from the latest literature [4.7].

4.1.3 Summary of the Beard & Chuang model

As well as offering a detailed, parameterised model, Beard & Chuang [4.6] also describe a simplified approximation to their model which requires much less effort to implement. This was chosen as the basis of implementing the model within this thesis. The model approximation is in the form of a cosinusoidal distortion of a circle representing the cross-section of a given raindrop. The method for calculating the drop radius, r at any given point along its cross-sectional circumference is given in polar form by Equation (4.1). This distortion is computed from a table representing the relative weights given to 0th to 10th order cosinusoidal calculations relative to the angular position along the circumference of the drop. This adjustment factor is then applied to the equivalent unmodified radius of the raindrop, a to be modelled. The distortion coefficients, C_n are shown in Table (4.1). The coefficients have been scaled up by a factor of 10^4 for tabulation.

	Distortion coefficients, ($C_n \times 10^4$)										
$d(mm)$	$n=0$	1	2	3	4	5	6	7	8	9	10
2.0	-131	-120	-376	-96	-4	15	5	0	-2	0	1
2.5	-201	-172	-567	-137	3	29	8	-2	-4	0	1
3.0	-282	-230	-779	-175	21	46	11	-6	-7	0	3
3.5	-369	-285	-998	-207	48	68	13	-13	-10	0	5
4.0	-458	-335	-1211	-227	83	89	12	-21	-13	1	8
4.5	-549	-377	-1421	-240	126	110	9	-31	-16	4	11
5.0	-644	-416	-1629	-246	176	131	2	-44	-18	9	14
5.5	-742	-454	-1837	-244	234	150	-7	-58	-19	15	19
6.0	-840	-480	-2034	-237	297	166	-21	-72	-19	24	23

Table 4.1: Cosinusoidal distortion coefficients for the Approximated Beard & Chuang model.

$$r = a\left(1 + \sum_{n=0}^{10} C_n \cos(n\theta)\right) \tag{4.1}$$

4.2 A rain-field modelling strategy

A robust raindrop modelling method had to be developed as part of the overall objective of having a parallelised, cluster-based 3-D TLM solver which is specialised for the analysis of the effects of rainfall on EHF signals. Following the selection of a suitable

raindrop geometry model (as discussed in section 4.1.), the modelling scheme can be designed and implemented. A suitable software package was required that could model a set of raindrops and then *mesh* them, converting them to node placement information files to be used by the solver and inserted into its TLM node mesh at a later date. SketchUp [4.9] was utilised as it is a free Computer Aided Design (CAD) software package that offers customisability via the optional use of custom script files written in the Ruby [4.10] language that can be used to automatically generate specific geometry. To this end, a script was implemented that was designed to generate random "blocks" of rainfall adhering to the following criteria:

1. The raindrop diameter is randomised according to a discretised normal distribution based on the possible values provided by the Beard & Chuang model (2.0mm – 6.0mm in 0.5mm steps).
2. Randomised X, Y, Z co-ordinates are generated for each drop centre position, provided:
   (a) The drop would not intersect another that has already been placed (in the real-world these two drops would coalesce into a larger drop)
   (b) The drop would not intersect the edge of the simulation volume (as this could produce undesirable results during a simulation since a partial raindrop would be present in the mesh).
3. Raindrops are added within the desired volume until a given volume ratio is reached (this ratio can be equated to a real-world rainfall rate via a simple calculation).

4.2.1 Calculating raindrop size occurence probabilities

The approximated Beard & Chuang raindrop model provides raindrop geometry information for raindrops of an equivalent diameter between 2.0 - 6.0mm inclusive at 0.5mm intervals. This forms a discrete set of raindrop sizes that can be modelled, as compared to a truly continuous size distribution in the real-world. The size distribution for the raindrops to be modelled is assumed to be normally-distributed. A suitable normal distribution curve for the approximated Beard & Chuang model can be constructed

by first calculating the variance, σ^2 of the sample set as in equation (4.2). Knowing the value of σ^2, it is possible to construct a standard normal distribution for the raindrop geometry model by calculating Z scores using equation (4.3). The resulting distribution with a mean value μ, of 4 and a standard deviation σ, of 1.36931 is shown in Fig. (4.1).

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n} |x_i - \mu|^2 \tag{4.2}$$

$$Z = \frac{X - \mu}{\sigma} \tag{4.3}$$

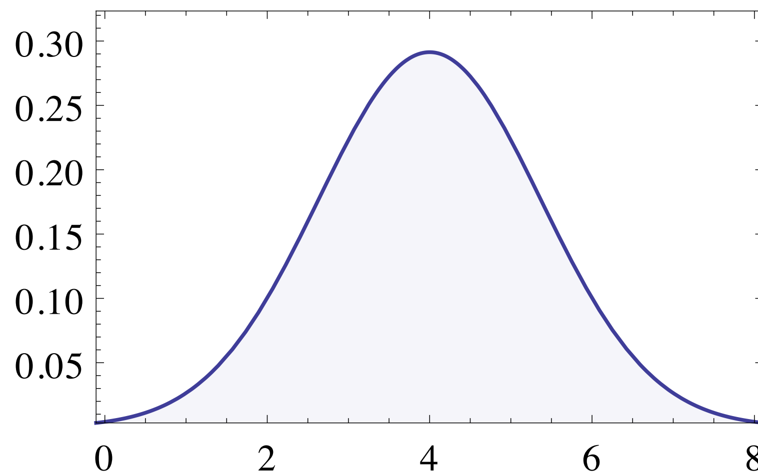


Figure 4.1: Normal distribution for the approximated Beard & Chuang raindrop geometry model. The x-axis shows the raindrop diameter (mm), and the y-axis shows the probability of occurrence.

An important consideration is the need to calculate the occurrence probabilities for each differently sized raindrop in the approximated Beard & Chuang model. It is possible to compute these probabilities by dividing the standard normal distribution into "bins", or slices each of which represents a probability of observing a raindrop of a particular diameter. As the model has an interval of 0.5mm in diameter between raindrop sizes, it is possible to centre the bins around each allowable raindrop size by calculating the cumulative probability +/-0.25mm from a particular diameter. Z scores relating to these intervals are calculated from equation (4.3) and a typical standard normal distribution look-up table provides the equivalent cumulative probability values. Subtracting one probability from the other allows the occurrence probability for each raindrop to be

determined. A summary of the calculated occurrence probability values is shown in Table (4.2).

d(mm)	P(d)
< 2.0	0.0502
2.0	0.0504
2.5	0.0801
3.0	0.1112
3.5	0.1357
4.0	0.1448
4.5	0.1357
5.0	0.1112
5.5	0.0801
6.0	0.0504
> 6.0	0.0502

Table 4.2: Occurence probability $P(d)$ for raindrops of diameter d.

4.2.2 Generating a discrete random variable for raindrop diameter

Constructing a discrete random variable (DRV) is a method for generate randomised values which occupy specific discrete values that follow a particular probability distribution. A typical method for generating a discrete random variable is to utilise a random number generator that produces uniformly-distributed random numbers and apply a suitable algorithm or other transformation in order to generate discrete values of the required distribution.

In this case we already know the discrete values of our set (2.0 - 6.0mm inclusive at 0.5mm intervals) that should be the possible outputs of the discrete random variable. The required probability distribution has previously been calculated in section (4.2.1) with the occurrence probabilities shown in Table (4.2). A typical maths library within a programming language will offer a random number generator that is capable of producing numbers within any specified range. Random numbers can also be generated using other methods, such as an XOR shift method proposed previously [4.11]. Two approaches to constructing a discrete random variable are shown below.

An initial approach

An initial approach to the problem would be to hold multiple copies of every raindrop diameter from the set in an array (including the outliers, which will be stored as 0.0 and 9.0 for recognition purposes so that they can be discarded later in the software) and then pick from the array using a random number generator. The number of copies of each diameter value is determined by the precision at which the occurrence probabilities have been calculated. In this case, the probability values are to 4 decimal places, so a 10 000 element array is required. In this case the random number generator is configured to produce integers in the interval 0 - 9 999 and the generated value is then used as the index for the V_d array to select a stored raindrop diameter value for use in modelling. A representation of the array in this case is shown in Table (4.3).

V_d **array index**	**Diameter (mm)**
[0]. . . [501]	0.0
[502]. . . [1005]	2.0
[1006]. . . [1806]	2.5
[1807]. . . [2918]	3.0
[2919]. . . [4275]	3.5
[4276]. . . [5723]	4.0
[5274]. . . [7080]	4.5
[7081]. . . [8192]	5.0
[8193]. . . [8993]	5.5
[8994]. . . [9497]	6.0
[9498]. . . [9999]	9.0

Table 4.3: A representation of the array holding the discrete random variable V_d using a naive construction approach.

A more memory-efficient approach

A large array holding multiple duplicate raindrop diameter values is not an optimal solution to the problem of producing a discrete random variable for modelling raindrops of different sizes. Whilst a 10 000 element array may not be particularly large by modern standards, the storage requirements grow by a factor of 10 for every extra decimal place of precision in the probability distribution. This means for applications requiring additional precision, the array quickly becomes very large, and will consume large amounts of available RAM.

It was evident in the naive approach that there is significant duplication of data in the form of raindrop diameter values stored in the array. However, it is possible to preserve the overall probability distribution whilst significantly reducing the memory requirements to produce the discrete random variable. The algorithm for implementing this memory-efficient version of the discrete random variable is detailed below, and based on methods proposed previously [4.12, 4.13].

1. Reorganise the 10000 element array so that the raindrop diameter values are stored in order of the decimal place digits present in each occurrence probability value, I.e.
   - 1000×3.0, 1000×3.5, 1000×4.0, 1000×4.5, 1000×5.0
   - 500×0.0, 500×2.0, 800×2.5, 100×3.0, 300×3.5, 400×4.0, 300×4.5, 100×5.0, 800×5.5, 500×6.0, 500×9.0
   - 10×3.0, 50×3.5, 40×4.0, 50×4.5, 10×5.0
   - 2×0.0, 4×2.0, 1×2.5, 2×3.0, 7×3.5, 8×4.0, 7×4.5, 2×5.0, 1×5.5, 4×6.0, 2×9.0
2. It is now evident that the start of the array contains 5 blocks of 1000 identical values. This could be replaced by simply select a number from an array containing {3.0, 3.5, 4.0, 4.5, 5.0} if the random number generator produces an output < 5000. The array index, i would calculated by `i = (int)RNG/1000`, where 'RNG' is a random number between 0-9 999. A similar algorithm can be applied to the subsequent blocks of duplicate values representing the hundredths, thousandths and ten-thousandths decimal places.

With this scheme, the discrete random variable for raindrop diameter, V_d can be implemented as follows in Ruby:

```
#Raindrop diameter values - 0.0 and 9.0 for N-dist tails
r = [0.0, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0, 9.0]

#Memory-efficient arrays for occurrence probabilities up to 4 decimal place
    precision
tenths = [r[3], r[4], r[5], r[6], r[7]]
hundredths = [r[0], r[0], r[0], r[0], r[0], r[1], r[1], r[1], r[1], r[1], r
    [2], r[2], r[2], r[2], r[2], r[2], r[2], r[2], r[3], r[4], r[4], r[4], r
```

```
    [5], r[5], r[5], r[5], r[6], r[6], r[6], r[7], r[8], r[8], r[8], r[8], r
    [8], r[8], r[8], r[8], r[9], r[9], r[9], r[9], r[9], r[10], r[10], r[10],
     r[10], r[10]]
thousandths = [r[3], r[4], r[4], r[4], r[4], r[4], r[5], r[5], r[5], r[5], r
    [6], r[6], r[6], r[6], r[6], r[7]]
tenThousandths = [r[0], r[0], r[1], r[1], r[1], r[1], r[2], r[3], r[3], r[4],
     r[4], r[4], r[4], r[4], r[4], r[4], r[5], r[5], r[5], r[5], r[5], r[5],
    r[5], r[5], r[6], r[6], r[6], r[6], r[6], r[6], r[6], r[7], r[7], r[8], r
    [9], r[9], r[9], r[9], r[10], r[10]]

#return a N-distributed, randomly generated drop diameter value
until (randDiameter > 0.0 && randDiameter < 9.0) do
        diameterRNG = rand(10000)
        if (diameterRNG >= 9960)
                randDiameter = tenThousandths[diameterRNG-9960]
        elsif (diameterRNG < 9960 && diameterRNG >= 9800)
                randDiameter = thousandths[(diameterRNG-9800)/10]
        elsif (diameterRNG < 9800 && diameterRNG >= 5000)
                randDiameter = hundredths[(diameterRNG-5000)/100]
        else (diameterRNG < 5000)
                randDiameter = tenths[diameterRNG/1000]
        end
end
```

The result for this particular application is that the memory usage has been reduced from a single 10 000 element array to four separate arrays totalling 109 elements. This represents a 98.9% reduction in memory usage as compared to the naive scheme, whilst retaining almost the same level of performance (at the expense of 4 conditional statements during raindrop diameter value look-up).

Verification of the discrete random number variable implementation

The implementation of the DRV for generating raindrop diameter values was tested by using it to generate a large sample set of 10000 diameter values at random. These were sorted and counted to determine observed occurrence frequency values, $P_o(d)$ for each diameter and then converted to an equivalent probability value. These values were then compared with the expected values, $P_e(d)$ calculated previously in section (4.2.1).

A summary of the verification results showing absolute probability values and relative error values, E_r is shown in Table (4.4). A graphical comparision is also shown in Fig. (4.2).

d(mm)	$P_e(d)$	$P_o(d)$	E_r
< 2.0	0.0502	0.0504	+0.40
2.0	0.0504	0.0486	-3.57
2.5	0.0801	0.0786	-1.87
3.0	0.1112	0.1102	-0.90
3.5	0.1357	0.1313	-3.24
4.0	0.1448	0.1426	-1.52
4.5	0.1357	0.1410	+3.91
5.0	0.1112	0.1169	+5.13
5.5	0.0801	0.0821	+2.50
6.0	0.0504	0.0453	-10.12
> 6.0	0.0502	0.0530	+5.58

Table 4.4: Discrete random variable verification results comparing observed and expected occurrence probability values, and showing relative error values.

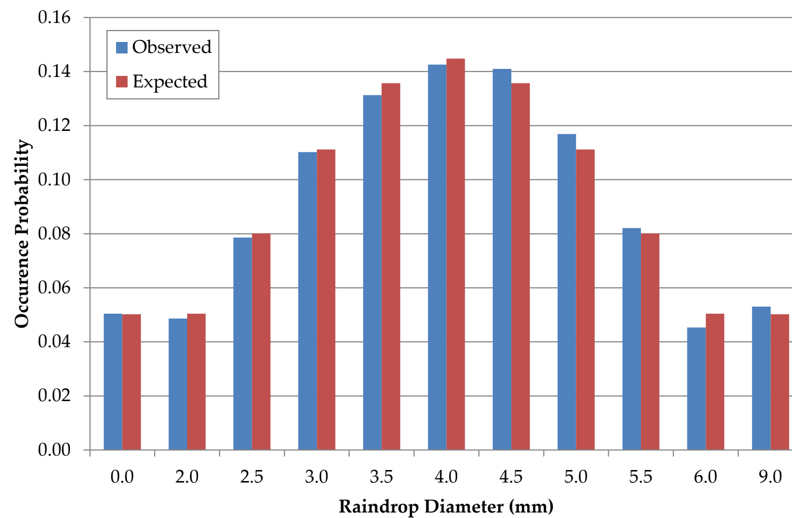


Figure 4.2: Comparison of the implemented discrete random variable (executed 10 000 times) with expected occurrence probability values calculated from section (4.2.1).

4.2.3 Discretising the raindrop geometry

A complete modelling strategy has now been outlined. This includes the overall approach (Section (4.2)), the calculation of occurrence probabilities of differently-sized drops (Section (4.2.1)), and the construction of a discrete random variable for use dur-

ing modelling (Section (4.2.2)). Suggested methods for discretising[1] the individual raindrop geometry are the subject of this section of the thesis.

Stacked circles

One method for discretising the drop geometry is that when slicing the drop along the Z-axis in single node steps, the cut-through sections formed as XY planes present themselves as perfect circles. By using the Beard & Chuang model, it is possible to compute the radius of each of these stacked circles that constitute a single drop. From this, the co-ordinates of the positions along the edge of each circle can be interpolated to the nearest node, and all nodes within the circular XY plane can be set to behave as a water-occupied node. Each of these steps can be accomplished by iterating across a square XY which exactly encompasses the desired circular plane of nodes. During the iteration across the square plane, a series of tests are conducted to test whether the node in question resides within the circle. This is summarised below in pseudo-code:

```
//Position variables...
x = nodePosX;
y = nodePosY;
xDist = abs(x - centerX);
yDist = abs(y - centerY);
R = radius;
stX = centreX - (R/2);
eX = centreX + (R/2);
stY = centreY - (R/2);
eY = centreY + (R/2);

for (int y = stY; y <= eY; y++) {
        for (int x = stX; x <= eX; x++) {
                //First test a square diamond which the circle would
                    encompass exactly...
                if (xDist + yDist <= R)
                        setNodeBehaviour(WATER_BEHAVIOUR, x, y);
                //Else, use Pythagoras to test region between the diamond and
                     circle arc...
```

[1]*Discretising* in this context is the act of taking a precisely-defined model of an object and then sub-dividing it into discrete, parallelepipic blocks of matter of a given size, based on a chosen value of $\Delta\ell$, which represents the mesh resolution, equivalent to the node size in a TLM mesh.

```
                else if (xDist^2 + yDist^2 <= R^2)
                        setNodeBehaviour(WATER_BEHAVIOUR, x, y);
                else
                        setNodeBehaviour(AIR_BEHAVIOUR, x, y);
        }
}
```

This set of tests for the construction of one of a series of stacked circles used to form a discretised equivalent of a drop is shown in Fig. (4.3). A test is shown where an arbitrary point, P at some co-ordinates, (x, y) is tested using radius r and centre-point C.

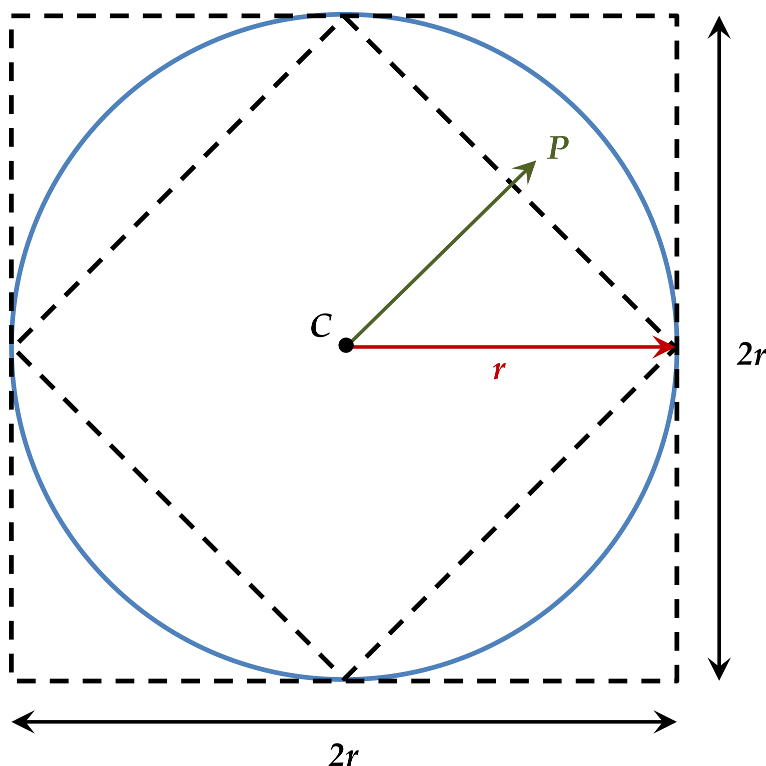


Figure 4.3: An illustration of the test used for drawing a circle of water-based nodes in the XY plane within the 3-D solver, showing center-point C, test-point P and radius r.

A custom Ruby script for SketchUp

In order to transform a piece of real-world geometry into mesh information suitable for use in a TLM solver, it must firstly be discretised. The basic premise is the act of sampling the object to be transformed at a particular spacings in all dimensions. This

sample spacing should be equal to the required $\Delta\ell$ value for the mesh node size. If the centre-point of the space bounded by the test region is occupied by part of the object, the entire voxel is deemed to be occupied, regardless of any partial occupancy in the real-world. This concept is illustrated in Fig. (4.4).

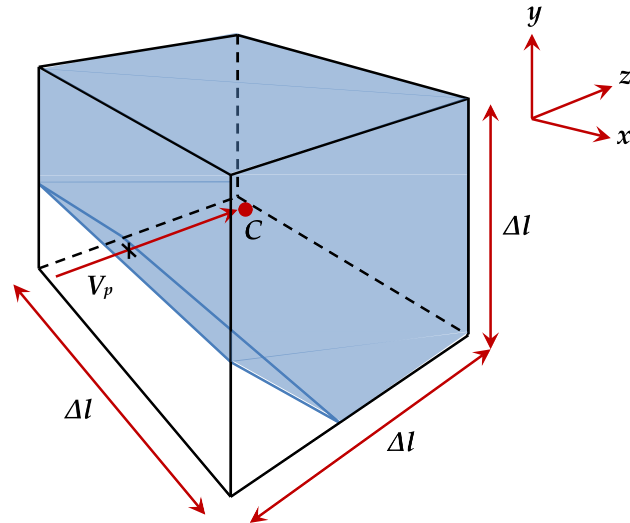


Figure 4.4: An illustration of the Voxelise process in operation, showing the voxel volume ($\Delta\ell^3$), as well as the probing vector V_p, used to test the centre-point of the voxel, C.

An open-source Ruby script for discretising 3-D objects that was designed to work with SketchUp was found and adapted to interface with the rest of the raindrop modelling script authored for this thesis [4.14]. A high-level description of operation of the script, *Voxelize* is included below (in pseudo-code):

```
# Create probing vector (in each axis direction)
xVect = Geom::Vector3d.new(corners[0],corners[1])
nx = (xVect.length/vSizeLength).round
xVect.length = vSizeLength

yVect = Geom::Vector3d.new(corners[0],corners[2])
ny = (yVect.length/vSizeLength).round
yVect.length = vSizeLength

zVect = Geom::Vector3d.new(corners[0],corners[4])
nz = (zVect.length/vSizeLength).round
zVect.length = vSizeLength
```

```
#Main loop
for yy in (0..(ny-1))
   for zz in (0..(nz-1))
      for xx in (0..(nx-1))
         pos = xVect.length * (xx+0.5)
         for nDist in intDist[yy][zz]
            if pos >= nDist
               binCubes[xx][yy][zz] = 1
            else
               binCubes[xx][yy][zz] = 0
            end
         end
      end
   end
end
```

The occupancy test method shown above iterates across a virtual bounding box, surrounding the raindrop model. A nested loop moves through an integer number of voxel spaces in each axis, testing for occupancy in each voxel space. An illustration of the resulting discretisation of part of a particular raindrop model is shown in Fig. (4.5) and Fig. (4.6).

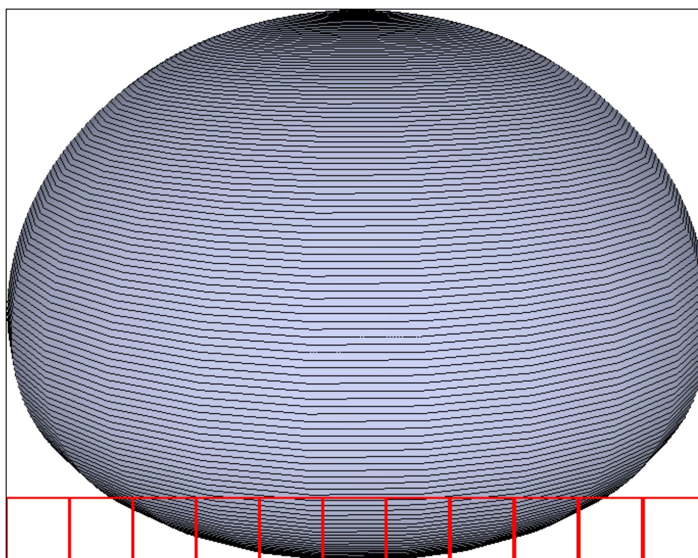


Figure 4.5: A raindrop model, shown with a bounding box and one row of the virtual mesh used during discretisation for an arbitrary value of $\Delta\ell$.

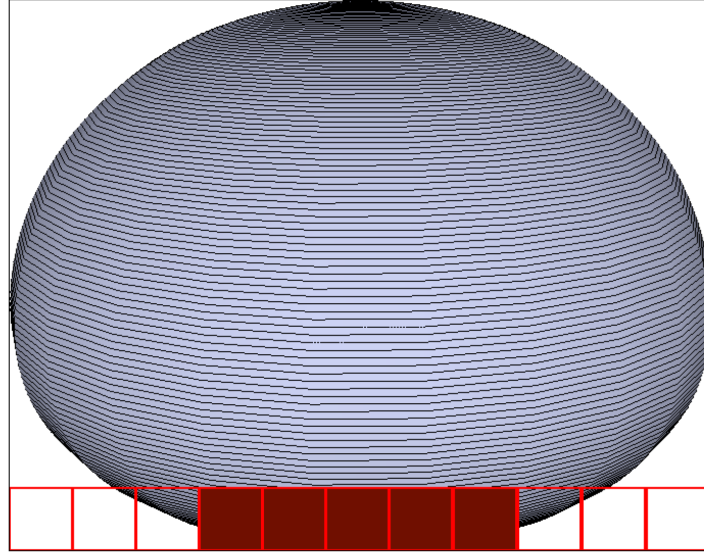


Figure 4.6: Raindrop geometry, shown with one row of the virtual mesh used during discretisation for an arbitrary value of $\Delta\ell$. Voxels set to behave as water are highlighted in dark red.

The above demonstrates that the accuracy of any discretised geometry is directly dependent on the desired node size, and hence the $\Delta\ell$ value used for the voxel dimensions during discretisation. A smaller $\Delta\ell$ results in mesh geometry which more closely approximates the original model, at the expense of higher memory occupancy and increased simulation time. An example of a single raindrop being meshed for a relatively high F_{max} value, allowing for a reasonable approximation, is shown in Fig. (4.7) and Fig. (4.8).

4.2.4 Collision detection strategies

As discussed in section (4.2.3), raindrops must not intersect, or "collide" during the simulation, to avoid spurious effects due to the irregular shape resulting from the collision. In the real world, such raindrops would in fact coalesce into one larger drop. To prevent this, a robust collision detection algorithm is needed based to check whether two drops will intersect if the new drop is placed in the environment with the current randomly-generated drop centre co-ordinates. A naïve approach is to simply consider a perfect sphere which encloses the raindrop. The radius of the sphere will be equal to the maximum radius of the raindrop from its centre of mass. This can be used in

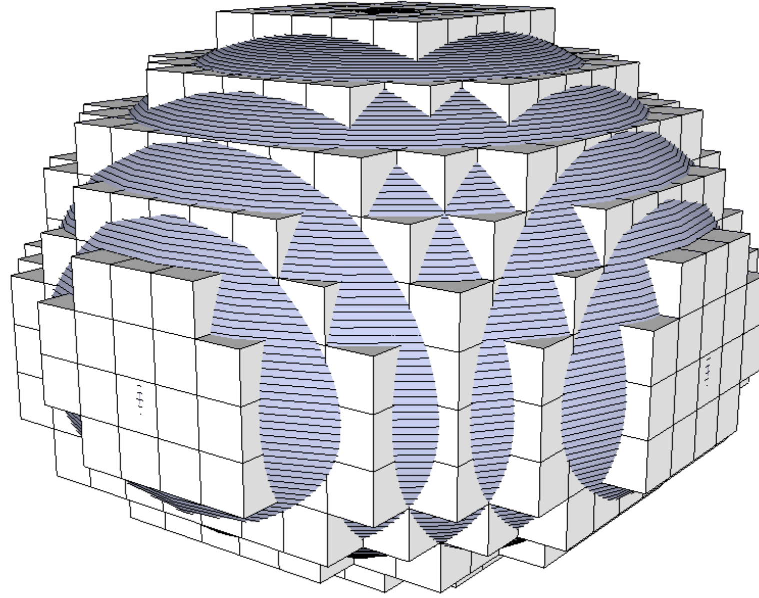


Figure 4.7: The same raindrop from Fig. (4.5) with its equivalent TLM mesh overlayed, meshed for a F_c of 10.053GHz.

a simple distance check by comparing the maximum radius of the current drop and the one it is being checked against, with the distance observed between the two drop centres. If the sum of the radii is larger than the distance computed, the raindrops are considered to have intersected, and the centre co-ordinates for the current drop will be re-generated by random number generators. This method results in no false-negative events, satisfying the modelling criteria set by ensuring drops never intersect each other. However, the use of the maximum radius of a given drop during computations does mean that occasional false-positive collision events will occur. Clearly, a more robust collision algorithm is required.

4.2.5 Implemented collision detection algorithm

Two vectors are constructed in order to find the angle between two raindrops that are to be checked for potential collisions. The first is a vector, **v** that joins the two drop centres (points A and B in Fig. (4.9)) with the second being a unit vector, **z** on the Z-axis (assuming the Z-axis is parallel to the direction of free-fall due to gravity). The acute angle between these two vectors, θ is found using equation (4.4) utilising the dot product of the vectors in the calculation. The corresponding angle for the other drop, α is also calculated, as in equation (4.5). These two angles can be utilised as inputs to the

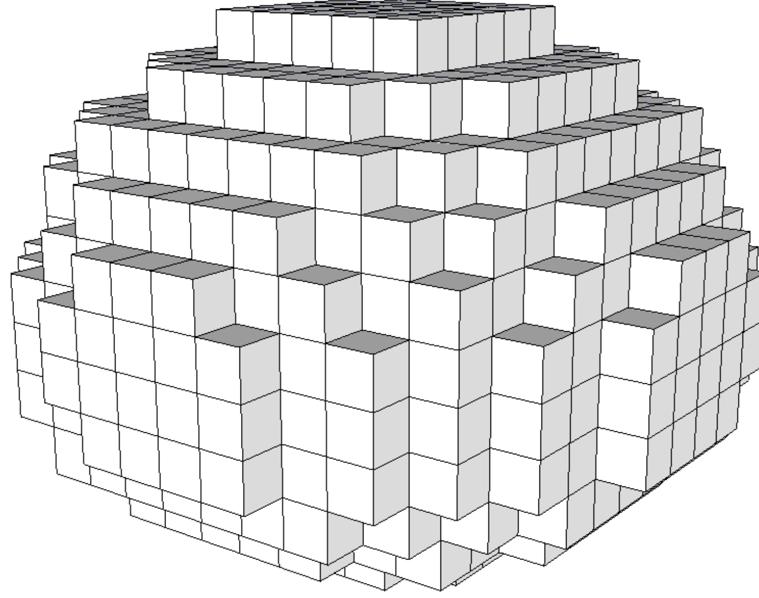


Figure 4.8: The raindrop TLM mesh data for the raindrop in Fig. (4.5) as it will appear in the simulation, meshed for a F_c of 10.053GHz.

Beard & Chuang raindrop model to calculate the exact drop radii, r_1 and r_2 at that point on their surfaces. These radii can then be used to conduct an accurate collision test, outputting a Boolean collision variable, C as shown in equation (4.6). This method will not generate false-positive results, as demonstrated with the maximum radius method detailed previously in section (4.2.4). An illustration of the vector collision detection method is shown in Fig. (4.9).

$$\theta = \arccos(\hat{\mathbf{v}} \cdot \hat{\mathbf{z}}), \quad \textit{where} \quad \hat{\mathbf{v}} = \frac{\mathbf{v}}{||\mathbf{v}||} \tag{4.4}$$

$$\alpha = \pi - \theta \tag{4.5}$$

$$C = \begin{cases} 1 & \text{if } ||\mathbf{v}|| < (r_1 + r_2), \\ 0 & \text{if } ||\mathbf{v}|| \geq (r_1 + r_2). \end{cases} \tag{4.6}$$

The result is that this method can be used to ensure several raindrops can be placed randomly within a simulation environment quickly and efficently, whilst no two drops intersect with oneanother, as well as not intersecting with the edges of the simulation volume. An example execution of the raindrop modelling script is shown in Fig. (4.10).

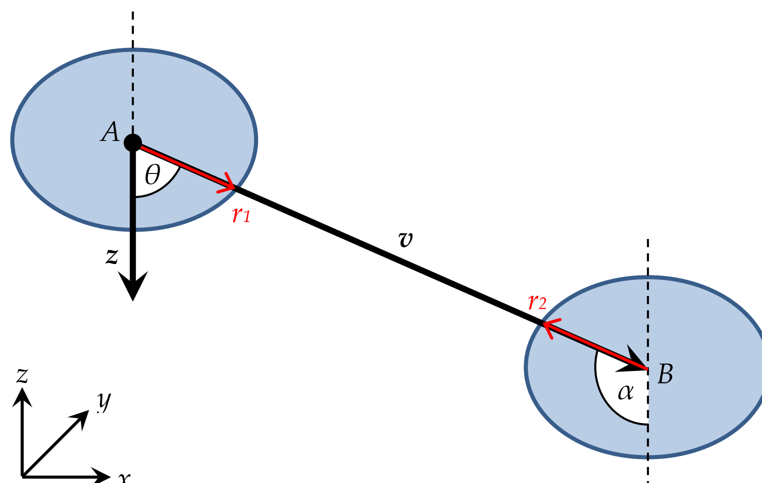


Figure 4.9: Collision detection between raindrops in close proximity using vectors.

4.2.6 Summary of raindrop modelling method

As referenced previously (in Section (4.2)), SketchUp was chosen as the CAD software used to model the raindrops within the required environment volume, and then discretise the geometry before outputting the mesh information for import into the desktop solver. The main considerations that informed this descision were:

1. SketchUp is free to download, allowing ubquitous adoption of any implemented modelling scheme with other interested parties.

2. Custom additional functionality and/or automatic geometry generation, commonly referred to as *macros*, can be implemented easily via authoring script files in the Ruby programming language.

A custom Ruby script for modelling raindrops

The methods described previously in this section for generating the DRV describing the raindrop size distribution, discretising the raindrop geometry, and collision detection during meshing, were combined to form a custom script to complete the rain-field modelling process with SketchUp. A flow diagram showing the steps executed by the script is shown in Fig. (4.11).

The result of executing the script is a collection of plain-text files that are used by the

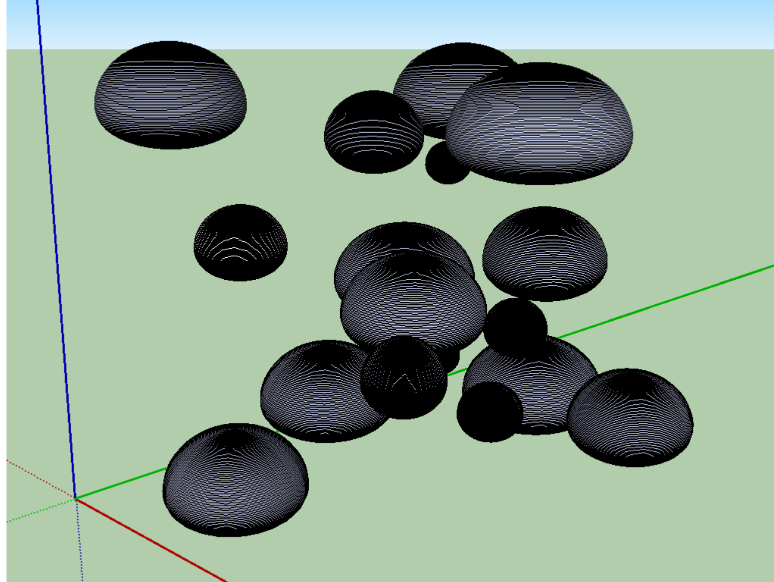


Figure 4.10: An example of several raindrops in close proximity, modelled using SketchUp.

3-D solver to initialise the environmental conditions of the TLM node mesh, as well as import the raindrop geometry information into the mesh itself. The files generated by the SketchUp script are as follows:

- **config.txt** - This file contains the physical dimensions (in metres) of the simulation environment, along each axis. It also contains the value to use for $\Delta\ell$, the equivalent size of each individual node within the mesh, as well as the number of raindrops, N_d present within the simulated volume.
- **proto[0-8].txt** - These files contain the geometry information for each of the nine prototype raindrop models, discretised at the selected $\Delta\ell$ resolution. Storing the geometry information in these files avoids several duplicate geometry datasets that would otherwise by stored during execution of the script.
- **drop[0-($N_d - 1$)].txt** - These files contain the positional information for the centre-point of each raindrop within the mesh, listed in terms of the physical centre-point location (in metres) relative to the origin point of the mesh.

An example of the layout of each of these files is shown below:

```
//config.txt
0.02                    //X-axis mesh dimension (m)
0.02                    //Y-axis mesh dimension (m)
```

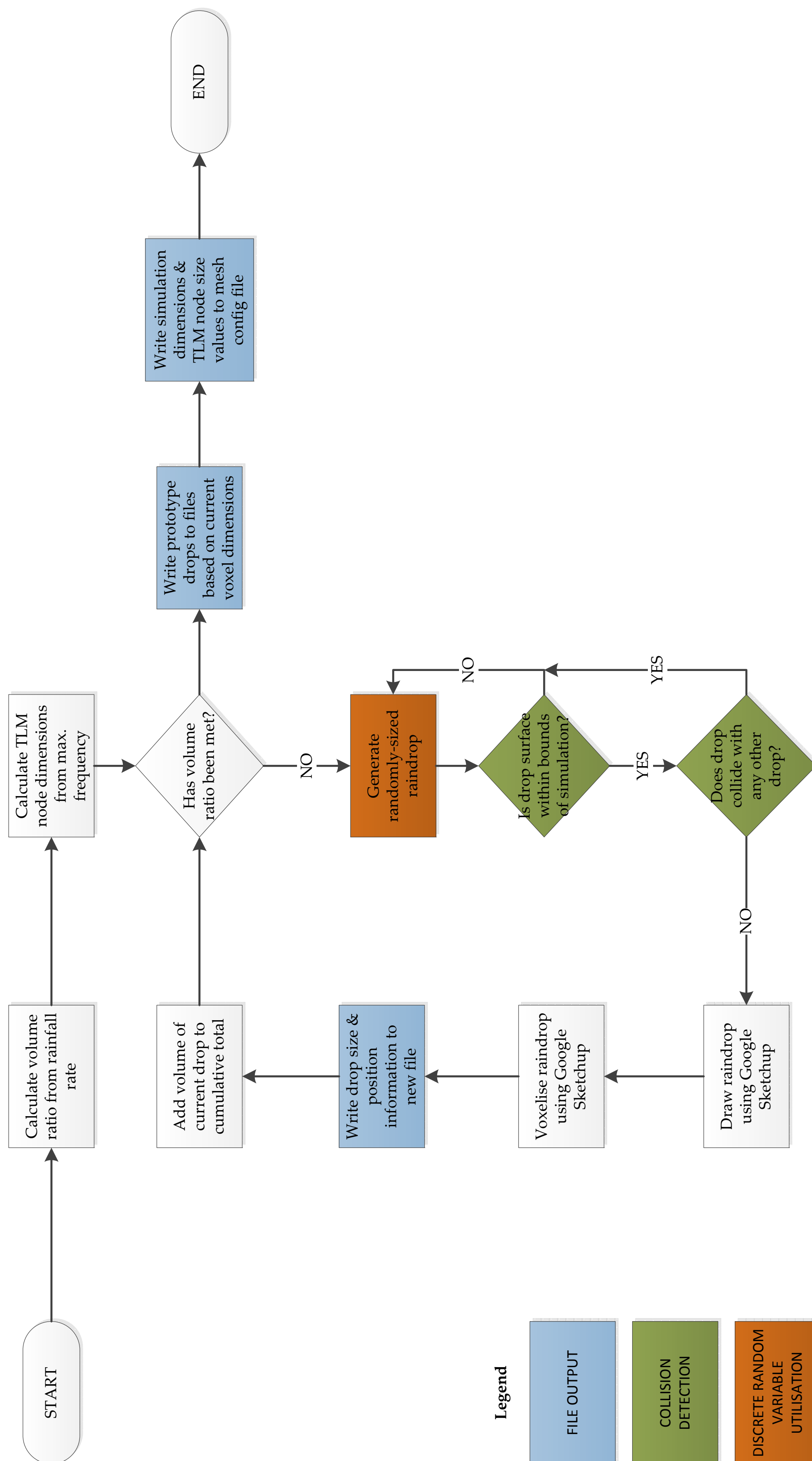


Figure 4.11: A flow diagram of the steps involved in meshing a randomised rain-field from within SketchUp.

```
0.02                        //Z-axis mesh dimension (m)
0.000100023323669017        //Cubic node side length, dL (m)
18                          //Number of raindrops to model

//proto0.txt
2.0                         //Diameter (mm)
PROTOTYPE                   //Drop-type flag
2                           //Number of nodes along X-axis
2                           //Number of nodes along Y-axis
2                           //Number of nodes along Z-axis

0                           //Raindrop mesh information
1                           //('0' for free-space, '1' for water)
1
0
1
0
1
1

//drop0.txt
3.5                         //Diameter (mm)
STANDARD                    //Drop-type flag
0.0134874919709937          //Raindrop centre-point, X
0.0159936447518827          //Raindrop centre-point, Y
0.00815280856082843         //Raindrop centre-point, Z
```

4.3 Conclusion

This chapter has presented a comprehensive strategy to allow for the generation of rainfields of arbitrary volumes. Based on the literature, a geometrical model for individual raindrops was selected that was shown to offer an accurate replication of real world raindrops.

Based on the selected geometrical model, a normal distribution was calculated for the available raindrop sizes within the model. A discrete random variable to allow raindrops to be generated of random diameter which would adhere to the normal

distribution calculated previously. Testing showed the DRV to be in good agreement with the theoretical result.

The free-to-use CAD modelling software SketchUp was identified for use as the integral component of the rain-field modelling procedure, due to its custom scripting ability and its ease of use. An open-source script to allow SketchUp objects to be discretised was adapted in order to convert individual raindrop models into their equivalent 3-D SCN mesh position co-ordinates.

A robust collision detection method was also demonstrated, utilising features of the geometrical raindrop model to ensure that randomly-placed raindrops would not intersect with one-another during the rain-field generation process.

More in-depth details for this script are available as a source code listing of the exact implementation of the Ruby script for SketchUp, which is included at the end of this thesis (Appendix A).

All of the above represents a method for modelling rain-fields of arbitrary volumes, where the variables for node dimensions (and therefore the desired F_{max} value) can be chosen by the user. The plain-text output files generated as a result are suitable for interpretation by a 3-D SCN solver, an implementation of which is described in the following chapter.

References

[4.1] G. C. McCormick, A. Hendry, and L. E. Allan, "Depolarization over a link due to rain: Measurements of the parameters," *Radio Science*, vol. 11, pp. 741–749, 1976.

[4.2] M. S. Pontes, "A method to estimate statistics of rain depolarization," *Annales des Télécommunications*, vol. 32, no. 11-12, pp. 372–376, 1977.

[4.3] M. M. J. L. Van de Kamp, "Depolarisation due to rain: the xpd-cpa relation," *Antennas and Propagation Society International Symposium*, vol. 1, pp. 400–403, July 1999.

[4.4] S. Okamura and T. Oguchi, "Electromagnetic wave propagation in rain and polarization effects," *Proceedings of the Japan Academy, Series B Physical and Biological Sciences*, vol. 86, pp. 539–562, June 2010.

[4.5] H. R. Pruppacher and R. L. Pitter, "A semi-empirical determination of the shape of cloud and raindrops," *Journal of the Atmos*, vol. 28, no. 1, pp. 86–94, 1971.

[4.6] K. V. Beard and C. Chuang, "A new model for the equilibrium shape of raindrops," *Journal of the Atmospheric Sciences*, vol. 44, no. 11, pp. 1509–1524, 1987.

[4.7] K. V. Beard, V. N. Bringi, and M. Thurai, "A new understanding of raindrop shape," *Atmospheric Research*, vol. 97, no. 4, pp. 396–415, 2010.

[4.8] E. Gorgucci, L. Baldini, and V. Chandrasekar, "Retrieval of raindrop shape-size relation using dual polarization radar measurements," *Geoscience and Remote Sensing Symposium*, pp. 4134–4137, July 2010.

[4.9] "SketchUp 2013." `http://www.sketchup.com/`.

[4.10] "Ruby programming lanuage (version 2.1.0)." `https://www.ruby-lang.org/en/`.

[4.11] G. Marsaglia, "Xorshift RNGs," *Journal of st*, vol. 8, no. 14, pp. 1–6, 2003.

[4.12] G. Marsaglia, "Generating discrete random variables in a computer," *Communications of the ACM*, vol. 6, no. 1, pp. 37–38, 1963.

[4.13] G. Marsaglia, W. W. Tsang, and J. Wang, "Fast generation of discrete random variables," *Journal of statistical software*, vol. 11, no. 3, pp. 1–11, 2004.

[4.14] A. Scullion, "Voxelize: Google Sketchup script for discretising 3-D objects," 2011.

Chapter 5

TLM on large-scale computing platforms

5.1 Introduction

This chapter presents an implementation of a 3-D SCN solver, with specialisations for the modelling of rain-fields using parallel computing to optimise simulation performance. The solver represents an integration of the Beard and Chuang raindrop geometry model [5.1, 5.2], as well as the techniques for randomly modelling a rain-field as described previously in section (4.2). Based on the investigations into parallel processing strategies, a hybrid approach utilising MPI and OpenMP has been taken in order to maximise the simulation performance. This strategy allows simulations to be targeted at EHF frequencies and above whilst retaining relatively short real-world simulation times on most platforms.

Included in this chapter are implementation details, which are given at both a high-level as well as a lower-level discussion of portions of the application relevant to meshing drops, parallelisation techniques and analysis of simulation data. The solver was validated by executing simulations with known empirical results, as well as by validating against previous studies into EHF interactions with rainfall. Finally, an analysis of simulation performance across different hardware configurations was undertaken and is also included.

5.2 Large-scale computing limitations & considerations

As discussed in previous sections of this thesis, the TLM method is inherently suited to parallel processing approaches due to data independence across the mesh within the confines of a single time-step. This section aims to discuss each of the methods considered for implementing parallel-processing within the solver, giving a summary of the advantages and countering with considerations to be made for each method.

It is clear that the choices made regarding the underlying hardware methods for enabling parallel-processing of the TLM algorithm will have a large overall impact on the processing performance of the solver. Below is a summary of the options that were considered.

5.2.1 General purpose computing on graphics processing units

It is possible to utilise graphics processing units (GPUs) for computations that are conventionally handled by the central processing unit (CPU) of a computer. General-purpose computing on GPUs (GPGPU) takes advantage of the fact that GPUs have a massively parallel hardware architecture, allowing several values held in memory to be modified simultaneously by the same mathematical operation. The use of GPUs for a parallel computing application is restricted to the stream processing[1] class of problems, due to this same hardware architecture. In the case of the TLM method, this does not necessarily present a problem, depending on the design of the solver in software.

However, obtaining the best performance for a TLM solver depends (in large part) on storing simulation mesh data inside RAM. This is due to the large amount of repetitive read and write operations which take place during the scatter and connect process, where involving additional transient caching of the mesh data to other media such as harddrives would prove too costly to simulation performance. It should be noted that whilst GPUs possess very fast, low-latency RAM which is used during calculations, it is also finite and not expandable by end users. For the proposed use-case, this would limit the size of the simulation mesh that could be held on any one machine.

[1]Stream processing is the act of performing the same operation on a set of values held in memory.

5.2.2 Single instruction, multiple data (SIMD)

SIMD architecture is a class of parallel processing whereby multiple processing elements execute the same operation on multiple data values simultaneously. Many modern CPUs contain hardware support for SIMD operations. This is a similar architecture to GPGPU, however SIMD is less restricted in the applications that it can be utilised with due to the fact it does not depend on the RAM configurations typical to GPUs. Using SIMD, parallelisation can occur at both the computation and memory I/O stages of an application. For example, assuming the data to be operated on is contiguous in memory, several values can be loaded into the SIMD arithmetic blocks at once before also being operated on simultaneously during calculations. SIMD is often used in multimedia applications such as image manipulation or digital signal processing (DSP) applications, where the same operation is taking place on all elements, and can happen simultaneously and in any order.

5.2.3 High-performance computing (HPC) clusters

A HPC cluster is the approach of utilising multiple processing units connected together across a network in order to work on a large problem by dividing it across all the processing units, each operating on a subset of the problem data. HPC cluster architecture usually assumes one of two architectures for dividing the processing. The first takes a distributed computing model, where several heterogeneous computers of varying specification are connected across a network (e.g. the internet). A central server then distributes packets of data for each client machine to operate on, utilising some or all of the processing time available on each client to complete the processing. The client machines then communicate the results back to the server for the overall result to be generated from the collated data.

Another approach to HPC clusters is to utilise a collection of homogeneous processing units located in close physical proximity, connected together via a specialised high-speed network in a cluster configuration. This reduces communication latency between processing units, and is more ideal for applications requiring a large amount of data to be stored as part of their computational processes. Commercial HPC cluster systems

often use this approach due to the increased data security as computational data is only broadcast across a local network during the simulation, as well as the improved stability of hosting several homogeneous processing units in the same location.

This strategy is certainly more flexible than the use of GPGPU computing when considering simulations requiring a large quantity of RAM in order to contain the entire mesh during program execution. Clusters can be configured to use as many compute nodes as needed and they can also run several different simulations or "jobs" simultaneously by sharing resources, as required by the end users.

5.2.4 Symmetric multiprocessing (SMP) hardware

A SMP hardware architecture is a parallel computing approach where two or more processors are interfaced with a single shared memory pool. Modern multi-core CPUs are an example of this architecture, where each core is a discrete processor having full I/O access to connected devices and is controlled by the same OS instance. SMP systems are able to execute different applications and tasks on each processor, and share a RAM memory pool whilst being interconnected via a system bus. It is also possible to have an SMP system execute the same application on each processor simultaneously. In this case, if the application has high levels of data independence, the performance of the application can be increased by dividing processing work across the available processors, each of which modifies the application data which is stored in the shared memory pool of the SMP system. However, in modern high-performance parallel computing, SMP systems are less common than HPC or GPGPU systems.

5.2.5 Symmetric multiprocessing (SMP) software

In order for a system using a SMP architecture to be taken advantage of when executing an application, it is necessary to utilise some form of SMP application programming interface (API) in order to maximise parallel computations within the application. An example use of such APIs allows global variables such as loop iteration counters to be properly managed across multiple processors. This is achieved by ensuring each instance of the application makes private copies of such variables to ensure loop

iterations are divided as equally as possible amongst the processors and that the same loop iterations are not computed twice. The result is that processing performance is improved by dividing these loop iterations across all the available processors on the SMP system.

5.2.6 Message passing interface (MPI)

As described previously in Section (3.2.3), the MPI programming standard allows for parallel computation across multiple processing cores or compute nodes via the exchange of relevant program data between the entities carrying out computations. These exchanges can occur locally, or across a network.

Two major considerations when implementing parallel processing schemes using MPI are deadlock[2] and load balancing[3].

In the context of a 3-D TLM solver, implementation issues relating to deadlock can be avoided by restricting any MPI communications between processing units to a single axis of the mesh. This significantly reduces implementation complexity and also has the added benefit of improving the compute to I/O ratio for each sub-mesh within the application. In the context of the proposed use-case of rain-field modelling, objects in mesh are distributed randomly, as detailed previously. This should result in computation times for each sub-mesh for each individual time-step that are approximately equal (assuming processing units of equal performance). This in turn should provide implicit load balancing within a 3-D TLM solver, and processing unit utilisation should therefore be maximised. When simulating enclosed metallic objects (especially large objects unevenly distributed in a simulation volume), computation times become more varied since nodes within enclosed metallic objects do not require scatter and connecting. Therefore load balancing becomes harder to manage in this case, and it would be pertinent to implement some form of dynamic sub-mesh resizing scheme for a parallel 3-D TLM solver in this case.

[2]Deadlock is the term for when a process becomes blocked waiting for a condition that will never become true.[5.3][5.4]

[3]Load balancing is the practice of ensuring that each processing unit within a parallel application is being utilised to similar levels. This ensures idle time waiting for other processes to finish computation is minimised.

5.3 Methodology

The design and development of the 3-D SCN solver represents a software application of non-trivial size and complexity. In order to discuss the operation of the solver, this section breaks down the application in terms of the different functionality that is present. An overall theory of operation is given, describing the solver operations on a high-level. This is then followed by a more in-depth discussion of specific portions of the software which manage parallel execution of the TLM algorithm, as well as analysing and visualising simulation data.

5.3.1 Theory of operation

The operation of the 3-D solver application can be divided in to three major sections:

1. Prototype drop meshing.
2. Simulated drop meshing.
3. Simulation execution.

An overview of the application execution for the drop meshing is shown in Fig. (5.1), and the methods for conducting both the prototype and simulated drop meshing are detailed below.

Prototype drop meshing

As discussed previously in Section (4.2.1), the Beard & Chuang raindrop model provides a set of geometry models at fixed dimensions. Therefore, it is possible to optimise the performance of the *meshing* stage of the execution of a simulation using the solver by producing a static set of *prototype* raindrop geometry models which can be copied into the mesh at the locations dictated by the configuration files output by the Sketchup script. This is possible since the generated raindrop models cannot vary continuously in size.

The prototype drop geometry information for each drop is held in an array which is populated with relative permittivity information for each node within the mesh region

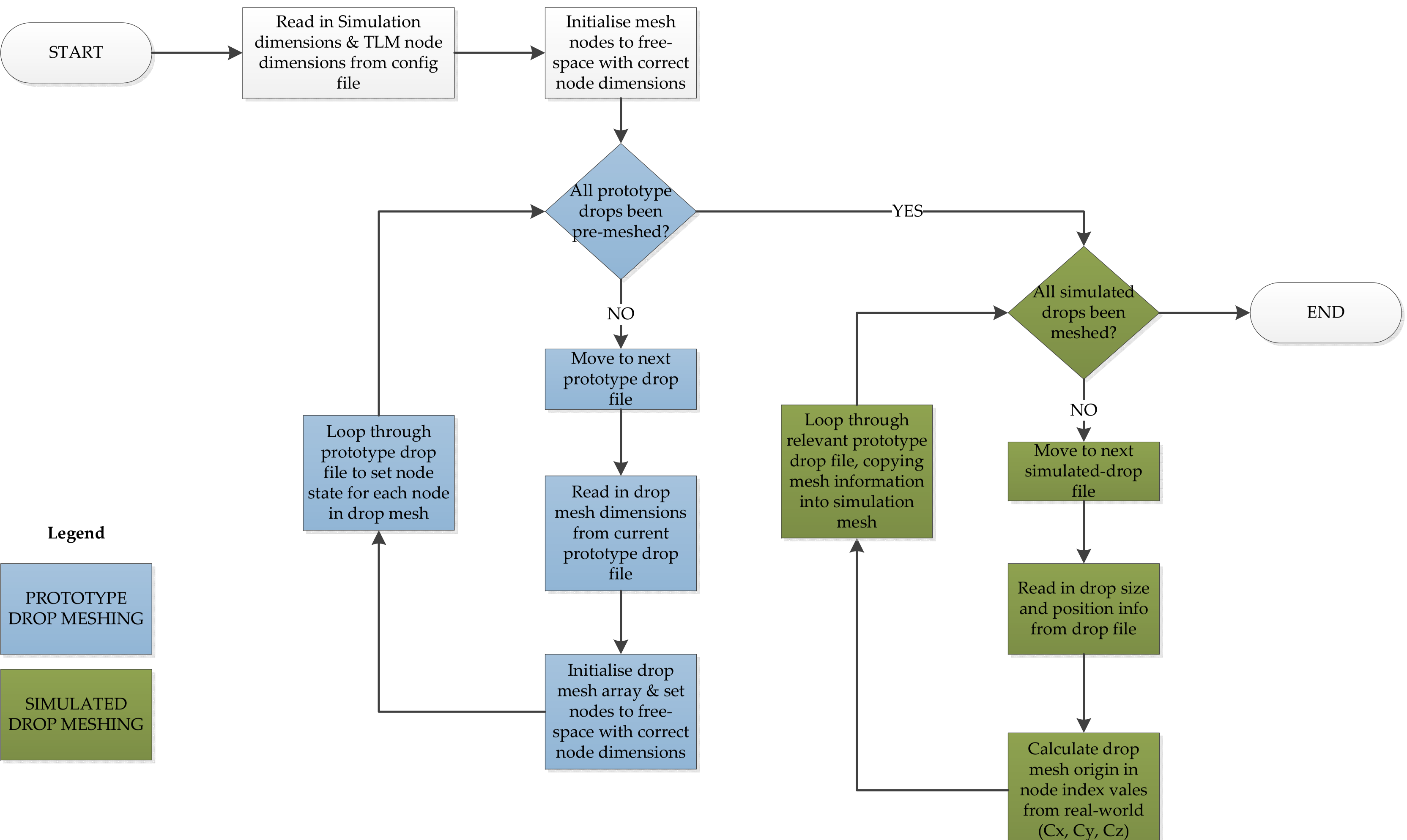


Figure 5.1: A flow diagram of the steps involved in importing the randomised raindrop field mesh information, ready for the main application to execute the simulation.

encompassing the drop. This is read in from the series of prototype drop plain-text configuration files generated by the Sketchup script. The layout and quantity of this per-node information is directly dependent on the required cut-off frequency of the simulation, which directly impacts the $\Delta\ell$ value, and in turn the mesh resolution.

Simulated drop meshing

Once the prototype drop files have been imported and converted to node information data, the solver is able to import the series of drops that must be placed inside the mesh volume. As shown in Section (4.2.6), the drop configuration files contain information on which drop size to use, as well as the absolute location of the centre of the drop within the simulated environment. Using this information, the solver selects the appropriate prototype drop to use. From this, the per-node dielectric properties are copied from the prototype drop array into the mesh at the appropriate position.

5.3.2 Compute-node boundary data exchange

As discussed previously in Section (5.3.1), the SCN mesh representing the simulated environment is divided between compute-nodes in order to take advantage of parallel computing, reducing the overall simulation time. However, it is necessary to implement a method for transferring the relevant node port voltage values across the boundary interfaces of these sub-meshes once per simulated time-step. This data exchange between sub-meshes then becomes equivalent to a single, contiguous mesh volume present on a single compute-node.

The specific port voltage values that are exchanged across a sub-mesh boundary is dependent on the axis on which the boundary interface lies. The ports used in each interface orientation can be determined easily from examining the topology of a SCN, as shown previously in Fig. (2.11).

The time taken to compute all node port values for a given time-step within a given sub-mesh will be influenced by the total number of nodes in the sub-mesh in comparison to the number of nodes in total on any boundary interfaces with neighbouring sub-meshes. This is known as the *compute-to-I/O ratio*. The number of nodes present on

boundary interfaces will in turn be influenced by the topology of the compute-node interconnections, and the geometry of the sub-meshes themselves. If some sub-meshes communicate on fewer boundary faces than others, it is possible for sub-meshes node calculations for a particular time-step to complete in slightly different amounts of time, even in the case of an uniform or empty mesh volume.

Assuming a cuboid sub-mesh configuration, with the sub-meshes themselves laid out as a cube, the boundary interfaces between neighbouring sub-meshes will occur in the X, Y and Z axes. However, the sub-meshes that are present adjacent to external mesh boundary faces, edges and vertices will not communicate on each of their six faces. This will influence the compute-to-I/O ratio as discussed previously. In this case, a set of expressions for calculating the number of sub-meshes in each category (internal, face adjacent, edge, and vertex) is shown in (5.1).

$$\begin{aligned} N_i &= (n-2)^3 \\ N_f &= 6(n-2)^2 \\ N_e &= 12(n-2) \\ N_v &= 8 \end{aligned} \tag{5.1}$$

The data exchange of port voltage values between sub-mesh boundary interfaces can be thought of as an extension of the similar operations that occur in the *connect* phase of the TLM algorithm. This allows the individual sub-meshes across several compute-nodes to behave as a single mesh volume. The MPI library is used in order to exchange the relevant port values across the sub-mesh boundary interfaces, in the same form as local exchanges between nodes during the the local *connect* operations. The overall process is shown below, considering sub-mesh boundary interfaces along the Y-axis only (in pseudo-code):

```
//Loop control...
int x, y, z;

//Connect temp variables...
```

```
float Vc;
SCNNode *nA, *nB;

//MPI START!
//MPI buffering for Tx
for (z = 0; z < ZDIM; z++) {
        for (x = 0; x < XDIM; x++) {
                //Node at X = x, Y = YDIM-1, Z = 1...
                nA = &mesh[(x*YDIM+(YDIM-1))*ZDIM+z];
                yTxNorthBuffer[(z*XDIM+x)*2+0] = nA->Vypx;
                yTxNorthBuffer[(z*XDIM+x)*2+1] = nA->Vypz;

                //Node at X = x, Y = 0, Z = 1...
                nA = &mesh[(x*YDIM+0)*ZDIM+z];
                yTxSouthBuffer[(z*XDIM+x)*2+0] = nA->Vynx;
                yTxSouthBuffer[(z*XDIM+x)*2+1] = nA->Vynz;
        }
}

//Start Persistent Communication - (MPI requests already initialised to
    reduce overhead)...
MPI_Startall(2, reqSend);
MPI_Startall(2, reqRecv);

//Connect locally...
#pragma omp parallel for private (y,x,z,Vc,nA,nB)
for (y = 0; y < YDIM; y++)
{
        for (x = 0; x < XDIM; x++)
        {
                for (z = 0; z < ZDIM; z++)
                {
                        //Current node...
                        nA = &mesh[(x*YDIM+y)*ZDIM+z];
                        //x
                        nB = &mesh[((x+1)*YDIM+y)*ZDIM+z];
                        Vc = nA->Vxpy;
                        nA->Vxpy = nB->Vxny;
                        nB->Vxny = Vc;
```

```
                        Vc = nA->Vxpz;
                        nA->Vxpz = nB->Vxnz;
                        nB->Vxnz = Vc;
                        //y
                        //(only locally connect in Y if not on a submesh
                            boundary)...
                        if (y < YDIM-1) {
                                nB = &mesh[(x*YDIM+(y+1))*ZDIM+z];
                                Vc = nA->Vypx;
                                nA->Vypx = nB->Vynx;
                                nB->Vynx = Vc;
                                Vc = nA->Vypz;
                                nA->Vypz = nB->Vynz;
                                nB->Vynz = Vc;
                        }
                        //z
                        nB = &mesh[(x*YDIM+y)*ZDIM+(z+1)];
                        Vc = nA->Vzpx;
                        nA->Vzpx = nB->Vznx;
                        nB->Vznx = Vc;
                        Vc = nA->Vzpy;
                        nA->Vzpy = nB->Vzny;
                        nB->Vzny = Vc;
                }
        }
}

//Wait until all requests have completed...
MPI_Waitall(2, reqSend, statusSend);
MPI_Waitall(2, reqRecv, statusRecv);

//Unpack MPI buffer into mesh...
for (z = 0; z < ZDIM; z++) {
        for (x = 0; x < XDIM; x++) {
                //Node at X = x, Y = 0, Z = 1...
                nA = &mesh[(x*YDIM+0)*ZDIM+z];
                nA->Vynx = yRxSouthBuffer[(z*XDIM+x)*2+0];
                nA->Vynz = yRxSouthBuffer[(z*XDIM+x)*2+1];
                //Node at X = x, Y = YDIM-1, Z = 1...
```

```
                nA = &mesh[(x*YDIM+(YDIM-1))*ZDIM+z];
                nA->Vypx = yRxNorthBuffer[(z*XDIM+x)*2+0];
                nA->Vypz = yRxNorthBuffer[(z*XDIM+x)*2+1];
        }
}
```

5.3.3 SMP acceleration strategy

Modern desktop and HPC cluster systems often utilise multi-core processors in their hardware architecture. By exploiting multiple processors on a system, additional performance gains can be achieved in software applications already designed for parallel computations using other schemes such as MPI, described in the previous Section. OpenMP was previously identified as a suitable candidate for exploiting multiple physical processing cores, as shown in Sections (3.1.2) and (3.1.2).

To reiterate; OpenMP is a SMP API that supports shared-memory multiprocessing on a number of platforms. During program execution, preprocessor directives before functions or sections of the code intended to be run in parallel instruct the creation of slave threads forked off of the master thread. This allows the parallel execution of that section of the application across the number of physical cores on the system by spawning slave threads equal to the number of cores.[4]

In the context of the 3-D desktop TLM solver presented, several computationally time intensive loops can be parallelised using the `#pragma omp parallel for` preprocessor directive. This is simply placed above the loop to be parallelised, and the loop iterations to be executed are divided between the slave threads that are spawned by the compiler.

Performance gains from utilising OpenMP vary, with the most noticeable increases seen with highly computationally-intensive operations executing on systems with several physical cores.

[4]Sometimes, the number of slave threads generated can be greater than the number of physical cores present on the system if the CPU supports multi-threading.

5.3.4 Extracting simulation data

During the execution of the solver, one or more mechanisms for viewing the EM field information are necessary in order to analyse the behaviour of any simulations that are executed. This access to the mesh data can typically be categorised into either informal or formal observations, depending on the requirements of the user.

In the informal observation case, the EM field information can be presented as an intensity colour map on a 2-D image. This allows for generalised observations of mesh behaviour, where knowledge of specific field component values is not required. In the formal observation case, individual nodes can be observed, and the EM field component values are outputted to a plain-text file after each time-step for post-simulation analysis.

The following sections describe both observation cases in greater detail.

Visualising the mesh

An informal analysis of the EM field within the mesh during the simulation is achieved by generating a series of 2-D images representing the mesh. Firstly, images are generated in the XY plane (Fig. (5.2) (a)), each representing a one node thick slice through the mesh along the Z-axis (Fig. (5.2) (b)). This image set is generated for each time-step, allowing analysis of wave propagation throughout the mesh for the duration of any simulation that is executed (Fig. (5.2) (c)).

The EM field information is represented in these images by mapping the field component chosen for visualisation from each node to an equivalent colour using a RGB colour map algorithm, which gives an *at-a-glance* visual indication of wave behaviour and intensity to the user examining the resulting image files. In order to improve performance of the visualisation functionality, the colour map values are pre-calculated at the start of simulation execution, and stored in a look-up table. The colour map algorithm, assuming 8-bits per channel is shown in (5.2):

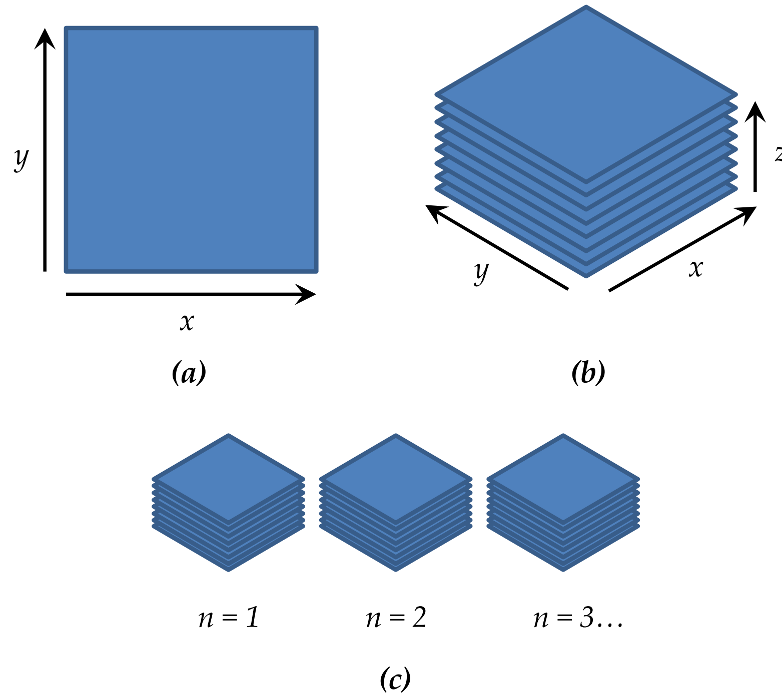


Figure 5.2: An overview of the mesh visualisation scheme, where individual slices through a 3-D mesh ***(a)*** are converted into a multi-page image ***(b)*** in order to represent the mesh at each time-step ***(c)***. The mesh dimensions are given by x, y and z and the time-step number is given by n.

$$
\begin{aligned}
p_r &= \frac{256 \times (109 - (219 - (i \times 256)))}{73} \\
p_g &= \frac{256 \times (109 - (146 - (i \times 256)))}{73} \\
p_b &= \frac{256 \times (109 - (73 - (i \times 256)))}{73}
\end{aligned}
\tag{5.2}
$$

where i is a EM field component value normalised to be between 0.0 - 1.0, and p_r, p_g, and p_b are the RGB pixel component values. A visualisation of the resulting colour map is shown in Fig. (5.3).

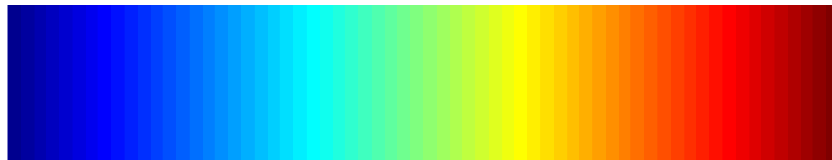


Figure 5.3: A visualisation of the colour map that results from the calculations shown in (5.2).

During image generation, boundary nodes are represented as entirely white pixels (i.e.

a RGB value of 255, 255, 255). Also, any nodes with dielectric properties other than free-space have their RGB values offset by +64, +64, +64, which allows objects within the mesh to be visible in the generated images as well as the wave information.

To compile the contents of each image slice, MPI is exploited once more to improve the performance of the image output scheme. Initially, the fact that the mesh is divided between different processing units could be viewed as a complication where generation of an image representing the mesh contents is concerned. This is because each image should represent the entire mesh as a whole, and not individual sub-meshes.

One method for collating the mesh data would be to transmit the contents of each submesh-slice to a single processing unit, before outputting the entire mesh-slice to an image. An obvious disadvantage to this method is that large amounts of data would be transmitted, reducing the performance of the image output function. However, this potential problem can be avoided by utilising the parallel file I/O functionality of MPI, which allows data from each sub-mesh to be written to one image file at the same time, forming a representation of the mesh as a whole. This also has the added benefit of reducing the time taken to generate images, and removes the need to transmit all sub-mesh data to a single processing unit before generating the image file.

MPI is able to write to the same file across multiple processing units by specifying a offset for the start-position for the write operation within the memory allocated for the image file. This allows multiple write operations to occur simultaneously without resulting in corrupted memory. This process is shown in Fig. (5.4).

As shown in the above figure, calculating the value for the offset variable is trivial, and is related to the X and Y dimensions of the submesh, as well as the processing unit ID number, r. It should also be noted that a full-colour image is generated hence the '3' in the calculation, as one value is stored for each channel in the RGB colourspace.

Observing single nodes

In order to obtain a more accurate analysis of the simulation behaviour, precise EM field component information must be obtained on an individual node basis from within the mesh. This allows for the frequency response of the simulation environment to be

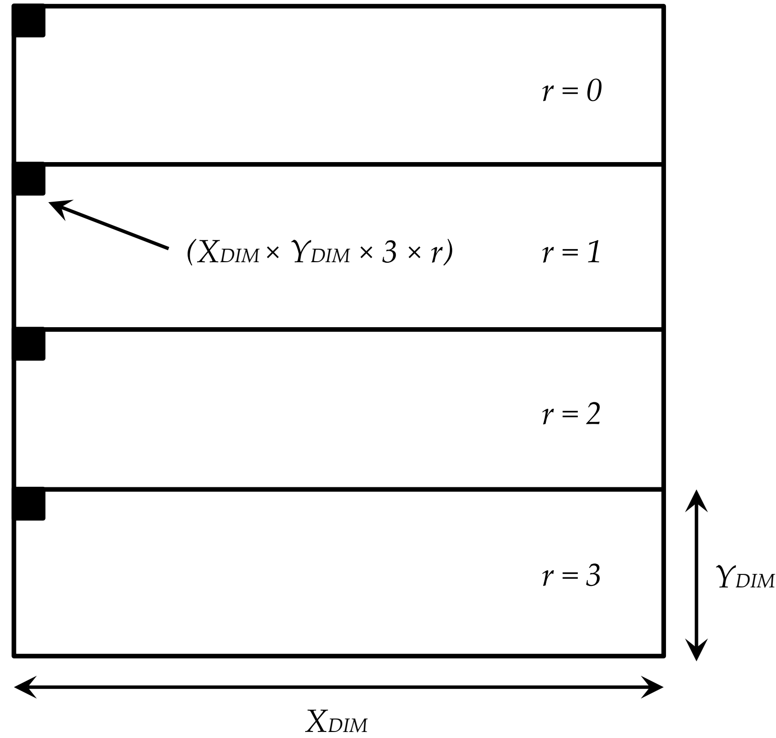


Figure 5.4: An illustration of how MPI is used to improve performance via parallel file I/O, in an example system with four processing units. X_{DIM} and Y_{DIM} are sub-mesh dimensions and r is the processing unit number. The black squares represent file pointers which move concurrently as each processing unit writes to the file in parallel.

computed throughout the execution of the simulation.

A `MeasurementPoint` class was created in order to fulfil this functionality. The frequency response at the node location that is observed is calculated by computing a Discrete Fourier Transform (DFT) at every time-step. The DFT algorithm can be adapted for this case, where the number of samples is not fixed, and in fact increases by one sample after every time-step. This modification to the DFT equation is shown in (5.3).

$$X(n) = \sum_{n=0}^{N-1} x(n) e^{-j2\pi f n \Delta t} \tag{5.3}$$

where $X(n)$ is the DFT of the original signal $x(n)$ at frequency f and time-step n, with a time-step of Δt.

For the 3-D solver application, the `MeasurementPoint` class is specified to have a start and stop frequency, as well as a frequency step value. This means that several DFTs

for frequencies of interest can be calculated during the simulation run. A `<vector>` array of complex number values stores the running-total DFTs for each frequency, and these are updated every time-step during the simulation execution. It is possible to optimise the DFT equation shown above in (5.3) to reduce the number of complex exponent calculations that are executed during a given time-step. The equation can be re-expressed to group together the values which remain constant for a given time-step within a simulation, meaning a single complex exponent can be pre-calculated rather than computing one for each frequency of interest. This is shown in (5.4).

$$X(n) = \sum_{n=0}^{N-1} x(n) W^{f} \tag{5.4}$$

where $W = e^{-j2\pi n \Delta t}$, i.e. the constant-value complex exponential term for the current time-step.

In the context of the 3-D solver, the $x(n)$ term is represented by one or more field component values that are of interest when computing the DFT values. That is to say that any single field component may be observed, or a vector representing the total E or H-field. If the overall E or H-field is to be observed, the vector values representing these fields can be calculated easily in the form shown in (5.5).

$$\mathbf{F} = \sqrt{{F_x}^2 + {F_y}^2 + {F_z}^2} \tag{5.5}$$

where **F** represents either the overall E or H-field vector, and F_x, F_y and F_z represent the individual E or H-field components directed along each respective axis.

5.3.5 Summary

This section has described and discussed the methodology and implementation details of a parallel computing optimised 3-D SCN TLM solver for desktop and HPC cluster systems at a level appropriate for most readers. More in-depth details are available within documentation that comprehensively details the exact software implementation, which is included at the end of this thesis (Appendix B).

5.4 Results

Implementation details for a 3-D SCN solver with parallel computing optimisations have been discussed. Suitable methods for extracting simulation data have also been proposed, again with optimisations for a parallel computing architecture. This section presents the results of several example simulation environments designed to test and verify the different elements of what makes a successful CEM solver application. An in-depth analysis is provided in each case, and findings are summarised at the end of the section.

5.4.1 Validation

For any CEM solver application, it is crucial that the implemented solver is working correctly during the course of executing simulations. In the case of the TLM method, it is a trivial process to ensure that node port voltages are correct. A subset of nodes in a small scale simulation are observed, and when combined with an impulsive excitation the node port voltages are recorded over a small number of time-steps and checked against the expected results via manual calculations. Whilst this method is indeed useful to minimise any potential software defects in the early stages of implementation, other methods should be employed on the macro scale to ensure the accuracy of simulation results.

Time-domain verification

The method previously presented for outputting PPM image files in order to represent time-domain data, in section (5.3.4) is also a useful approach to a second-stage verification of the simulation accuracy of the 3-D TLM solver. Any mistakes in the implementation of the scatter and/or connect functionality of the solver will be immediately evident during examination of time-domain image outputs, even over the course of a relatively small number of time-steps. Typically, software defects relating to the node stub value calculations and/or time-step calculations can present themselves as a continued increasing in overall mesh energy, indicating that the mesh becomes

unstable. Software defects within the connect process can present as unsymmetrical wave propagation behaviour, and problems with boundary condition implementations are also easily diagnosed via visual inspection of the time-domain PPM image files.

Frequency-domain verification

The final stage in the verification of the 3-D TLM solver is to ensure that frequency-domain results are sufficiently accurate to be within a suitable margin of error. A first step is to implement a canonical problem within the solver, for which empirical results can be calculated. A popular example is a cuboid cavity modelled with metallic walls and dimensions of 1 m × 1 m × 1 m. Any lossless cavity such as this example will be strongly resonant at specific frequencies, whilst being mostly unresponsive at others. The frequencies at which a cavity resonates (also known as *modes*) are dependant on its dimensions along each axis. An equation for calculating transverse electric (TE) modes of such cavities is shown in (5.6) [5.5].

$$TE_{mnp} = \frac{c}{2\sqrt{\epsilon_r \mu_r}} \left(\sqrt{\frac{m}{dim_x}^2 + \frac{n}{dim_y}^2 + \frac{p}{dim_z}^2} \right) \tag{5.6}$$

where TE_{mnp} is the relevant TE mode frequency and m, n and p are the number of standing waves present along the x, y and z axes inside the cavity, representing the specific mode configuration. The dimensions of the cavity along each axis are given by dim_x, dim_y, and dim_z respectively. For a 1 m^3 resonant cavity in a free-space environment, the TE mode equation can be simplified to that shown in (5.7).

$$TE_{mnp} = \frac{c}{2} \left(\sqrt{m^2 + n^2 + p^2} \right) \tag{5.7}$$

The fundamental TE mode, TE_{110} for a 1 m^3 resonant cavity is therefore calculated to be 211.985 MHz. As stated previously, this provides a target known result to use when executing the resonant cavity simulation on the 3-D TLM solver. Depending on the frequency-domain analysis parameters that are chosen, other TE modes such as TE_{220} = 423.971 MHz may also be observed.

The 3-D TLM solver was validated against this example of a 1 m^3 resonant cavity where

$\Delta\ell = 0.01$ m. The resulting mesh dimensions were therefore $1000 \times 1000 \times 1000$ nodes, and the simulation was executed for 40 000 time-steps (equivalent to a real-world time of 6.673×10^{-7} s). This experimental configuration is similar to approaches utilised previously by Herring and Flint. [5.6, 5.7]. The results of simulation are shown in Fig. (5.5).

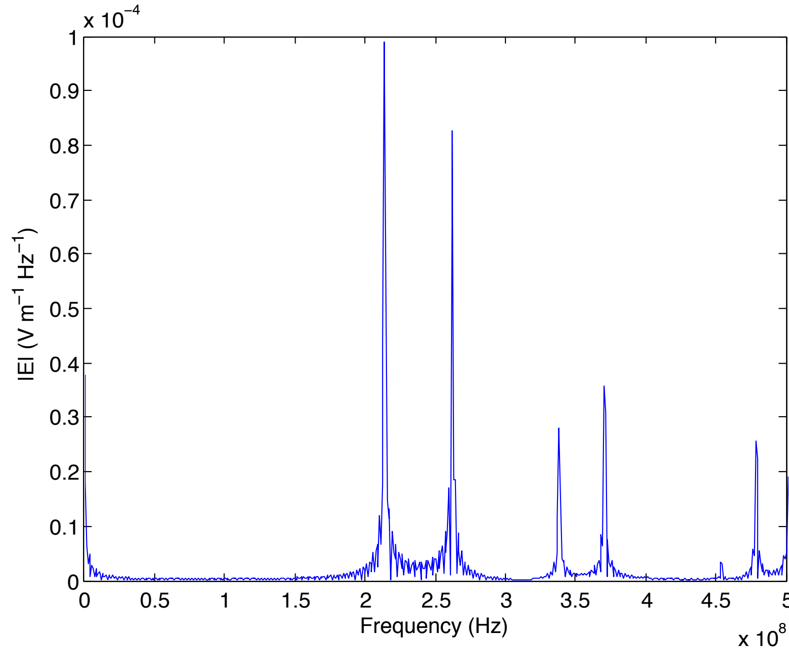


Figure 5.5: Frequency domain results showing resonant frequencies in a simulated 1 m^3 cavity.

In the above figure the TE_{110} mode is the most strongly excited, as is expected due to the fact that it is a fundamental TE mode. Examining the results shows a frequency of 214 MHz, an error of +0.95% from the calculated result. The amount of error in the resolving of frequency responses when using DFT or FFT algorithms is dictated by the intervals in the frequency domain between sampling points. This is turn is governed by a combination of the time-step size and number of samples in the time domain. The number of frequency domain samples, commonly referred to as *bins*, grows as their interval narrows. An equation for the frequency domain interval, Δf is shown in (5.8).

$$\Delta f = \frac{f_c}{N} \tag{5.8}$$

where f_c is the simulation cut-off frequency (i.e. $\frac{1}{\Delta t}$) and N is the number of samples in the time domain. It should be noted that in the case of using the FFT algorithm to

compute the frequency domain result, if N is not a power of 2, the time domain signal is padded with zeros until the signal is the next highest power of 2 in length. This is necessary for proper calculation using FFT.

5.4.2 Benchmarking

The benchmarking problem

In contemporary CEM applications, it is often the case that researchers require solver applications that are able to model increasingly large volumes (for example when modelling large anechoic chambers) and/or small geometry at increasingly high mesh resolutions (for example, in the case of small antenna designs with intricate geometry). For these and other applications requiring a large number of mesh nodes to be present in the simulation volume, it is clear that a CEM solver that is optimised for parallel computing architectures should be able to outperform a comparable solver running on fewer processing units. This is advantageous to researchers aiming to complete simulations as quickly as possible in terms of their wall-clock time.

Following the development of the rain-field modelling method presented in the previous chapter, a trivial example problem was devised based on a approximation of a rain-field, to be utilised during the benchmarking tests detailed below. A rain-field of dimensions 20 mm $\times$ 20 mm $\times$ 20 mm was simulated at a mesh resolution of 0.1 mm.node^{-1}, for a total of $8{\times}10^6$ nodes. This was filled with modelled raindrops using the methods detailed in the previous chapter, until the total raindrop volume was 10% of that of the overall simulation volume. A step impulse of 1 V was injected into the mesh at time-step 0, and the simulation was executed for 1 000 time-steps. The resulting real-world simulation time was equivalent to $1.65{\times}10^{-10}$ s.

Results

The baseline performance figure for a CEM solver would be the *raw performance*, or the processing speed of the solver application during a simulation run without any I/O operations to storage media. This is the theoretical peak performance of the CEM solver for a given platform configuration, as real-world processing speeds will be lower than

this due to the fact that some amount of I/O is necessary in order to obtain simulation results (which is true in both the time or frequency domains). The raw performance of a CEM solver is useful however in order to determine relative performance figures when benchmarking other scenarios.

With any application that is optimised for parallel computing architectures, it is clear that the main factor that determines performance is the number of available processing units. In the context of the 3-D SCN TLM solver that has been implemented, raw performance figures were obtained for an example simulation rain-field volume when being executed on a typical desktop workstation computer[5] with varying numbers of processing cores being made available to the solver. These results are shown in Fig. (5.6).

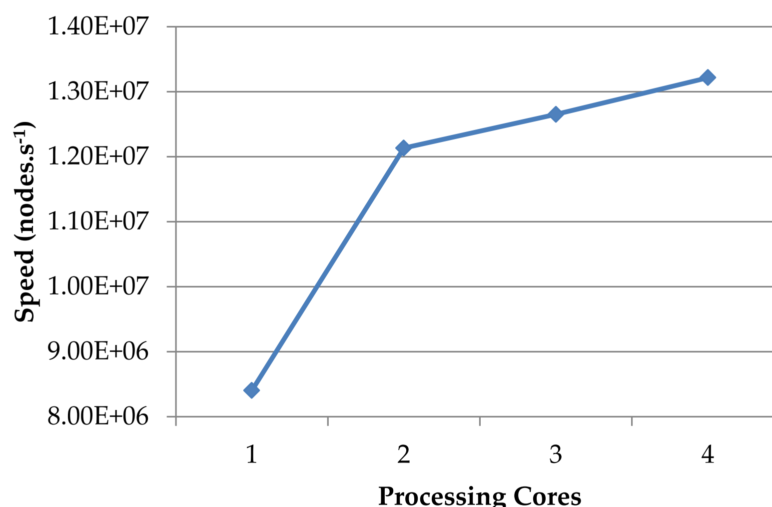


Figure 5.6: The raw performance metrics of the 3-D SCN TLM solver when running on varying numbers of processing cores with no write operations to disk. (An average of three simulation runs).

It should be noted that these figures are the real-world times that the simulations took to complete execution. In the case of CEM solvers, it is also useful to be able to demonstrate processing performance in terms of a figure in the units $nodes.s^{-1}$ as this allows a more direct comparison across different solvers and simulation volumes than the wall-clock time figures previously presented. The raw performance figures in $nodes.s^{-1}$ are shown in Fig. (5.7).

[5]In all benchmarking tests, the solver was run on a Windows 7 64-bit desktop workstation with the following specifications: Quad-core Intel i5 650 3.2GHz CPU, 12GB 1066MHz RAM, 160GB HDD.

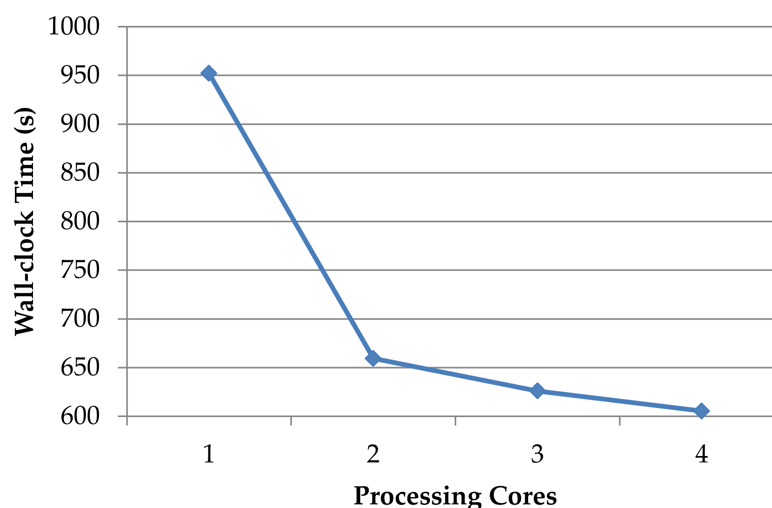


Figure 5.7: The wall-clock execution times of the 3-D SCN TLM solver when running on varying numbers of processing cores with no write operations to disk. (An average of three simulation runs).

As discussed previously in section (5.3.4), the ability to extract simulation parameters and resultant data is essential to a CEM solver application. Both time and frequency domain results can prove useful in different simulation configurations, and in both cases data output to a hard drive or other persistent storage medium will result in a performance penalty. In the time domain case this data output happens at regular intervals in the form of generated PPM image files, and the output interval width (or *stride*) will have a direct bearing on the overall processing performance of the solver in any given simulation run. The relationship between data output interval, the number of processing cores utilised and the simulation time is shown in Fig. (5.8).

5.5 Performance analysis

It is evident from the results presented in the previous section that there are clear benefits to a parallel processing approach when implementing a 3-D CEM solver application. In order to better compare the raw performance of the solver, the normalised raw performance figures of the solver are shown in Table (5.1).

When comparing the case of using two processing cores as opposed to a single processing core, there is a large relative performance increase of 44.4%. Following this in the cases of using three and four processing cores respectively, the relative solver

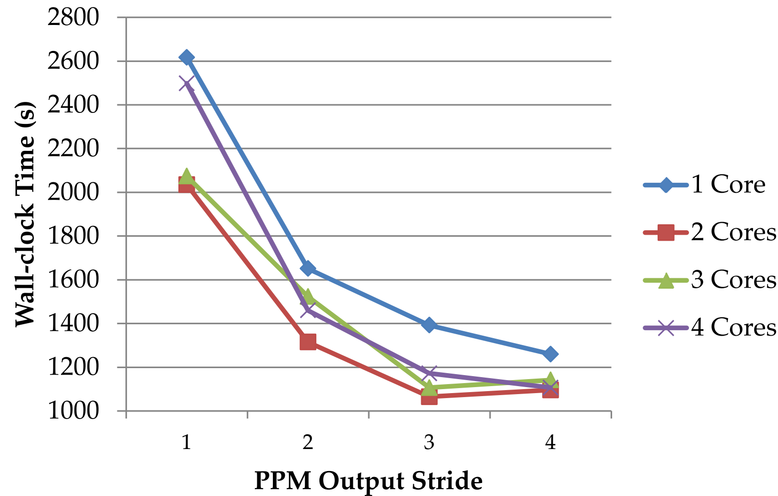


Figure 5.8: The wall-clock execution times of the 3-D SCN TLM solver when running on varying numbers of processing cores and with different stride values for the generated PPM image outputs. (An average of three simulation runs for each number of cores under test).

	Normalised Performance	
Cores	**Absolute**	**Relative**
1	1.000	1.000
2	1.444	1.444
3	1.506	1.043
4	1.573	1.045

Table 5.1: The absolute and relative normalised performance figures for the raw 3-D SCN TLM solver based on the results as shown in Fig. (5.6).

performance increase slows to between 4.3 - 4.5% in each case. This could be an indication of the principles put forward by Amdahl [5.8] relating to software applications and/or hardware systems that contain a mixture of both parallel and serial operations in systems with varying numbers of processing units. Another explanation would be a performance bottleneck due to CPU cache saturation. In this case whilst it still beneficial to increase the numbers of processing units during execution of the 3-D solver, clearly it is a case of diminishing returns as the pace of performance increase slows and becomes approximately linear.

The time-domain performance benchmarking shown previously in Fig (5.8) relates the simulation performance of the solver to the PPM image file output interval (or *stride*) as well as the number of processing cores in use. This is intended to demonstrate the performance of the solver in a scenario closer to real-world usage. As noted previously

in section (5.3.4), all PPM file write operations are being executed in parallel across all processing cores via MPI. To reiterate, the reasoning for this was two-fold in order to reduce implementation complexity (since sub-mesh image data would not need assembling into a full mesh before the file write operation) and to improve simulation performance by sharing write operations across processing cores.

It is interesting to note that the benchmarking results in this time-domain case demonstrate behaviour that is not necessarily immediately intuitive. Firstly, the wall-clock simulation time when using three or four processing cores is longer than when using just two cores (however, using a single core is the slowest as expected). This is true in every test case when varying the stride value for PPM output. This is likely a side-effect of the fact that ultimately any write operation to a hard-drive happens in a serial fashion. A unavoidable disadvantage of hard-drives is rotational latency[6]. Also, ultimately each processing core will take it in turns to write their portion of each file to disk, resulting in movements of the read/write head itself. Both of these phenomena will impair simulation performance, and illustrates that in certain configurations it is not always beneficial to utilise more processing cores when considering parallel file write operations. In the case of other persistent storage options such as solid-state disks (SSDs) that typically possess considerably faster read and write speeds, it is likely that simulation performance would be closer to that shown as the raw performance figures in Fig. (5.7). Due to the lack of mechanical parts, the behaviour when observing file write operations with multiple processing cores is also likely to differ.

Finally, in the case of all of the performance benchmarking tests executed using four processing cores will likely incurred an additional performance penalty as the test workstation possessed only a quad-core CPU. This means that performance was not likely to be at absolute peak potential due to the operating system also requiring CPU time for lower level processes (which is true even if no other user-level software is running). This provides an additional probable explanation for slower than expected wall-clock execution times and normalised performance figures in the four core test cases.

[6]Rotational latency is the delay caused by the hard-drive having to physically move the correct disk sector into place under the read/write head during its rotation. It is a function of the rotational speed of a hard-drive, and for a typical 7200pm consumer hard-drive the maximum delay is 4.17ms

It has been shown that despite the impairments due to disk contention and mechanical latencies, the use of two or more processing cores offers better overall performance characteristics with every tested stride value for PPM image outputs to disk. For real-world applications this is the most relevant performance metric, since it demonstrates that parallel computation and file write operations are indeed faster than execution on a single processing core. It is however also pertinent to characterise the solver performance in these cases relative to the demonstrated maximum raw performance figures. These normalised performance figures are shown in Fig. (5.9).

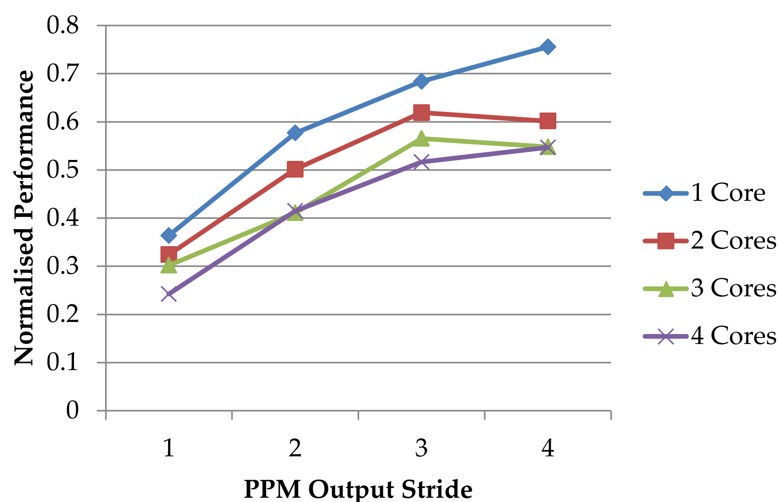


Figure 5.9: The performance figures of the 3-D SCN TLM solver when running on varying numbers of processing cores and with different stride values for the generated PPM image outputs, normalised to the raw performance figures shown in Fig. (5.7).

It is worthy noting that the normalised performance figures demonstrate that the single core test case is more efficient than when using two or more cores. This reinforces the hypothesis that parallel file write operations to different portions of a PPM image file in combination with the associated latencies of mechanical persistent storage are impairing simulation performance. Indeed, even in the case of utilising two processing cores where absolute simulation performance is shown to be the highest of all test cases, normalised performance (and therefore efficiency) is shown to be lower than in the single core case.

5.6 Summary

A 3-D SCN TLM solver application has been implemented based on a hybrid approach to parallel computing, utilising a combination of the MPI and OpenMP software libraries in order to maximise processing performance on both multi-core desktop and HPC cluster systems. These computing platforms were favoured over GPU-based approaches for their ubquity in academic and commericial environments, as well as the flexibility and expandability of available processing power when using these platforms.

A methodology for implementing the raindrop meshing functionality was proposed, based on the raindrop modelling implementation presented in the previous chapter. The discrete nature of the raindrop model (in terms of modelling raindrops of different diameters) was exploited and for any given simulation a number of *prototype raindrops* are generated by the Ruby script, which are imported and then re-used during raindrop meshing. This has a two-fold effect of reducing computational overhead during solver initialisation, as well as removing redudant data usage within the simulation configuration files.

Strategies were also presented for the exchange of the relevant node port voltage information between neighbouring processing units during solver execution, whilst avoiding the issue of deadlock.

Time and frequency-domain methods for simulation observations have been proposed, and methods discussed for utilising parallel computing architectures to also accelerate the storage of this data, with focus to the time-domain case and the output of text and image files representing the simulation volume.

The performance of the solver was mathematically validated using a model of a resonant cavity, which is an accepted canonical problem within CEM applications and provides a straightforward means of validating the solver as its frequency domain response is empirically predicted. Performance of the solver on a typical desktop workstation was also presented in a variety of processing configurations, and the results were discussed and analysed.

References

[5.1] K. V. Beard and C. Chuang, "A new model for the equilibrium shape of raindrops," *Journal of the Atmospheric Sciences*, vol. 44, no. 11, pp. 1509–1524, 1987.

[5.2] K. V. Beard, V. N. Bringi, and M. Thurai, "A new understanding of raindrop shape," *Atmospheric Research*, vol. 97, no. 4, pp. 396–415, 2010.

[5.3] P. S. Pacheco, *Parallel programming with MPI*. Morgan KaufmaMcGraw-Hill, San Francisco, 1997.

[5.4] M. J. Quinn, *Parallel programming in C with MPI and OpenMP*. McGraw-Hill, New York, 2004.

[5.5] R. E. Collin, *Foundations for microwave engineering*. Electrical Engineering Series, McGraw-Hill, New York, 2nd ed., 1992. ch. 7, pp. 500-504.

[5.6] J. L. Herring, *Developments in the transmission-Line modelling method for electromagnetic compatibility studies*. PhD thesis, University of Nottingham, 1993.

[5.7] J. A. Flint, *Efficient automotive electromagnetic modelling*. PhD thesis, Loughborough University, 2000.

[5.8] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," *IEEE Solid-State Circuits Society Newsletter*, vol. 12, no. 3, pp. 19–20, 1967.

Chapter 6

TLM on mobile computing platforms

6.1 Introduction

Any TLM solver designed for mobile computing platforms must be highly-optimised in order to maximise simulation performance. Due to the constraints on processing power and quantity of memory available on such devices, a completely distinct use case must be considered for any mobile solver, as full 3-D simulations would prove impractical when attempting to provide instantaneous feedback to users.

This chapter discusses the limitations of mobile platforms, and the considerations that must therefore be made when developing a TLM solver targeted for mobile devices (section (6.3)). These factors inform the motivations for the application use case, which is as an educational learning aid for undergraduate level students studying in the field of communications and wave theory, as part of a higher-level education in electronic and electrical engineering or physics, for example.

Implementation details are also discussed, including strategies for ensuring simulation performance is maximised despite the constraints present in mobile platforms (section (6.2)). These strategies are analogous to those presented in the previous chapter utilised for the 3-D TLM implementation. However despite some similarities, platform-specific solutions are also presented. Benchmarking results are also presented, with an in-depth performance analysis also included (section (6.4) and (6.5)). Finally a summary

of the mobile solver is given, including discussion on potential avenues for future improvements (section (6.6)).

6.2 Methodology

Accelerating the performance of a TLM solver has been a topic for a wide variety of research papers in recent years [6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7]. Much of the focus of such research has been on parallel computing implementations focusing on medium to large-scale computing platforms and applications. Many of these implementations have focused on desktop multi-core and multi-threading strategies to achieve performance gains with a particular solver, using either CPU, GPU or hybrid approaches with several cores available. In contrast, modern small-scale computing platforms often have a processor on board that contains 1, 2 or sometimes 4 processing cores. For these platforms, it is now feasible for developers to embrace multi-core and multi-threaded programming techniques in order to improve software performance. This strategy is useful for the vast majority of commercial software, as tasks are normally defined at a relatively high-level within source code.

The execution of such tasks can therefore be managed with relative ease when implementing multi-threaded software architectures. However, algorithms for tasks such as image filtering, video encoding, and similar tasks that are suited for parallel processing are defined at a much lower-level, and traditional parallel processing techniques can become harder to manage and implement in these cases. The vast majority of modern small-scale computing platforms utilise CPUs that have an ARM core (or cores) as part of the processor. Of these, most have support for the ARM NEON Media Processing Engine via the adoption of the ARMv7 hardware architecture [6.8]. This is a combination of specialised hardware registers and a programming instruction set that is designed for the vector processing of data values stored within the registers. It should be noted that an orthogonal approach to SIMD processing is also available on desktop and HPC computing systems via the Intel Streaming SIMD Extensions (SSE) architecture present on the majority of Intel-produced CPUs [6.9]. This was extended with the availability of Advanced Vector Extensions (AVX) in CPUs available from 2011 from both Intel and

AMD [6.10]. Intel also provides a comprehensive set of intrinsic functions for both SSE and AVX [6.11], in much the same way as is available for ARM NEON instructions.

This section presents an implementation of a 2-D TLM solver that uses ARM NEON features to achieve a large performance increase without the need to implement traditional multi-core and multi-threaded parallel computing techniques. This reduces implementation time and software complexity for the developer, and was chosen as the preferred strategy for parallelisation before considering multi-threaded computing.

6.2.1 ARM NEON overview

Support for ARM NEON is included within Cortex-A8 processors (as well as optional support within Cortex-A9 processors), which are designed to use the ARMv7 Advanced Single Instruction, Multiple Data (SIMD) instruction set [6.12]. The majority of modern mobile and embedded devices utilise System on Chip (SoC) processors which include these ARM cores. This includes SoCs manufactured in-house, such as the Snapdragon family of processors by Qualcomm [6.13]. In these cases, vendors license the use of the ARMv7 architecture and instruction set. ARM NEON instructions make use of the concept of vector processing, where the same mathematical operation is conducted on multiple values at the same time. This is accomplished via the use of a register bank within Cortex-A8 processors, which the ARM core can be instructed to view as follows [6.14]:

- Thirty-two 64-bit registers
- Sixteen 128-bit registers

The ARM core views these large registers as collections of individual numerical values. The number of different values that can be contained in each register is dictated by the type and precision of the numerical values chosen. Table (6.1) shows some typical configurations that can be chosen for data storage in a NEON register.

As shown in Table (6.1), higher precision values take up more space in memory, and therefore fewer individual values can be stored in a NEON register. Also, as expected, a 128-bit register will always hold twice the amount of values as a 64-bit register for a given data-type. Once data has been loaded into one or more NEON registers, a

Data type	Precision	Values stored in 64-bits	Values stored in 128-bits
Signed int	Byte	8	16
Signed int	Short	4	8
Signed int	Long	2	4
Float	Single	2	4

Table 6.1: Typical ARM NEON register configurations for 64-bit & 128-bit registers.

developer can use the SIMD instruction set to carry out mathematical operations on more than one value at the same time in order to improve performance of an algorithm. An important note is that to date, most of the SIMD instructions can only operate on 64-bits at one time. This means that often, storing values in a 128-bit NEON register before processing will not offer a performance improvement over using 64-bit registers, however some recent versions of ARM cores typically utilised in mobile devices offer a full 128-bit pipeline.

6.2.2 Two-dimensional TLM with ARM NEON

Typically when developing a TLM solver, floating point variables (commonly called 'floats' within many programming languages) are used to hold node port values for use by the scatter and connect function during a simulation. Single precision floats are sufficient for a typical TLM solver, which means each value occupies 32-bits of memory. Therefore, when implementing the TLM solver, two values can be stored in each 64-bit NEON register. This will allow two nodes to be worked on simultaneously. An illustration of this is shown in Fig. (6.1).

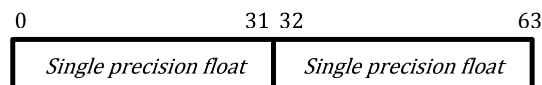


Figure 6.1: Illustration of the storage method within a 64-bit NEON register for two single precision floats.

For 2-D TLM, each node has four port values associated with it. During the scatter and connect phase of the simulation for a given node being processed, the four neighbouring port values belonging to adjacent nodes to the north, east, south and west must be accessed. Therefore, we must use an array of four individual NEON registers within our program. Each register holds a particular neighbouring node's port value for two

consecutive nodes, ready for vector processing. This is illustrated in Fig. (6.2).

$twoN.val[0]$	$V_{xn_{E0}}$	$V_{xn_{E1}}$
$twoN.val[1]$	$V_{yn_{N0}}$	$V_{yn_{N1}}$
$twoN.val[2]$	$V_{xp_{W0}}$	$V_{xp_{W1}}$
$twoN.val[3]$	$V_{yp_{S0}}$	$V_{yp_{S1}}$

Figure 6.2: An array of NEON registers holding neighbouring port values for two adjacent nodes to be processed.

Within a NEON register, individual values are given the name *elements* whilst using the SIMD instruction set. The registers themselves are referred to as *vectors*. This alludes to the parallel nature of the mathematics used during manipulation of the values residing within the registers. The basic premise of vector programming is that contents of one or more vectors are manipulated at the same time in order to speed up algorithms that apply the same operations to large datasets. An example of a vector addition operation using two NEON registers for operands and another as a result register is shown in Fig. (6.3).

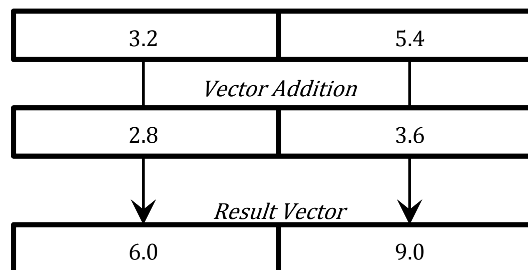


Figure 6.3: An example of vector addition with single precision floats using NEON registers.

6.2.3 A SIMD approach to the TLM method

The investigations for this section of the thesis were conducted using an Apple iPhone 4S, iPad 2, iPad 3, iPhone 5 and iPad 4, in order of processor clock-speed. These modern mobile computing devices contain SoC CPUs which contain a dual core ARM Cortex-A8 or ARM Cortex-A15. Whilst development and benchmarking was conducted on this family of devices, the source code for the vectorised scatter and connect function is highly portable to other mobile and embedded systems using compatible ARM CPUs.

This is due to the low-level nature of the code, as well as the use of NEON intrinsic functions, which are included in the majority of ARM compatible compilers.

The vectorised version of the 2-D TLM algorithm is very similar to that of a traditional serial solver. However, the end result with the NEON implementation is that the source code is more verbose because most arithmetical vector operations can only work on two registers at the same time. This means calculations that can be completed in a single line in the serial version of the solver are split over several lines in the ARM NEON code. After loading the array of registers with neighbouring port values as shown in Fig. (6.2), the total node voltage, V_z, for two consecutive nodes can be vector computed as shown in (6.1).

$$V_z = \sum_{n=0}^{3}(half \times twoN.val[n]) \tag{6.1}$$

In (6.1) above, V_z is a result register which will hold the resulting total node voltage for the two consecutive nodes. When computing V_z using NEON instructions, it is necessary to initialise a separate NEON register named *half*, with a value of 0.5 in floating point representation, ready for use with the vector multiplication operation. The NEON instruction set provides a set of functions for initialising registers so that a given value is duplicated the relevant amount of times across an entire register. Computing E and H field values is accomplished in a similar fashion with vector operations. Examples of these are shown in (6.2), (6.3) and (6.4).

$$E_z = -V_z \tag{6.2}$$

$$H_x = half \times (twoN.val[1] - twoN.val[3]) \tag{6.3}$$

$$H_y = half \times (twoN.val[0] - twoN.val[2]) \tag{6.4}$$

The final step during the scatter and connect process is to vector compute the new port values for both nodes for the subsequent time step. This uses the value of V_z which was

computed earlier, along with the neighbouring port values in the register array. The computation is in the form of (6.5) and repeated for all four ports.

$$twoNReordered.val[2] = V_z - twoN.val[0] \tag{6.5}$$

It should be noted in (6.5) that the result of computing the new port value for each of the two nodes being worked on is stored in a new register array, twoNReordered, of the same dimensions as the original register array, twoN. Each result is stored to a register in this new array, but at a different index. Reordering the new port values in this way allows for the use of a vector store of the array to the data buffer which holds the mesh port values for the subsequent time step. The NEON instruction set includes the ability to populate registers from data buffers and vice versa. Developers can also use specialised load and store functions which operate using pointer offsets to allocated blocks memory in order to load and store multiple values simultaneously. In this case, the port values were reordered in the register array such that the parallel store function vst4_f32 is used to write all eight port values for both nodes to the mesh array at once. A representation of the internal operations of this function is shown in (6.6) and (6.6).

$$pTPlus1 + n = twoNReordered.val[n]_0 \tag{6.6}$$

$$pTPlus1 + n + 4 = twoNReordered.val[n]_1 \tag{6.7}$$

In (6.6) and (6.6) above, pTPlus1 is a pointer to the mesh port value array for the subsequent time step. Specifically, the pointer is set to the first of the four port values for the first of the two nodes being operated on concurrently by the NEON code. Within the vst4_f32 function, the index n is incremented from 0 to 3 inclusive in order to access all four port value registers in the twoNReordered array and store them at the correct points in the TPlus1 data buffer. $[n]_0$ refers to the first element in a given register and $[n]_1$ refers to the second element. In reality, these operations occur in parallel, and therefore storing the port value data back to memory is sped up considerably.

6.2.4 Multi-threaded optimisations

It has been shown that mobile devices present considerations for maximising performance of computationally-intensive algorithms, such as TLM. The previous sections have illustrated that it is first necessary to fully utilise all available hardware acceleration features of a mobile device CPU, in order to ensure maximimum algorithm performance during execution. Once any hardware acceleration features have been utilised, the developement focus can be shifted to multi-threading strategies in order to fully utilise processing hardware on mobile devices that possess more than one physical processor core.

In this case, all of the mobile devices used for development and testing contained a CPU with at least two cores, and indeed this is common place in commercially-available mobile computing devices. Mobile device CPUs adhering to the ARM architecture present identical hardware features across all available cores in the majority of cases. This fact can be exploited to ensure that the performance of the 2-D TLM solver is truly maximised by using multi-threaded computing to spread TLM computations across several threads running on all available processor cores.

Using iOS development as an example, multi-threaded processing is most easily achieved by using the Grand Central Dispatch (GCD) multi-threading library. One of the features of GCD is its ease of use concerning executing functions concurrently across multiple threads. A function which computes the scatter and connect process across a section of the 2-D mesh can be executed concurrently, with each instance working on a different section of the mesh in order to improve simulation performance.

Since creating threads and sending functions to them for execution takes a small amount of processor time itself, the developer can experiment to find the optimum amount of mesh which each instance of the scatter and connect function works on at a time. Changing the amount of nodes processed in turn effects the number of times the function must be executed, and therefore the number of threads that must be spawned every timestep. An example of this using a variable, `stride` to control the amount of Y-axis rows of nodes processed at once, is shown below (in pseudo-code):

```
//GCD Queue, work count and stride variables
```

```
dispatch_queue_t queue = dispatch_get_global_queue(
    DISPATCH_QUEUE_PRIORITY_HIGH, 0);
int stride = 32;

//Dispatch the block (int)YDIM/stride times to the queue
//(asynchronous working but returns only when all blocks executed)
dispatch_apply(YDIM/stride, queue, ^(size_t blockIdx) {
        int startY = blockIdx * stride;
        [self scatterAndConnectNEONWithYStart:startY andYEnd:startY+stride];
});

//Remaining y-iterations (accounts for a remainder from (float)YDIM/stride)
[self scatterAndConnectNEONWithYStart:YDIM-(YDIM%stride) andYEnd:YDIM-1];
```

6.2.5 Summary

This section has described and discussed the methodology and implementation details of a parallel computing optimised 2-D TLM solver for mobile computing devices at a level appropriate for most readers. More in-depth details are available within documentation that comprehensively details the exact software implementation for the iOS mobile platform, which is included at the end of this thesis (Appendix C).

6.3 Motivations for education as a use case

6.3.1 Limitations and considerations

Any software application designed for a mobile platform has a number of restrictions placed upon it simply due to the nature of its execution on a small-scale computing platform. In the case of a TLM solver, the mobile platform presents a number of challenges that must be considered during the design and development stages, which in turn influence potential use cases for the resulting solver.

Hardware considerations

Most obviously, hardware limitations are at the forefront of these considerations. For instance, a typical smartphone or tablet computer will have less (in some cases significantly less) RAM available for applications to utilise during execution than an equivalent desktop system. Depending on the mobile operating system, third-party applications often have lower priority RAM privileges than OS processes and first-party applications. Sometimes hard limits are also enforced on the maximum RAM consumption. For a TLM solver, this limitation results in two design choices; 3-D simulations and the use of SCNs should be avoided to minimise RAM usage. The restriction to 2-D simulations without stubs in turn reduces the complexity of problems that can be modelled using the solver.

Processing performance is also often much lower than a typical desktop system. In this case, the restriction to a 2-D non stub-loaded solver also reduces the number of mathematical operations required per-node in the scatter and connect process. This ensures that processing performance is maximised, even before considering additional optimisation techniques such as SIMD mathematical operations or multi-threading.

User interface considerations

Smartphones and tablets have an established touch-screen interaction method between user and software applications. Very often a physical keyboard is also not present. In the context of the mobile solver, this means that precisely defining mesh objects and pulse excitations is not possible due to a lack of a precision pointing device. A mixture of pre-configured simulation environments and engaging free-form simulations based on the touch-screen UI were favoured to reduce the complexity of use whilst the solver is executed. Precisely configured user simulation environments should be restricted to being authored on external computers and imported in via a webserver link or a similar scheme.

6.3.2 Education as a use case

The limitations and considerations discussed previously for a solver running on mobile devices result in a 2-D simulation environment with a greatly simplified user interaction method, via the touch-screen. Presenting the solver as an education learning aid application ensures these limitations are not a hindrance to the use of the solver. Several pre-configured 2-D simulations can be included to demonstrate specific principles regarding wave theory within the scope of an Electronic and Electrical engineering or physics degree. Free-form dynamic simulations can also be facilitated by allowing the user to place point source excitations and boundaries simply by using their finger with the touch-screen. Finally, additional pre-configured simulations can be specified by lecturers or students by using a suitable XML scheme and having the application parse files held on remote webservers.

Educational software for mobile computing devices must be multi-platform to remove the need for dedicated devices to be provided to students. This ensures application ubiquity amongst a student cohort. Recent smartphone market share figures show that Android at 75%, with iOS at 14.9% and Windows Phone at 2% [6.15]. These three platforms represent over 90% of the smartphones sold. Development of the application therefore focused on supporting these platforms to ensure ubiquitous take-up of the application.

The application has several pre-defined simulation modes that illustrate different principles and phenomena from electromagnetic wave theory. End-users are also able to interact with the application in a free-form manner, initiating point sources of both Gaussian and sinusoidal types in response to finger presses on the screen. Custom boundaries can also be placed by dragging a finger on the screen. Educators have flexibility to implement simulations to their own design. These simulations are authored using CEML [6.16], an XML-based configuration language designed by the authors as a platform-agnostic standard for defining Computational Electromagnetic (CEM) simulations.

6.4 Benchmarking results

In order to test the performance of both the serial and SIMD versions of the solver, an example problem will be used during execution of the simulation. We have chosen a widely-used example, a 1 m^2 cavity. In our case, we will model the cavity at a resolution of 301 by 301 nodes (a total of 90,601 nodes), with boundaries configured to be a perfect electric conductor (PEC). The final benchmarking was conducted without outputting the results to the screen in order to obtain a peak performance figure, however for clarity a screenshot of the simulated cavity is shown in Fig. (6.4).

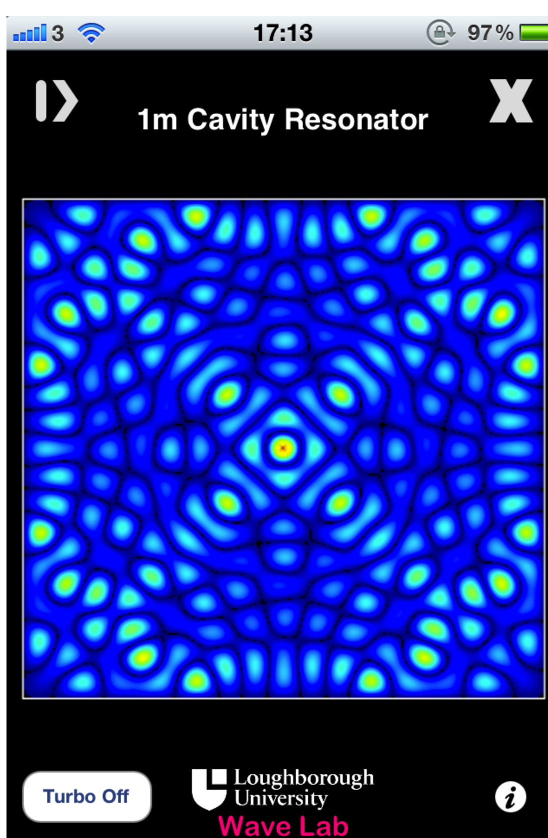


Figure 6.4: A visualisation of the simulated 1 m^2 cavity resonator used for benchmarking.

Following the implementation of the SIMD scatter and connect function using the NEON instruction set, a benchmarking test was conducted to compare the simulation speed with that of the original serial TLM solver. Details of the resulting performance metrics are included in Table (6.2).

With the results shown above in Table (6.2), all simulations were conducted using the

	Processing Speed ($nodes.s^{-1} \times 10^6$)	
Device	**Serial**	**NEON SIMD**
iPhone 4S	6.010	11.385
iPad 2	7.479	12.506
iPad 3	7.521	12.525
iPhone 5	17.766	29.393
iPad 4	18.965	34.962

Table 6.2: Benchmarking results comparing ARM NEON SIMD & serial-processing TLM solvers on mobile devices.

same mesh dimensions in order to provide a fair comparison for benchmarking both versions of the solver. The values that were obtained for processing speed in $nodes.s^{-1}$ were calculated with time stamping code, with final speed calculations completed using the method shown in (6.8).

$$v_{proc} = \frac{N}{t_{sc}} \tag{6.8}$$

As shown above in (6.8), the processing speed in $nodes.s^{-1}$ v_{proc}, is calculated using the time taken for each iteration of the scatter and connection function, t_{sc}. This time difference is calculated using accurate time stamps both before and after each completed iteration of the scatter and connect routine. To find $_{proc}$, the total number of nodes in the simulation, N, is also required. In practice, the value obtained for t_{sc} was the mean value taken over 10 iterations to reduce the effect of any inaccuracies in the time stamp values. Fig. (6.5) shows results of executing the benchmarking problem on a variety of devices using ARM Cortex-A8 and A9 processors. All results are shown as the factor of performance improvement, normalised to the performance of the serial code on the iPhone 4S.

6.5 Performance analysis

A parallelised implementation of a traditional 2-D TLM solver has been demonstrated on the iOS platform. ARM NEON hardware support was exploited, with the potential for a maximum performance increase of 100% (i.e. a doubling of performance) in agreement with the principles proposed by Amdahl [6.17]. Amdahl's Law is a method for

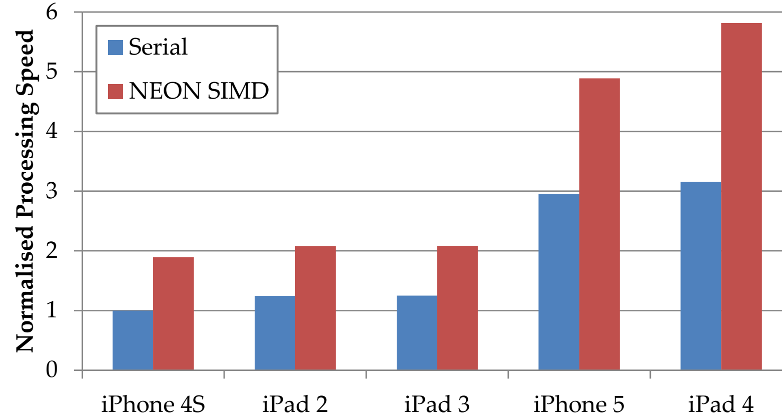


Figure 6.5: Normalised processing speed comparision between the ARM NEON SIMD TLM solver and the serial TLM solver.

calculating the expected performance increase factor of a piece of software due to improvements in performance to a portion of the application. Originally, the relationship represented a performance increase in terms of a ratio of improved to non-improved code, as well as factoring in a specific speed up factor observed for the improved code. It is trivial to modify Amdahl's Law to illustrate the performance increase due to the ratio of parallel to serial code present in a piece of software, whilst taking into account the number of processing units available on the hardware running the application. Equation (6.9) demonstrates this, with S being the performance increase factor, P being the ratio of parallel code in the application, and N being the number of processing units. Through benchmarking, the performance increase factor S was found for a number of devices, with N equal to two in each case as ARM NEON vector operations occur on two values at once. The original equation for Amdahl's Law can be re-arranged to solve for P to determine the exact ratio of parallel to serial code in the application. This is shown in equation (6.10).

A comprehensive review of the relationship between the TLM method and general approaches to adapting it for parallel computation has previously been conducted by Stothard [6.18]. As stated previously, it is possible for both the scatter and connect processes within the TLM algorithm to be completed by a parallel processing scheme due to data independence across mesh nodes within a single time-step. This suggests that in the case of two processing units, attainable performance increases should approach 100% as the scatter and connect processes form a large part of the computation time of

the TLM algorithm.

As shown previously in Fig. (6.5), observations from benchmarking both versions of the solver showed a maximum performance increase of 89% over the serial version of the application, which was equivalent to the parallelised fraction of the solver code as being 94%. This is very close to the theoretical limit for a system using two computing units, and confirms an efficient parallel TLM implementation aided by the potential for the scatter and connect process to be almost completely parallelised. On the iPad 2 and iPad 3, the performance increase observed when using the NEON SIMD solver is 67%. It is interesting to note that the increased CPU clock speed of the iPad 2 and iPad 3 (at 1GHz) as compared to the iPhone 4S (at 800MHz) does not produce the same relative performance increase factor as is observed with the iPhone 4S.

$$S = \frac{1}{(1-P) + \frac{P}{N}} \tag{6.9}$$

$$P = \frac{N(S-1)}{S(N-1)} \tag{6.10}$$

6.6 Summary

Using devices containing SoCs leveraging the ARMv7 architecture and the NEON instruction set, it has been shown that performance increases of nearly double the base figures can be achieved. It has been demonstrated that the conversion of the scatter and connect function of the solver to a version that uses ARM NEON intrinsic functions to parallelise the algorithm requires a relatively low amount of effort from the developer. This is in contrast to utilising traditional parallel computing techniques for increasing the performance of the solver, which require management of multiple threads and/or parallel data transfer across multiple processing cores.

The observed large increases in performance observed on all devices that were benchmarked, coupled with the low complexity of the parallelization method justifies the case for altering the parallel computing strategy when implementing TLM solvers on mobile and embedded platforms. Further research and testing is required to fully understand the cause of the difference in performance increase factors when comparing the iPhone

4S results to those of the iPad 2 and iPad 3. The hypothesis was that a performance increase factor that was similar to that of the iPhone 4S would be observed in the other devices. For this to have been the case, NEON SIMD performance on the both iPads should have achieved approximately 14.140×10^6 $nodes.s^{-1}$. It is possible that when running the solver on the iPad 2 and iPad 3 that a bottleneck has been reached with regards to the memory hierarchy and the ARM NEON registers used for parallel vector computation within the TLM mesh. This would explain why the expected increase in the performance (approximately 25%) was observed with the serial solver when comparing the iPhone 4S and iPad results, as the serial solver does not utilise the NEON registers, whereas the NEON SIMD solver does.

References

[6.1] P. P. M. So, "Time-domain computational electromagnetics algorithms for GPU based computers," in *The Intl. Conf. on "Computer as a Tool"*, pp. 1–4, 9-12 Sept. 2007.

[6.2] F. V. Rossi, P. P. M. So, N. Fichtner, and P. Russer, "Massively parallel two-dimensional TLM algorithm on graphics processing units," in *IEEE MTT-S Intl. Microwave Symp. Digest*, pp. 153–156, 15-20 June 2008.

[6.3] F. V. Rossi and P. P. M. So, "Accelerated symmetrical condensed node TLM algorithms for NVIDIA CUDA enabled graphics processing units," in *Intl. Conf. on Electromagnetics in Advanced Applications*, pp. 170–173, 14-18 September 2009.

[6.4] V. A. Chouliaras, J. A. Flint, and L. Yibin, "Parametric data-parallel architectures for TLM acceleration," in *Computational Electromagnetics and Its Applications, 2004. Proceedings. ICCEA 2004. 2004 3rd Intl. Conf. on*, pp. 569–572, 1-4 Nov. 2004.

[6.5] V. A. Chouliaras, J. A. Flint, L. Yibin, and J. L. Nunez-Yanez, "A system-on-chip vector multiprocessor for transmission line modelling acceleration," in *Signal Processing Systems Design and Implementation, IEEE Workshop on*, pp. 568–572, 2-4 Nov. 2005.

[6.6] C. Altinbasak and L. T. Ergene, "Two-dimensional parallel transmission line matrix solver for waveguide analysis," *Microwave and Optical Technology Letters*, vol. 50, no. 1, pp. 95–98, 2007.

[6.7] P. Lorenz, J. V. Vital, B. Biscontini, and P. Russer, "High-throughput transmission line matrix (TLM) system in grid environment for microwave design, analysis

and optimizations," *IEEE MTT-S Intl. Microwave Symp. Digest*, pp. 1115–1118, 2005.

[6.8] "ARM Cortex-A series processors." http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.set.cortexa/index.html.

[6.9] "Intel SSE4 programming reference." http://home.ustc.edu.cn/~shengjie/REFERENCE/sse4_instruction_set.pdf.

[6.10] "Intel AVX." https://software.intel.com/en-us/intel-isa-extensions\#pid-16007-1495.

[6.11] "Intel intrinsics guide." https://software.intel.com/sites/landingpage/IntrinsicsGuide/.

[6.12] "Arm Cortex A8 technical reference manual." http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0344k/index.html.

[6.13] "Qualcomm Snapdragon processors detail page." http://www.qualcomm.com/snapdragon.

[6.14] "VFP views of the register bank." http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0344k/ch13s02s02.html.

[6.15] "Third quarter 2012 smartphone market share results." http://www.idc.com/getdoc.jsp?containerId=prUS23771812.

[6.16] "Computational electromagnetic markup language." https://github.com/danielrbrowne/CEML.

[6.17] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," *IEEE Solid-State Circuits Society Newsletter*, vol. 12, no. 3, pp. 19–20, 1967.

[6.18] D. Stothard, *The development of an application specific processor for the transmission line matrix method*. PhD thesis, Loughborough University, 1999. pp. 17-27.

Chapter 7

Conclusions

The aim of this thesis was to identify different approaches for conducting CEM simulations on both large and small-scale computing platforms. These aims have been met by formulating a suitable case study for both classes of computing platform, and focussing on optimisation techniques as part of the implementation for both platforms. This chapter details the contributions of this thesis to the field, as well as suggesting avenues that warrant further study.

7.1 Contributions to the field

It is clear that increasingly higher frequencies in the radio spectrum are being used across the communications industry and this places new demands on high-performance CEM applications. This thesis has explored the challenges and strategies required in order to design and develop CEM solvers that are optimised for both small-scale and large-scale computing platforms. Mobile computing devices (such as smartphones and tablets) and desktop/HPC cluster systems have been addressed by the thesis.

A robust method for modelling randomly-generated rain-fields has been demonstrated and validated. An existing geometrical model has been utilised in combination with a DRV in order to generate raindrops of random sizes that adhere to a given size distribution. A method for collision prevention was also demonstrated, ensuring the randomly-placed raindrops do not intersect one another during placement. Two meth-

ods for discretising the raindrop geometry were discussed, including a free-to-use tool *SketchUp* which offers flexible Ruby-based scripting to end users.

An implementation of a 3-D SCN TLM solver was presented, which was optimised for desktop and HPC cluster systems and investigations in to rain-fields was chosen as a case study problem for use with the solver. In order to execute simulations using frequencies in the EHF band and above in a timely fashion, it was shown that a hybrid approach to parallel computation was necessary. Optimisations to the TLM algorithm in order to reduce the number of mathematical operations were also shown to be of benefit in order to ensure maximum processing performance. A novel method for estimating algorithm speeds when employing different implementations of the same mathematical algorithm was presented. This allows a more direct comparison of algorithmic implementations when exact low-level processor instruction timing is not known for a given computing platform.

The first known implementation of 2-D TLM on mobile computing devices was presented, available on the iOS platform (iPhone, iPad and iPod touch devices). A case study was devised for the application to be developed as an educational learning aid targeted at undergraduate-level students studying electromagnetics, and their lecturers. The challenges of developing a computationally time-intensive application such as this have been discussed in-depth. Strategies for optimising the performance of a mobile 2-D TLM solver have been developed, utilising a combination of floating-point vector mathematics and multi-threading in order to maximise the performance of the solver on contemporary mobile computing devices.

7.2 Further work

The work carried out in this thesis has presented several areas for further research.

7.2.1 Rain-field modelling performance

An in-depth comparative study is required between the performance of a rain-field volume utilising periodic boundaries and a larger volume discretised at the same

resolution, with simulations executed for equivalent periods of time. The effects on excitation by multiple passes through the quasi-random rain-fields in meshes utilising periodic boundaries, as compared to a larger truly-random meshes where excitation passes through only once is an interesting problem. Any differences in behaviour should therefore be quantified.

7.2.2 Modelling performance study using large-scale platforms

Work in this thesis has demonstrated that it is possible to implement a 3-D TLM solver that is optimised for parallel computation running on either high-performance desktop workstations or HPC cluster systems depending on user requirements. Extending this proof of concept implementation for rain-field modelling into a comprehensive performance study is the logical next step. As identified in this thesis, the case study topic of rain-field modelling provides ample opportunity for real-world simulations to be conducted. An in-depth understanding of the performance characteristics of the 3-D TLM solver across a number of different platforms would prove useful in this regard.

7.2.3 3-D simulations on mobile devices

It has been demonstrated that by using a hybrid approach to parallel computing on mobile devices, it is possible to produce software for mobile computing devices which can execute 2-D TLM at a more than adequate processing rate to allow for dynamic simulations that an end user can interact with. The continually-improving processing performances and hardware features available on such mobile devices suggest that the same style of approach can be extended to explore the possibility of a full 3-D TLM solver in the not too distant future. Any innovations along these lines will greatly extend the potential for educational assistance by allowing a larger number of electromagnetic principles and phenomena to be demonstrated to students. It should also be noted that a full 3-D solver on mobile devices may also provide some use to industrial end users in commercial applications.

7.2.4 Mobile solver usage study

A large amount of anonymised usage data has been generated by the inclusion of an analytics library to the iOS version of the 2-D TLM solver. After data collection began, it became clear that any in-depth analysis of this data, or further study was outside of the scope of this thesis. However, it is clear that a review of the usage data (as well as expanding data collection to other mobile platforms) could inform future developments in mobile CEM applications. It is also entirely possible that any study based on this usage data could aid in the development and understanding of other mobile device applications that are not necessarily directly related to CEM applications. A record of this analytics data is included (Appendix D).

7.2.5 Ad-hoc cluster computing using mobile devices

Mirroring the approach taken in the development of the 3-D TLM solver application, there is an opportunity to explore the use of the wifi networking hardware on almost all modern smartphones and tablets for cooperative computation of larger mesh volumes. The majority of smartphone operating systems allow for some form of ad-hoc network connections between multiple devices, and therefore a software architecture to divide a larger mesh between devices would present an interesting avenue for investigation. Sub-mesh data could be exchanged in a similar fashion to that employed in the 3-D implementation presented in this Thesis, with the exception of using a wireless link between compute nodes instead of a physical one.

7.3 Overall conclusion

In order to produce efficient and optimised solver applications, it is necessary to take a very different approach when comparing small-scale mobile and desktop/HPC cluster platforms. In both cases however, some form of parallel computing has been shown to be required in order to achieve the best possible simulation performance figures.

Investigations with 3-D TLM solvers for desktop and HPC cluster systems have demonstrated the ability for extremely large or extremely high frequency simulations to be

possible, due to advances in modern parallel computing hardware and software. By choosing a flexible software architecture, a parallel computing 3-D TLM solver can keep pace with the ever-increasing requirements of industrial customers by employing computers with larger numbers of compute nodes and more RAM per node in order to execute larger, or more finely-graded simulations.

On the opposite end of the computing spectrum, mobile computing devices are now a viable platform for CEM solver applications, and it is possible to offer a piece of multi-platform software as an educational application that allows simulations to be rendered at or above 60Hz. Viable optimisation techniques have been presented which allow this high level of performance despite the restrictions of a mobile computing platform.

Appendix A

SketchUp raindrop meshing script

```
# First we pull in the standard API hooks.
require 'sketchup.rb'

# Add a menu item to launch our plugin.
UI.menu("PlugIns").add_item("Simulate␣Rain-field") {
  prompts = ["R.␣rate␣(mm/h):", "Sim␣X␣(mm):", "Sim␣Y␣(mm):", "Sim␣Z␣(mm):",
      "Fmax␣(GHz):", "Draw␣Voxels:"]
  defaults = ["3178800", "20.0", "20.0", "20.0", "3.351", "NO"]
  list = ["", "", "", "", "", "YES|NO"]
  input = UI.inputbox prompts, defaults, list, "Randomised␣Rain-field␣Options
      "

  #Input variables and constants
  #terminal velocity of raindrops in air (m/s)
  vTDrop = 8.83
  #calculate volume ratio, convert rainfall rate to m/h figure,
  #then divide by the height a drop will fall in 1hr
  volumeRatio = (input[0].to_f/1000)/(vTDrop*3600)
  fieldX = input[1].to_f
  fieldY = input[2].to_f
  fieldZ = input[3].to_f
  fMax = input[4]
  if (input[5] == "YES")
        shouldDrawVoxels = true
  else
```

```
      shouldDrawVoxels = false
end
totalDropsVolume = 0.0
simVolume = fieldX*fieldY*fieldZ
fourThirdsPI = (4.0/3.0)*Math::PI
dropDistArray = []
fileNameCount = 0
#First drop occurence flags
firstDrops = [true, true, true, true, true, true, true, true, true]

#Raindrop size markers
rDrops = [0.0, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0, 9.0]      #0.0
    and 9.0 for N-dist tails
#maximum radii of each drop size, pre-computed from Beard and Chuang model
#maxRadius = [1.026, 1.299, 1.582, 1.876, 2.177, 2.487, 2.808, 3.138,
    3.477]
maxRadiusAngle = [80, 80, 80, 81, 81, 82, 82, 82, 83]

#Decimal places arrays for creating a N-dist of raindrops when applying a
    RNG
# p(0.0) = 0.0502 # p(2.0) = 0.0504 # p(2.5) = 0.0801 # p(3.0) = 0.1112
# p(3.5) = 0.1357 # p(4.0) = 0.1448 # p(4.5) = 0.1357 # p(5.0) = 0.1112
# p(5.5) = 0.0801 # p(6.0) = 0.0504 # p(9.0) = 0.0502
tenths         = [rDrops[3], rDrops[4], rDrops[5], rDrops[6], rDrops[7]]
hundredths     = [rDrops[0], rDrops[0], rDrops[0], rDrops[0], rDrops[0],
    rDrops[1], rDrops[1], rDrops[1], rDrops[1], rDrops[1], rDrops[2],
    rDrops[2], rDrops[2], rDrops[2], rDrops[2], rDrops[2], rDrops[2],
    rDrops[2], rDrops[3], rDrops[4], rDrops[4], rDrops[4], rDrops[5],
    rDrops[5], rDrops[5], rDrops[5], rDrops[6], rDrops[6], rDrops[6],
    rDrops[7], rDrops[8], rDrops[8], rDrops[8], rDrops[8], rDrops[8],
    rDrops[8], rDrops[8], rDrops[8], rDrops[9], rDrops[9], rDrops[9],
    rDrops[9], rDrops[9], rDrops[10], rDrops[10], rDrops[10], rDrops[10],
    rDrops[10]]
thousandths    = [rDrops[3], rDrops[4], rDrops[4], rDrops[4], rDrops[4],
    rDrops[4], rDrops[5], rDrops[5], rDrops[5], rDrops[5], rDrops[6],
    rDrops[6], rDrops[6], rDrops[6], rDrops[6], rDrops[7]]
tenThousandths = [rDrops[0], rDrops[0], rDrops[1], rDrops[1], rDrops[1],
    rDrops[1], rDrops[2], rDrops[3], rDrops[3], rDrops[4], rDrops[4],
    rDrops[4], rDrops[4], rDrops[4], rDrops[4], rDrops[4], rDrops[5],
```

```
    rDrops[5], rDrops[5], rDrops[5], rDrops[5], rDrops[5], rDrops[5],
    rDrops[5], rDrops[6], rDrops[6], rDrops[6], rDrops[6], rDrops[6],
    rDrops[6], rDrops[6], rDrops[7], rDrops[7], rDrops[8], rDrops[9],
    rDrops[9], rDrops[9], rDrops[9], rDrops[10], rDrops[10]]

#Unit vector in z-axis (direction of freefall) for angle comparison
zAxisVect = Geom::Vector3d.new(0.0,0.0,1.0)
#Normalised drop vector for angle calculations
normalisedDropVect = Geom::Vector3d.new

currentDropRArray = []
previousDropRArray = []
dropMaxRArray = []
dropZeroDegRArray = []
drop180DegRArray = []

#Physical constants for water (worst case meshing)
epsilonR = 80.0
muR = 1.0

#Calculate the deltaL for node size & output it to the config file, along
    with simulation environment dimensions (converted to metres)
dL = 299792458/(Math::sqrt(epsilonR*muR)*10*((fMax.to_f)*1000000000))
fileNameString = 'C:\Users\eldrb2\Personal␣Documents␣-␣Not␣Backed␣Up\
    MyDropBox\Dropbox\Uni\PhD\Development\Sketchup␣Files\config.txt'
f = File.open(fileNameString, 'w')
f.puts (fieldX/1000.0)
f.puts (fieldY/1000.0)
f.puts (fieldZ/1000.0)
f.puts dL
f.close

#RNG and relevant drop size selection
startTime = Time.now
while ((totalDropsVolume/simVolume) <= volumeRatio) do
      xRNG = 0.0
      yRNG = 0.0
      zRNG = 0.0
      randDiameter = 0.0
```

```
distFlag = 0
comp = 0
simBoxFlag = 0

#return a N-distributed, randomly generated drop diameter value
until (randDiameter > 0.0 && randDiameter < 9.0) do
        diameterRNG = rand(10000)
        if (diameterRNG >= 9960)
                randDiameter = tenThousandths[diameterRNG-9960]
        elsif (diameterRNG < 9960 && diameterRNG >= 9800)
                randDiameter = thousandths[(diameterRNG-9800)/10]
        elsif (diameterRNG < 9800 && diameterRNG >= 5000)
                randDiameter = hundredths[(diameterRNG-5000)/100]
        else (diameterRNG < 5000)
                randDiameter = tenths[diameterRNG/1000]
        end
end

#compute the volume of current drop
dropRadius = randDiameter/2.0
dropVolume = fourThirdsPI*(dropRadius**3)
#add to total drops volume (only if adding drop would be nearer
    volumeRatio target than not adding)
#if ((volumeRatio - ((totalDropsVolume += dropVolume)/simVolume)).abs
     < (volumeRatio - (totalDropsVolume/simVolume)).abs)
        totalDropsVolume += dropVolume
#end

#generate random X, Y, Z coordinates
#(making sure edge of drop inside simulated space + no drop overlap)
dropIndex = (randDiameter-2.0)*2.0
#Get radius, y and z co-ordinates for the drop at it's maximum radius
     extent
#As well as radius, y and z co-ordinates for drop parallel to z-axis
    (gravity)
#(Used for simulation-bound testing to ensure drops are not out-of-
    bounds)
dropMaxRArray = dropRadiusAtAngle(randDiameter, maxRadiusAngle[
    dropIndex])
```

```
dropZeroDegRArray = dropRadiusAtAngle(randDiameter, 0)
drop180DegRArray = dropRadiusAtAngle(randDiameter, 180)

until (distFlag == 1 && simBoxFlag == 1) do
        xRNG = rand()*fieldX
        yRNG = rand()*fieldY
        zRNG = rand()*fieldZ
        simBoxFlag = 0

        #check for out of bounds drops
        if (xRNG > dropMaxRArray[1].abs && xRNG < (fieldX -
            dropMaxRArray[1].abs) && yRNG > dropMaxRArray[1].abs &&
            yRNG < (fieldY - dropMaxRArray[1].abs) && zRNG >
            dropZeroDegRArray[2].abs && zRNG < (fieldZ -
            drop180DegRArray[2].abs))
                simBoxFlag = 1
        else
                simBoxFlag = 2
        end

        #check for overlapping drops
        comp = 0
        if (simBoxFlag == 1)
                for ii in dropDistArray
                        #Vector joining current and previous drops
                        dropV = Geom::Vector3d.new(xRNG-ii[0], yRNG-
                            ii[1], zRNG-ii[2])
                        #distance between current drop and drop to be
                             checked against
                        #convert from model units to raw float
                            numerical value
                        dropVLength = dropV.length
                        dropVLength = dropVLength.to_f
                        normalisedDropVect.x = dropV.x / dropVLength
                        normalisedDropVect.y = dropV.y / dropVLength
                        normalisedDropVect.z = dropV.z / dropVLength

                        #dot product of the two vectors
                        vectDotProduct = normalisedDropVect.dot
```

```
                            zAxisVect

                        #coerce dot product into -1 to +1 range for
                            acos
                        if (vectDotProduct < -1.0)
                                vectDotProduct = -1.0
                        end
                        if (vectDotProduct > 1.0)
                                vectDotProduct = 1.0
                        end
                        #Compute angle between two vectors for both
                            current and previous drops
                        currentDropAngle  = Math::acos(vectDotProduct
                            )*(180/Math::PI)   #Two unit vectors used
                            , so dot prod == cos(angle)
                        #the angle between the vector and the z-axis
                            on the previous drop being tested
                        #is the complementary angle on a straight
                            line to the angle calculated above
                        previousDropAngle = 180 - currentDropAngle
                        currentDropRArray = dropRadiusAtAngle(
                            randDiameter, currentDropAngle)
                        previousDropRArray = dropRadiusAtAngle(ii[3],
                             previousDropAngle)

                        #check whether radii sum of current +
                            previous drop is smaller than distance
                            between drop centres
                        #if so, collision has occured
                        if (dropVLength < (currentDropRArray[0]+
                            previousDropRArray[0]))
                                comp += 1
                        end
                end
        end
        if (comp > 0)
                distFlag = 2
        else
                distFlag = 1
```

```
            end
        end

        #draw raindrops
        dropStartTime = Time.now

        #convert diameter value to index for array
        dIndex = (randDiameter-2.0)*2.0

        #IF      first occurence of particular diameter, also write prototype
            drop file
        #ELSE   draw a 'regular' drop where prototype mesh info is not saved
            to disk
        if firstDrops[dIndex]
                isPrototypeDrop = true
                fileCounter = dIndex.to_i
                #reset flag
                firstDrops[dIndex] = false
        else
                isPrototypeDrop = false
                fileCounter = fileNameCount
        end

        #draw raindrop, with shouldDrawVoxels and isPrototypeDrop options
            decided previously
        dropGroup = drawRaindrop(randDiameter, xRNG, yRNG, zRNG, fileCounter,
             isPrototypeDrop)
        voxelize(dropGroup, dL, fileCounter, shouldDrawVoxels,
            isPrototypeDrop)

        puts "Drop␣Draw␣Time:␣"+(Time.now-dropStartTime).to_s

        #append to array
        dropDistArray << [xRNG, yRNG, zRNG, randDiameter]

        #increment fileNameCount
        fileNameCount +=1
    end
```

```
  #Write the total number of raindrops to the config.txt file
  fileNameString = 'C:\Users\eldrb2\Personal␣Documents␣-␣Not␣Backed␣Up\
      MyDropBox\Dropbox\Uni\PhD\Development\Sketchup␣Files\config.txt'
  f = File.open(fileNameString, 'a')
  f.puts fileNameCount
  f.close

  puts "Rain-field␣Simulation␣Time:␣"+(Time.now-startTime).to_s
}

### ----------------------------------------------------------------- ###
#
#  dropRadiusAtAngle(d, angle)
#
def dropRadiusAtAngle(d, angleInDegs)
  #convert diameter value to index for array
  dIdx = (d-2.0)*2.0
  #original radius value for draw loop
  originalRadius = d/2.0
  #pi divided by 180 to save time in loop
  piOver180 = Math::PI/180
  #drop radius co-ordinates array
  rCoords = []

  #Distortion coefficients array for Beard & Chuang raindrop model
  distCoefficients = []
  #Nth order components     1         2         3         4         5       6
            7         8         9        10        11
  distCoefficients[0] = [-0.0131, -0.0120, -0.0376, -0.0096, -0.0004, 0.0015,
       0.0005,   0.0000,  -0.0002,  0.0000,   0.0001]     #2.0mm
  distCoefficients[1] = [-0.0201, -0.0172, -0.0567, -0.0137,  0.0003, 0.0029,
       0.0008,  -0.0002,  -0.0004,  0.0000,   0.0001]     #2.5mm
  distCoefficients[2] = [-0.0282, -0.0230, -0.0279, -0.0175,  0.0021, 0.0046,
       0.0011,  -0.0006,  -0.0007,  0.0000,   0.0003]     #3.0mm
  distCoefficients[3] = [-0.0369, -0.0285, -0.0998, -0.0207,  0.0048, 0.0068,
       0.0013,  -0.0013,  -0.0010,  0.0000,   0.0005]     #3.5mm
  distCoefficients[4] = [-0.0458, -0.0335, -0.1211, -0.0227,  0.0083, 0.0089,
       0.0012,  -0.0021,  -0.0013,  0.0001,   0.0008]     #4.0mm
  distCoefficients[5] = [-0.0549, -0.0377, -0.1421, -0.0240,  0.0126, 0.0110,
```

```
      0.0009,  -0.0031,  -0.0016,  0.0004,  0.0011]     #4.5mm
  distCoefficients[6] = [-0.0644, -0.0416, -0.1629, -0.0246,  0.0176, 0.0131,
      0.0002,  -0.0044,  -0.0018,  0.0009,  0.0014]     #5.0mm
  distCoefficients[7] = [-0.0742, -0.0454, -0.1837, -0.0244,  0.0234, 0.0150,
      -0.0007, -0.0058,  -0.0019,  0.0015,  0.0019]     #5.5mm
  distCoefficients[8] = [-0.0840, -0.0480, -0.2034, -0.0237,  0.0297, 0.0166,
      -0.0021, -0.0072,  -0.0019,  0.0024,  0.0023]     #6.0mm

  #construct the radius by computing overall distortion to a spherical drop
  distSum = 0.0
  for i in 0..10
          distSum += distCoefficients[dIdx][i]*Math.cos((i*angleInDegs)*
              piOver180)
  end
  rCoords[0] = (originalRadius*(1.0+distSum))
              #r
  rCoords[1] = (rCoords[0]*Math.sin(angleInDegs*piOver180))             #y
  rCoords[2] = (-(rCoords[0]*Math.cos(angleInDegs*piOver180)))  #z

  return rCoords
end

### ----------------------------------------------------------------- ###
#
#  drawRaindrop(d, cX, cY, cZ, fCount, isProtoDrop)
#
def drawRaindrop(d, cX, cY, cZ, fCount, isProtoDrop)
  # IF regular raindrop, write raindrop diameter and centre-point info to
      file
  # ELSE for Prototype raindrops, only write diameter (centre-point is
      arbitrary)
  if (isProtoDrop == false)
        fileNameString = 'C:\Users\eldrb2\Personal␣Documents␣-␣Not␣Backed␣Up\
            MyDropBox\Dropbox\Uni\PhD\Development\Sketchup␣Files\drop' <<
            fCount.to_s
        fileNameString <<= '.txt'

        f = File.open(fileNameString, 'w')
        f.puts d
```

```
      f.puts "STANDARD"
      f.puts (cX/1000.0)
      f.puts (cY/1000.0)
      f.puts (cZ/1000.0)
      f.close
else
  fileNameString = 'C:\Users\eldrb2\Personal␣Documents␣-␣Not␣Backed␣Up\
      MyDropBox\Dropbox\Uni\PhD\Development\Sketchup␣Files\proto' << fCount
      .to_s
      fileNameString <<= '.txt'

      f = File.open(fileNameString, 'w')
      f.puts d
      f.puts "PROTOTYPE"
      f.close
end

# Get "handles" to our model and the Entities collection it contains.
model = Sketchup.active_model
entities = model.entities

#Make a group to place raindrop into
group = entities.add_group
entities_group = group.entities

#placeholder array inits for pointsArray & dropRArray
pointsArray = []
dropRArray = []

#Create a pointsArray for coordinates for a drop outline curve
for degreesIdx in 0..180
  #generate the radius, y and z co-ordinates for the given drop outline at
      the current angle
      dropRArray = dropRadiusAtAngle(d, degreesIdx)
      #scale from inches to mm and also use a 1000x multiplier for
          increased prescision when drawing drops
      pointsArray[degreesIdx] = [(1000*(0.0+cX)).mm, (1000*(dropRArray[1]+
          cY)).mm, (1000*(dropRArray[2]+cZ)).mm]
```

```
  end

  # Make raindrop curve outline into a face
  new_face = entities_group.add_face pointsArray
  # Draw a circle perpendicular to the drop outline as a path for a
      rotational sweep
  center_point = Geom::Point3d.new(1000*(cX.mm),1000*(cY.mm),1000*(cZ.mm))
  normal_vector = Geom::Vector3d.new(0,0,1)
  radius = 800.mm
  edgearray = entities_group.add_circle center_point, normal_vector, radius
  first_edge = edgearray[0]
  arccurve = first_edge.curve
  # Do the rotational sweep
  new_face.followme edgearray
  #Set the model's entities using the entities contained in the group
  entities = group.entities
  return group
end

### ----------------------------------------------------------------- ###
#
#  voxelize(dGroup, dL, fCount, shouldDrawCubes, isProtoDrop)
#
def voxelize (dGroup, dL, fCount, shouldDrawCubes, isProtoDrop)
        #standard model stuff
        model = Sketchup.active_model

        #Physical constants for water (worst case meshing)
        epsilonR = 80.0
        muR = 1.0

        #convert to a voxel size scaled up by a factor of 1000 - since we are
             drawing drops 1000x larger than normal due to Sketchup's
            precision issues
        #convert to a string (seems to be the only way for length conversion
            to work properly)
        vSizeLString = (dL*1000).to_s
        #convert to a length object for voxel Sketchup drawing use
        vSizeLength = vSizeLString.to_l
```

```
    #Halfscale constant
    halfScale = Geom::Transformation.scaling 0.5

    #begin
    t1 = Time.now
    model.start_operation "Voxelize(cubegrid␣#{fCount})" ### 0.3.0  added
         op number

# Get corners of group
bound = dGroup.local_bounds
corners = []
for ii in 0..7
    corners[ii] = bound.corner(ii)
end

# Create probing vector (in x direction)
xVect = Geom::Vector3d.new(corners[0],corners[1])
nx = (xVect.length/vSizeLength).round
xVect.length = vSizeLength

yVect = Geom::Vector3d.new(corners[0],corners[2])
ny = (yVect.length/vSizeLength).round
yVect.length = vSizeLength

zVect = Geom::Vector3d.new(corners[0],corners[4])
nz = (zVect.length/vSizeLength).round
zVect.length = vSizeLength

diagVect = (xVect+yVect+zVect).transform! halfScale
diagTrans = (Geom::Transformation.translation(diagVect)).invert!

pPointInit = corners[0]

# puts init time
#puts "Init: "+(Time.now-t1).to_s
t1 = Time.now
# Get all faces
faces = []
```

```
for ii in dGroup.entities
            if ii.class == Sketchup::Face
        faces << ii
    end
end

#puts "Init Faces: "+(Time.now-t1).to_s
t1 = Time.now

binCubes = Array.new(nx).map!{ Array.new(ny).map!{ Array.new(nz) } }
intDist =  Array.new(ny).map!{ Array.new(nz).map!{ Array.new() } }

nyVect = yVect.normalize.to_a
nzVect = zVect.normalize.to_a
orig = corners[0].to_a
frontPlane = [corners[0], xVect]

for ii in faces
    miny = ny-1
    maxy = 0
    minz = nz-1
    maxz = 0

    for vv in ii.vertices
            vertVect = orig.vector_to vv.position
        yDist = nyVect.dot(vertVect)/vSizeLength
        zDist = nzVect.dot(vertVect)/vSizeLength
        # puts yDist
        if yDist<miny then miny = yDist end
        if yDist>maxy then maxy = yDist end
        if zDist<minz then minz = zDist end
        if zDist>maxz then maxz = zDist end
    end # vv

    #rounding of min and max values
    miny = miny.floor
            maxy = maxy.ceil
            minz = minz.floor
            maxz = maxz.ceil
```

```
    ns = 50
    for jj in miny..maxy
    yScale = Geom::Transformation.scaling(jj+0.5)
pPointInit1 = corners[0].offset(yVect.transform(yScale))

for kk in minz..maxz
    zScale = Geom::Transformation.scaling(kk+0.5)
    pPointInit2 = pPointInit1.offset(zVect.transform(zScale))

    curPoint = Geom.intersect_line_plane([pPointInit2,xVect],ii.
        plane)

    # Ignore faces parallel to probing vector
                next if curPoint == nil

                # Check where point is relative to face
                cPoint = ii.classify_point(curPoint)

                # Point is on face, good
                if (cPoint == Sketchup::Face::PointInside)
                        intDist[jj][kk] << pPointInit2.
                            distance(curPoint)
                        next
                # If point isn't on edge or vertex, go to
                    next
                elsif ((cPoint != Sketchup::Face::
                    PointOnVertex) and
                        (cPoint != Sketchup::Face::
                            PointOnEdge))
                        next
                end # if

                # If point is on edge or vertex, begin spiral
                     algorithm
                for ss in 1..ns
                        # shift probing point around a spiral
                        sPoint = pPointInit2.offset(yVect.
                            to_a.collect{|x| x*ss/ns*Math.cos
```

```
                                        (Math::PI*2*ss/ns)})
                                    sPoint.offset!(zVect.to_a.collect{|x|
                                         x*ss/ns*Math.sin(Math::PI*2*ss/
                                        ns)})

                                    curPoint = Geom.intersect_line_plane
                                        ([sPoint,xVect],ii.plane)

                                    # Try to hit the face again
                                    cPoint = ii.classify_point(curPoint)

                                    # Probing vector hit the face!
                                    if (cPoint == Sketchup::Face::
                                        PointInside)
                                            intDist[jj][kk] <<
                                                pPointInit2.distance(
                                                curPoint)
                                            break
                                    # Probing vector hit a vertex or edge
                                         again, continue spiral
                                    elsif ((cPoint == Sketchup::Face::
                                        PointOnVertex) or
                                            (cPoint == Sketchup::Face::
                                                PointOnEdge))
                                            next
                                    # Probing vector does not intersect
                                        face
                                            else
                                                    break
                                    end # if
                            end # ss
                    end # kk
            end # jj
    end # ii

    for jj in 0..(ny-1)
        for kk in 0..(nz-1)
                # Solid is not intersected
                        next if intDist[jj][kk].empty?
```

```
                    # Sort internal distances
            intDist[jj][kk].sort!

            # If inside the solid, cube should be made
            inside = false
            for ii in 0..(nx-1)
                pos = xVect.length*(ii+0.5)

                # Has solid been intersected?
                for nDist in intDist[jj][kk]
                        if pos>=nDist
                                intDist[jj][kk] = intDist[jj][kk][1..-1]
                                inside = !inside
                        else
                                break
                        end
                end # nDist
                # Go to next iteration if not inside solid
                # if not inside then next end
                                next if not inside
                # Inside solid: cube should be made
                binCubes[ii][jj][kk] = 1
                end # ii
                end # kk
    end # jj

        #puts "Get binCubes: "+(Time.now-t1).to_s

        t1 = Time.now
        if isProtoDrop then
                fileNameString = 'C:\Users\eldrb2\Personal␣Documents␣-␣Not␣
                    Backed␣Up\MyDropBox\Dropbox\Uni\PhD\Development\Sketchup␣
                    Files\proto' << fCount.to_s
                fileNameString <<= '.txt'

                f = File.open(fileNameString, 'a')
                # write the voxel array dimensions to file
                f.puts nx
```

```
                f.puts ny
                f.puts nz
                f.puts "\n"
                for jj in 0..(ny-1)
                        for kk in 0..(nz-1)
                                for ii in 0..(nx-1)
                                        if (binCubes[ii][jj][kk] == nil)
                                                f.puts "0"
                                        else
                                                f.puts (binCubes[ii][jj][kk].
                                                    to_s)
                                        end # if
                                end # ii
                        end # kk
                end # jj
                f.close
                #puts "Write binCubes to file: "+(Time.now-t1).to_s
        end

    t1 = Time.now
        cubes = nil
        if shouldDrawCubes then
        # Draw cubes
                gTrans = dGroup.transformation
        cubes = drawcubes(binCubes, corners[0],[xVect,yVect,zVect],diagTrans,
            gTrans)
        puts "Draw␣Cubes:␣"+(Time.now-t1).to_s
        t1 = Time.now
    end
    # End of operation
    model.commit_operation

end

### ----------------------------------------------------------------- ###
#
#   cubecomp()
#
#   Make a cube component.
```

```
#
def cubecomp(x,y,z)

        model = Sketchup.active_model
    entities = model.entities
    selection = model.selection

    orig = ORIGIN.clone
    pts = []
    pts[0] = orig
    pts[3] = orig + x
    pts[1] = orig + y
    pts[2] = orig + x + y

        cubeGroup = entities.add_group
        ents = cubeGroup.entities
    cubeface = ents.add_face pts
    cubeface.pushpull -z.length, true
                        cubeface.reverse!
    cubeinst = cubeGroup.to_component
    cubedef = cubeinst.definition
    entities.erase_entities cubeinst
    return cubedef
end

### ----------------------------------------------------------------- ###
#
#  drawcubes()
#
def drawcubes(binCubes, pPointInit, vect, diagTrans, gTrans)

    model = Sketchup.active_model
    entities = model.entities
        selection = model.selection

    cubedef = cubecomp(vect[0],vect[1],vect[2])

    n = [binCubes.length, binCubes[0].length, binCubes[0][0].length]
    cubes = []
```

```
    for jj in 0..(n[1]-1)
        yScale = Geom::Transformation.scaling(jj+0.5)
        pPointInit1 = pPointInit.offset(vect[1].transform(yScale))

        for kk in 0..(n[2]-1)
                zScale = Geom::Transformation.scaling(kk+0.5)
                pPointInit2 = pPointInit1.offset(vect[2].transform(zScale))

                for ii in 0..(n[0]-1)

                if binCubes[ii][jj][kk] == 1
                        xScale = Geom::Transformation.scaling(ii+0.5)
                        pPoint = pPointInit2.offset(vect[0].transform(xScale)
                            )
                        trans = Geom::Transformation.translation pPoint.to_a
                        cubes << entities.add_instance(cubedef, trans)
                end # if
                end # ii
        end # kk
    end # jj
    entities.transform_entities((gTrans*diagTrans), cubes)
    return cubes
end
```

Appendix B

3-D SCN TLM solver documentation

3D-TLM

1.0

Generated by Doxygen 1.8.5

Tue Feb 18 2014 09:56:03

Contents

1 3DTLM **1**

2 Class Index **3**

2.1 Class List 3

3 Class Documentation **5**

3.1 Field Class Reference 5

3.1.1 Detailed Description 6

3.2 GaussianPulse Class Reference 6

3.2.1 Detailed Description 7

3.2.2 Constructor & Destructor Documentation 7

3.2.2.1 GaussianPulse 7

3.2.3 Member Function Documentation 7

3.2.3.1 normalDistWithX 7

3.2.3.2 updatePulse 7

3.3 MeasurementPoint Class Reference 8

3.3.1 Detailed Description 9

3.3.2 Constructor & Destructor Documentation 9

3.3.2.1 MeasurementPoint 9

3.3.2.2 MeasurementPoint 9

3.3.3 Member Function Documentation 9

3.3.3.1 getFreqStart 9

3.3.3.2 getFreqStep 10

3.3.3.3 getFreqStop 10

3.3.3.4 outputNodeListToFile 10

3.3.3.5 recordNodeState 10

3.4 MeshPoint Struct Reference 10

3.4.1 Detailed Description 11

3.5 Pixel Struct Reference 11

3.5.1 Detailed Description 12

3.6 PointSource Class Reference 12

3.6.1 Detailed Description 12

3.6.2 Constructor & Destructor Documentation 12
3.6.2.1 PointSource 13
3.6.3 Member Function Documentation 14
3.6.3.1 updatePointSource 14
3.7 ProtoDrop Class Reference 14
3.7.1 Detailed Description 15
3.7.2 Constructor & Destructor Documentation 15
3.7.2.1 ProtoDrop 15
3.8 Raindrop Class Reference 15
3.8.1 Detailed Description 16
3.9 ScatterVariables Struct Reference 16
3.9.1 Detailed Description 18
3.10 SCNNode Class Reference 18
3.10.1 Detailed Description 20
3.11 SimulationConstants Struct Reference 20
3.11.1 Detailed Description 21
3.12 Solver Class Reference 21
3.12.1 Detailed Description 24
3.12.2 Member Function Documentation 24
3.12.2.1 calcMaxTimestep 24
3.12.2.2 calcMeshCutoff 24
3.12.2.3 calcStubSCNParams 24
3.12.2.4 configMeshExcitation 24
3.12.2.5 configMeshObservationType 25
3.12.2.6 connectMesh 25
3.12.2.7 convertSubPixelValuesToCharArray 26
3.12.2.8 createFDBinFiles 26
3.12.2.9 createLineBoundary 26
3.12.2.10 createMeshEdgeBoundaries 27
3.12.2.11 createSingleTDMeasurementPoint 27
3.12.2.12 createXYBoundaryFaceRect 28
3.12.2.13 createXZGaussianPlanewave 28
3.12.2.14 createXZMeasurementPointsPlane 29
3.12.2.15 createYZBoundaryFaceRect 29
3.12.2.16 exciteMeshWithFieldAtCoordinates 30
3.12.2.17 implementBoundaryNode 30
3.12.2.18 importRaindropGeometryFromFile 31
3.12.2.19 importSimulationConfigParameters 31
3.12.2.20 initMeshNodesProperties 32
3.12.2.21 initMPIChannels 32

3.12.2.22 initRaindrops 32
3.12.2.23 itoa 32
3.12.2.24 meshSingleAbsorber 33
3.12.2.25 numberOfDigits 34
3.12.2.26 outputMeasurements 34
3.12.2.27 outputMeshToPPMFile 34
3.12.2.28 outputSingleTDMeasurementPointLog 35
3.12.2.29 outputZSlicesForTimeStep 35
3.12.2.30 roundToNearestInt 36
3.12.2.31 setMeshPoint 36
3.12.2.32 specMap 37
3.12.2.33 swapInts 37
3.12.2.34 updateMeasurementPoints 37
3.12.2.35 updateMeasurements 38
3.12.2.36 updateSingleTDMeasurementPoint 38
3.13 StubSCNProperties Struct Reference 39
3.13.1 Detailed Description 40

Index **41**

Chapter 1

3DTLM

A 3D Computational Electromagnetics Solver designed for rainfall simulations.

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Field
A class describing a set of field components at a particular node location 5
GaussianPulse
A class describing a Gaussian point-source pulse event 6
MeasurementPoint
A class describing a particular node to observe within the mesh 8
MeshPoint
A struct representing the location of a node within the mesh 10
Pixel
A struct representing a pixel onscreen 11
PointSource
A class describing a point-source of energy to be imparted to the mesh at a particular location 12
ProtoDrop
A class to describe a prototype raindrop, including its geometry information 14
Raindrop
A class to describe a simulated raindrop, including its size and location information 15
ScatterVariables
A struct storing temporary variables required for use during the scatter process 16
SCNNode
A class representing a SCN, a 3-D node for TLM 18
SimulationConstants
A struct storing mesh node dimensions, free-space and water timestep values, and free-space and water max frequency values 20
Solver
A class which conducts all operations concerning simulation execution 21
StubSCNProperties
A struct describing the stub properties for all the SCNs in the mesh 39

Chapter 3

Class Documentation

3.1 Field Class Reference

A class describing a set of field components at a particular node location.

```
#include <Field.h>
```

Collaboration diagram for Field:

Field
+ Ex
+ Ey
+ Ez
+ Hx
+ Hy
+ Hz
+ Field()
+ ~Field()

Public Attributes

- float Ex

    The x-directed electric field component.
- float Ey

    The y-directed electric field component.
- float Ez

    The z-directed electric field component.
- float Hx

    The x-directed magnetic field component.
- float Hy

    The y-directed magnetic field component.
- float Hz

The z-directed magnetic field component.

3.1.1 Detailed Description

A class describing a set of field components at a particular node location.

The documentation for this class was generated from the following files:

- 3D-TLM/Field.h
- 3D-TLM/Field.cpp

3.2 GaussianPulse Class Reference

A class describing a Gaussian point-source pulse event.

```
#include <GaussianPulse.h>
```

Collaboration diagram for GaussianPulse:

GaussianPulse

+ x
+ y
+ z
+ hasPulseEnded

+ GaussianPulse()
+ GaussianPulse()
+ updatePulse()
+ normalDistWithX()
+ ~GaussianPulse()

Public Member Functions

- GaussianPulse (int, int, int, int, float, int)
- Field * updatePulse (void)
- float normalDistWithX (float, float, float)

Public Attributes

- int x

  The x-axis co-ordinate position of the Gaussian pulse.
- int y

  The y-axis co-ordinate position of the Gaussian pulse.
- int z

  The z-axis co-ordinate position of the Gaussian pulse.
- bool hasPulseEnded

  A Boolean flag to determine if the Gaussian pulse has ended.

3.2.1 Detailed Description

A class describing a Gaussian point-source pulse event.

3.2.2 Constructor & Destructor Documentation

3.2.2.1 GaussianPulse::GaussianPulse (int *xPos*, int *yPos*, int *zPos*, int *waveLength*, float *amplitude*, int *excitedFields*)

Custom constructor for gaussian pulse object.

Parameters

xPos	X position in mesh (nodes).
yPos	Y position in mesh (nodes).
zPos	Z position in mesh (nodes).
waveLength	Wavelength of the pulse (nodes).
amplitude	Maximum amplitude of the pulse (volts).
excitedFields	Which field components should be excited.
∗*field*	Pointer to a field object.

3.2.3 Member Function Documentation

3.2.3.1 float GaussianPulse::normalDistWithX (float *x*, float *mean*, float *stdDev*)

Generate a point along a Gaussian bell curve.

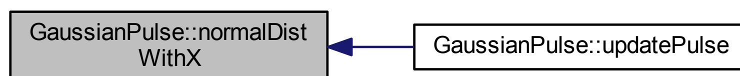
Parameters

x	X value.
mean	The mean value for the distribution.
stdDev	The standard deviation value for the distribution.

Returns

Returns the Y-axis bell-curve value for the given x.

Here is the caller graph for this function:



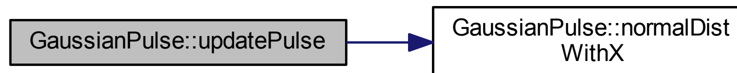
3.2.3.2 Field ∗ GaussianPulse::updatePulse (void)

Update the pulse excitation value Update the pulse excitation value based on the current timestep since start, then inject the pulse into the mesh at relevant location.

Returns

∗f Returns a pointer to a Field object.

Here is the call graph for this function:



The documentation for this class was generated from the following files:

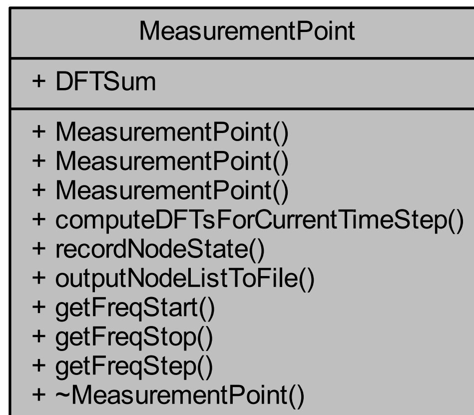
- 3D-TLM/GaussianPulse.h
- 3D-TLM/GaussianPulse.cpp

3.3 MeasurementPoint Class Reference

A class describing a particular node to observe within the mesh.

`#include <MeasurementPoint.h>`

Collaboration diagram for MeasurementPoint:



Public Member Functions

- MeasurementPoint (unsigned int, float, float, float, double, SCNNode ∗)
- MeasurementPoint (unsigned int, SCNNode ∗)
- void computeDFTsForCurrentTimeStep (void)

  Compute the DFT of the observed node (upto the current timestep), for the field components set in 'DFTMode'.

- void recordNodeState (void)
- void outputNodeListToFile (string)
- double getFreqStart (void)
- double getFreqStop (void)
- double getFreqStep (void)

Public Attributes

- vector< complex< double > > DFTSum

  A vector array of running sum values for the calculated DFTs.

3.3.1 Detailed Description

A class describing a particular node to observe within the mesh.

3.3.2 Constructor & Destructor Documentation

3.3.2.1 MeasurementPoint::MeasurementPoint (unsigned int *mode,* float *fStart,* float *fStop,* float *fStep,* double *tStep,* SCNNode ∗ *nodePointer*)

Custom constructor to assign DFT mode, observed harmonic, and a pointer to the node to observe.

Parameters

mode	Which field components to use in DFT calculations.
fStart	Start frequency for DFT calculations.
fStop	Stop frequency for DFT calculations.
fStep	Frequency step for DFT calculations.
tStep	The time step value that has been configured already for the simulation.
nodePointer	A Pointer to the node to observe.

3.3.2.2 MeasurementPoint::MeasurementPoint (unsigned int *mode,* SCNNode ∗ *nodePointer*)

Custom constructor to assign DFT mode, observed harmonic, and a pointer to the node to observe.

Parameters

mode	Which field components to use in DFT calculations.
nodePointer	A Pointer to the node to observe.

3.3.3 Member Function Documentation

3.3.3.1 double MeasurementPoint::getFreqStart (void)

Returns the frequency start value

Returns

The freqStart value.

3.3.3.2 double MeasurementPoint::getFreqStep (void)

Returns the frequency step value

Returns

The freqStep value.

3.3.3.3 double MeasurementPoint::getFreqStop (void)

Returns the frequency stop value

Returns

The freqStop value.

3.3.3.4 void MeasurementPoint::outputNodeListToFile (string *fileName*)

Outputs the nodeList data to a file for analysis.

Parameters

fileName	The file name to use as the log file name.

3.3.3.5 void MeasurementPoint::recordNodeState (void)

Add a node to the vector list of nodes.

Parameters

node	Node to add to the list.

The documentation for this class was generated from the following files:

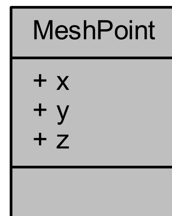
- 3D-TLM/MeasurementPoint.h
- 3D-TLM/MeasurementPoint.cpp

3.4 MeshPoint Struct Reference

A struct representing the location of a node within the mesh.

```
#include <Solver.h>
```

Collaboration diagram for MeshPoint:



Public Attributes

- int x
  X-axis location (in nodes).
- int y
  Y-axis location (in nodes).
- int z
  Z-axis location (in nodes).

3.4.1 Detailed Description

A struct representing the location of a node within the mesh.

The documentation for this struct was generated from the following file:

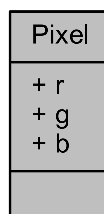
- 3D-TLM/Solver.h

3.5 Pixel Struct Reference

A struct representing a pixel onscreen.

```
#include <Solver.h>
```

Collaboration diagram for Pixel:



Public Attributes

- char r

    Red component value (0-255).
- char g

    Green component value (0-255).
- char b

    Blue component value (0-255).

3.5.1 Detailed Description

A struct representing a pixel onscreen.

The documentation for this struct was generated from the following file:

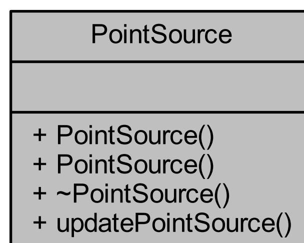
- 3D-TLM/Solver.h

3.6 PointSource Class Reference

A class describing a point-source of energy to be imparted to the mesh at a particular location.

```
#include <PointSource.h>
```

Collaboration diagram for PointSource:



Public Member Functions

- PointSource (int, int, int, float, int, double)
- void updatePointSource (int)

3.6.1 Detailed Description

A class describing a point-source of energy to be imparted to the mesh at a particular location.

3.6.2 Constructor & Destructor Documentation

3.6.2.1 PointSource::PointSource (int *x*, int *y*, int *z*, float *sourceFrequency*, int *amplitudeMultiplier*, double *timeStepValue*)

Custom constructor for a sinusoidal point source pulse.

Parameters

x	X position of the point source in the mesh (nodes).
y	Y position of the point source in the mesh (nodes).
z	Z position of the point source in the mesh (nodes).
source-Frequency	Frequency of the sinusoidal wave emitted from the point source.
amplitude-Multiplier	The amount by which to increase the default amplitude of the pulse.
timeStepValue	The current amount of real-world time that has passed since the start of the simulation.

3.6.3 Member Function Documentation

3.6.3.1 void PointSource::updatePointSource (int *timeStepNumber*)

Update the value of the field being emitted by the point source.

Computes the value along the sine wave to allow the point source to emit the correct field.

Parameters

timeStepNumber	The current time step number since simulation start.

The documentation for this class was generated from the following files:

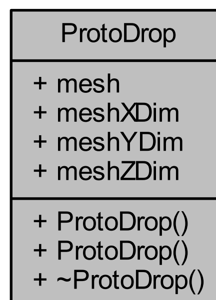
- 3D-TLM/PointSource.h
- 3D-TLM/PointSource.cpp

3.7 ProtoDrop Class Reference

A class to describe a prototype raindrop, including its geometry information.

`#include <ProtoDrop.h>`

Collaboration diagram for ProtoDrop:



Public Member Functions

- ProtoDrop (float, int, int, int)

Public Attributes

- vector< SCNNode > mesh

  A vector containing the mesh information for the prototype raindrop.

- int meshXDim

  The number of mesh nodes in the x-axis direction.

- int meshYDim

  The number of mesh nodes in the y-axis direction.

- int meshZDim

  The number of mesh nodes in the z-axis direction.

3.7.1 Detailed Description

A class to describe a prototype raindrop, including its geometry information.

3.7.2 Constructor & Destructor Documentation

3.7.2.1 ProtoDrop::ProtoDrop (float *dropD*, int *meshX*, int *meshY*, int *meshZ*)

Custom constructor for a prototype drop.

Parameters

dropD	The raindrop diameter.
meshX	The X position in the mesh (nodes).
meshY	The Y position in the mesh (nodes).
meshZ	The Z position in the mesh (nodes).

The documentation for this class was generated from the following files:

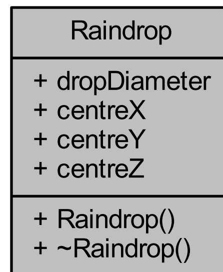
- 3D-TLM/ProtoDrop.h
- 3D-TLM/ProtoDrop.cpp

3.8 Raindrop Class Reference

A class to describe a simulated raindrop, including its size and location information.

```
#include <Raindrop.h>
```

Collaboration diagram for Raindrop:



Public Attributes

- float **dropDiameter**
- double **centreX**
- double **centreY**
- double **centreZ**

3.8.1 Detailed Description

A class to describe a simulated raindrop, including its size and location information.

The documentation for this class was generated from the following files:

- 3D-TLM/Raindrop.h
- 3D-TLM/Raindrop.cpp

3.9 ScatterVariables Struct Reference

A struct storing temporary variables required for use during the scatter process.

```
#include <Solver.h>
```

Collaboration diagram for ScatterVariables:

ScatterVariables

+ Vx
+ Vy
+ Vz
+ ZIx
+ ZIy
+ ZIz
+ tmpVxpz
+ tmpVxpy
+ tmpVxnz
+ tmpVxny
+ tmpVypx
+ tmpVypz
+ tmpVynx
+ tmpVynz
+ tmpVzpx
+ tmpVzpy
+ tmpVznx
+ tmpVzny
+ tmpVoxYox
+ tmpVoyYoy
+ tmpVozYoz
+ tmpVsx
+ tmpVsy
+ tmpVsz

Public Attributes

- float Vx

  Equivalent total voltages along each axis.
- float **Vy**
- float **Vz**
- float ZIx

  Product of link-line impedance and equivalent total current.
- float **ZIy**
- float **ZIz**
- float tmpVxpz

  Temporary copies of port voltages.
- float **tmpVxpy**
- float **tmpVxnz**
- float **tmpVxny**
- float **tmpVypx**
- float **tmpVypz**
- float **tmpVynx**
- float **tmpVynz**

- float **tmpVzpx**
- float **tmpVzpy**
- float **tmpVznx**
- float **tmpVzny**
- float tmpVoxYox

  Temporary copies of the open-circuitproducts of stub voltages and their respective normalised admittance.

- float **tmpVoyYoy**
- float **tmpVozYoz**
- float tmpVsx

  Temporary copies of short-circuit stub voltages.

- float **tmpVsy**
- float **tmpVsz**

3.9.1 Detailed Description

A struct storing temporary variables required for use during the scatter process.

The documentation for this struct was generated from the following file:

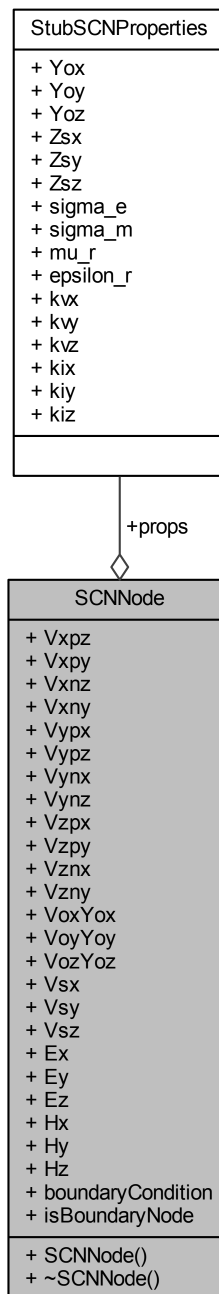
- 3D-TLM/Solver.h

3.10 SCNNode Class Reference

A class representing a SCN, a 3-D node for TLM.

```
#include <SCNNode.h>
```

Collaboration diagram for SCNNode:



Public Attributes

- float Vxpz

    X-axis port voltages.
- float **Vxpy**
- float **Vxnz**
- float **Vxny**

- float Vypx

  Y-axis port voltages.
- float **Vypz**
- float **Vynx**
- float **Vynz**
- float Vzpx

  Z-axis port voltages.
- float **Vzpy**
- float **Vznx**
- float **Vzny**
- float VoxYox

  Open-circuit products of stub voltages and their respective normalised admittance (convience variables instead of repeating stuff later)
- float **VoyYoy**
- float **VozYoz**
- float Vsx

  Short-circuit stub voltages.
- float **Vsy**
- float **Vsz**
- float Ex

  Electric-field components.
- float **Ey**
- float **Ez**
- float Hx

  Magnetic-field components.
- float **Hy**
- float **Hz**
- float **boundaryCondition**
- bool **isBoundaryNode**
- StubSCNProperties * props

  Pointer to a struct holding simulation properties.

3.10.1 Detailed Description

A class representing a SCN, a 3-D node for TLM.

The documentation for this class was generated from the following files:

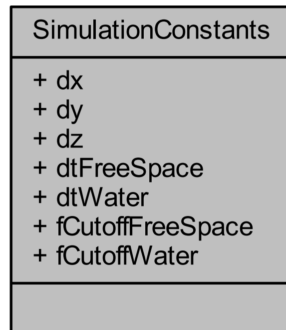
- 3D-TLM/SCNNode.h
- 3D-TLM/SCNNode.cpp

3.11 SimulationConstants Struct Reference

A struct storing mesh node dimensions, free-space and water timestep values, and free-space and water max frequency values.

```
#include <Solver.h>
```

Collaboration diagram for SimulationConstants:



Public Attributes

- double dx

  X-axis node size (in metres).
- double dy

  Y-axis node size (in metres).
- double dz

  Z-axis node size (in metres).
- double dtFreeSpace

  Free-space timestep value (in seconds).
- double dtWater

  Water-based timestep value (in seconds).
- double fCutoffFreeSpace

  Free-space cutoff frequency (in Hertz).
- double fCutoffWater

  Water-based cutoff frequency (in Hertz).

3.11.1 Detailed Description

A struct storing mesh node dimensions, free-space and water timestep values, and free-space and water max frequency values.

The documentation for this struct was generated from the following file:

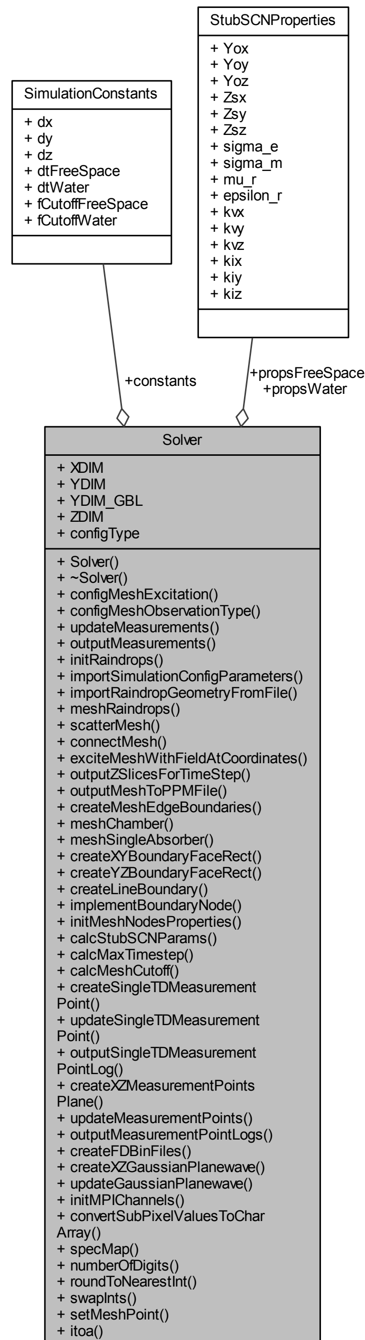
- 3D-TLM/Solver.h

3.12 Solver Class Reference

A class which conducts all operations concerning simulation execution.

```
#include <Solver.h>
```

Collaboration diagram for Solver:



Public Member Functions

- void configMeshExcitation (void)
- void configMeshObservationType (int, float, float, float)
- void updateMeasurements (int)
- void outputMeasurements (int)
- void initRaindrops (void)

- void importSimulationConfigParameters (int, int)
- void importRaindropGeometryFromFile (string, int)
- void meshRaindrops (void)

  Place the simulated raindrops imported previously into the mesh.
- void scatterMesh (void)

  Run the 'scatter' process of the TLM algorithm.
- void connectMesh (bool)
- void exciteMeshWithFieldAtCoordinates (Field ∗, int, int, int)
- void outputZSlicesForTimeStep (int, bool)
- void outputMeshToPPMFile (int, int, bool)
- void createMeshEdgeBoundaries (float, float, float, float, float, float)
- void meshChamber (void)

  Initialise a replica of the Holywell Park Anechoic Chamber.
- void meshSingleAbsorber (MeshPoint, MeshPoint, int, int, float)
- void createXYBoundaryFaceRect (MeshPoint, MeshPoint, float)
- void createYZBoundaryFaceRect (MeshPoint, MeshPoint, float)
- void createLineBoundary (MeshPoint, MeshPoint, float, int)
- void implementBoundaryNode (int, int, int)
- void initMeshNodesProperties (void)
- int calcStubSCNParams (StubSCNProperties ∗, double)
- double calcMaxTimestep (StubSCNProperties ∗)
- double calcMeshCutoff (StubSCNProperties ∗)
- void createSingleTDMeasurementPoint (void)
- void updateSingleTDMeasurementPoint (void)
- void outputSingleTDMeasurementPointLog (void)
- void createXZMeasurementPointsPlane (int, int, int, int, int, float, float, float)
- void updateMeasurementPoints (void)
- void outputMeasurementPointLogs (void)

  Write out the measurement point information to a log file.
- void createFDBinFiles (int, int)
- void createXZGaussianPlanewave (int, int, int, int, int)
- void updateGaussianPlanewave (void)

  Inject the most recently calculated value of the Gaussian pulse into the mesh.
- void initMPIChannels (bool)
- char ∗ convertSubPixelValuesToCharArray (int, bool)
- Pixel specMap (float)
- int numberOfDigits (const int)
- int roundToNearestInt (double)
- void swapInts (int ∗, int ∗)
- void setMeshPoint (MeshPoint ∗, int, int, int)
- string itoa (int)

Public Attributes

- int **XDIM**
- int **YDIM**
- int **YDIM_GBL**
- int **ZDIM**
- SimulationConstants **constants**
- StubSCNProperties **propsWater**
- StubSCNProperties **propsFreeSpace**
- string **configType**

3.12.1 Detailed Description

A class which conducts all operations concerning simulation execution.

3.12.2 Member Function Documentation

3.12.2.1 double Solver::calcMaxTimestep (StubSCNProperties ∗ *props*)

Calculate the maximum timestep value for the mesh. Calculates the maximum timestep value for material that keeps all stubs positive...

Parameters

∗*props*	A pointer to a StubSCNProperties struct.

Returns

dt The maximum save timestep value for the mesh to ensure stability.

3.12.2.2 double Solver::calcMeshCutoff (StubSCNProperties ∗ *props*)

Calculate the mesh cutoff frequency to ensure accuracy of results.

Parameters

∗*props*	A pointer to a StubSCNProperties struct.

Returns

fcutoff The cutoff frequency for the simulation.

3.12.2.3 int Solver::calcStubSCNParams (StubSCNProperties ∗ *props*, double *dt*)

Calculate the mesh node stub parameter values. These stub values are dependent on the calculated timestep value.

Parameters

∗*props*	A pointer to a StubSCNProperties struct.
dt	The calculated timestep value for the mesh.

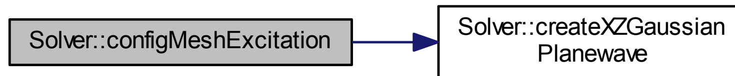
Returns

Success on completetion.

3.12.2.4 void Solver::configMeshExcitation (void)

Configures the mesh excitation Exciation scheme dependent on simulation being executed (e.g. rainfield, rectangular chamber, tapered chamber).

Here is the call graph for this function:



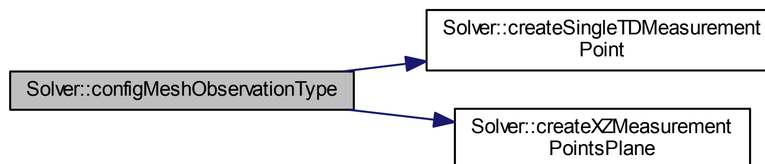
3.12.2.5 void Solver::configMeshObservationType (int *measMode*, float *fStart*, float *fStep*, float *fStop*)

Configures the mesh observation type Measurement type (time or frequency domain) is dependent on user input at start of program execution.

Parameters

measMode	The measurement mode requested by the user.
fStart	Start frequency for DFT calculations.
fStop	Stop frequency for DFT calculations.
fStep	Frequency step for DFT calculations.

Here is the call graph for this function:



3.12.2.6 void Solver::connectMesh (bool *shouldWrapAround*)

Run the 'connect' process of the TLM algorithm.

Parameters

shouldWrap-Around	A flag to decided whether periodic boundaries should be used.

Here is the call graph for this function:

3.12.2.7 char * Solver::convertSubPixelValuesToCharArray (int *z*, bool *drawBoundaries*)

Convert an entire XY-slice through the mesh into RGB pixel values.

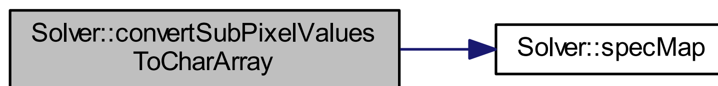
Parameters

z	The Z-axis offset for the XY-slice through the mesh.
drawBoundaries	A flag to decide whether to visualise boundary nodes as white pixels.

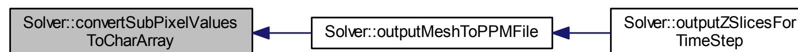
Returns

RGBSliceBuffer A 1-D char array representing the XY-slice pixel values.

Here is the call graph for this function:



Here is the caller graph for this function:



3.12.2.8 void Solver::createFDBinFiles (int *measMode*, int *rank*)

Reformats the Frequency-domain measurement data. Seperates the original FD data file into seperate TXT files for each bin (for both Magnitude and Phase).

Here is the call graph for this function:

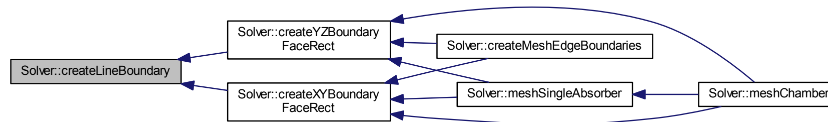
3.12.2.9 void Solver::createLineBoundary (MeshPoint *stPoint*, MeshPoint *ePoint*, float *cond*, int *mode*)

Initialise a straight-line boundary with a particular condition.

Parameters

stPoint	The start point of the line (in nodes).
ePoint	The end point of the line (in nodes).
cond	The boundary condition for the line boundary.
mode	Decides which plane to operate the line-drawing algorithm in.

Here is the caller graph for this function:



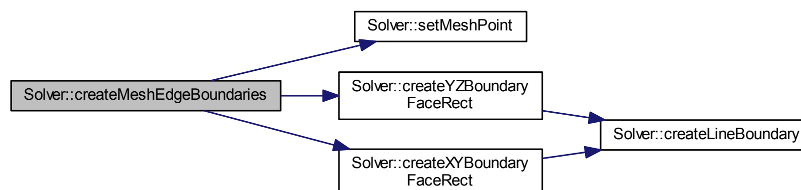
3.12.2.10 void Solver::createMeshEdgeBoundaries (float *topCond*, float *bottomCond*, float *leftCond*, float *rightCond*, float *frontCond*, float *backCond*)

Initialise all the mesh-edge boundaries

Parameters

topCond	The boundary condition for the top of the mesh.
bottomCond	The boundary condition for the bottom of the mesh.
leftCond	The boundary condition for the left of the mesh.
rightCond	The boundary condition for the right of the mesh.
frontCond	The boundary condition for the front of the mesh.
backCond	The boundary condition for the back of the mesh.

Here is the call graph for this function:



3.12.2.11 void Solver::createSingleTDMeasurementPoint (void)

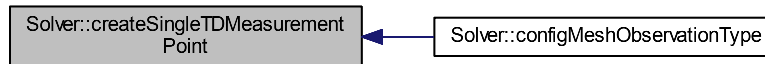
Initialise a single time-domain MeasurementPoint object.

Parameters

mode	The measurement mode for field observations.

nA	A pointer to a SCNNode.

Here is the caller graph for this function:



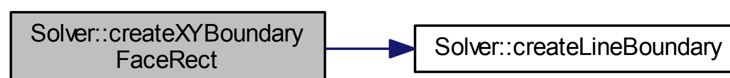
3.12.2.12 void Solver::createXYBoundaryFaceRect (MeshPoint *stPoint*, MeshPoint *ePoint*, float *cond*)

Initialise a boundary plane rectangle in the XY plane.

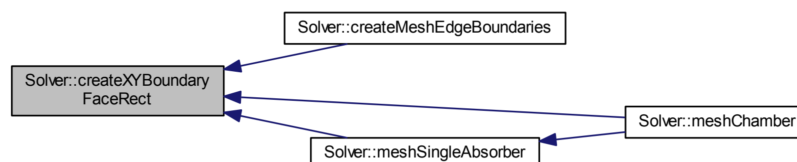
Parameters

stPoint	The start point of the base of the absorber (in nodes).
ePoint	The end point of the base of the absorber (in nodes).
cond	The boundary condition for the absorber material.

Here is the call graph for this function:



Here is the caller graph for this function:



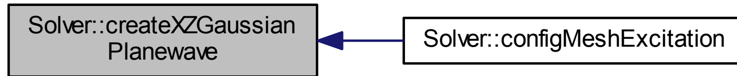
3.12.2.13 void Solver::createXZGaussianPlanewave (int *sX*, int *sZ*, int *eX*, int *eZ*, int *yPlane*)

Initialise a Gaussian planewave in the XZ-plane.

Parameters

sX	The start value along the X-axis (in terms of nodes).
sZ	The start value along the X-axis (in terms of nodes).
eX	The end value along the X-axis (in terms of nodes).
eZ	The end value along the X-axis (in terms of nodes).
yPlane	The Y-axis offset in which to place the XZ plane.

Here is the caller graph for this function:



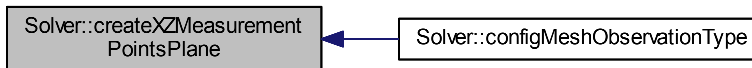
3.12.2.14 void Solver::createXZMeasurementPointsPlane (int *sX*, int *sZ*, int *eX*, int *eZ*, int *yPlane*, float *fStart*, float *fStep*, float *fStop*)

Initialise a plane of MeasurementPoints for DFT purposes in the XZ-plane.

Parameters

sX	The start value along the X-axis (in terms of nodes).
sZ	The start value along the X-axis (in terms of nodes).
eX	The end value along the X-axis (in terms of nodes).
eZ	The end value along the X-axis (in terms of nodes).
yPlane	The Y-axis offset in which to place the XZ plane.
fStart	Start frequency for DFT calculations.
fStop	Stop frequency for DFT calculations.
fStep	Frequency step for DFT calculations.

Here is the caller graph for this function:



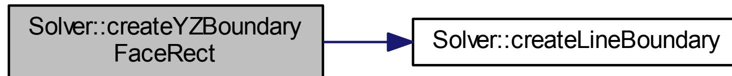
3.12.2.15 void Solver::createYZBoundaryFaceRect (MeshPoint *stPoint*, MeshPoint *ePoint*, float *cond*)

Initialise a boundary plane rectangle in the YZ plane.

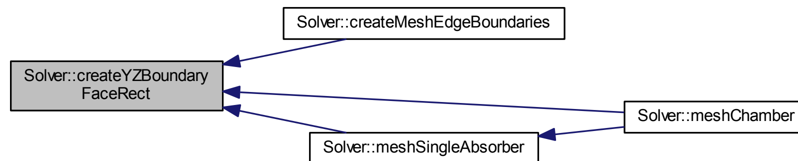
Parameters

stPoint	The start point of the base of the absorber (in nodes).
ePoint	The end point of the base of the absorber (in nodes).
cond	The boundary condition for the absorber material.

Here is the call graph for this function:



Here is the caller graph for this function:



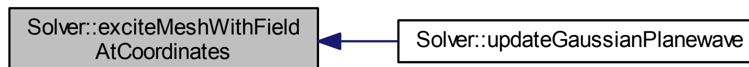
3.12.2.16 void Solver::exciteMeshWithFieldAtCoordinates (Field ∗ *f*, int *x*, int *y*, int *z*)

Excite the mesh at a particular node with a given field.

Parameters

∗*F*	A pointer to a Field object.
x	X-axis location of the node (specified in terms of nodes).
y	Y-axis location of the node (specified in terms of nodes).
z	Z-axis location of the node (specified in terms of nodes).

Here is the caller graph for this function:



3.12.2.17 void Solver::implementBoundaryNode (int *x*, int *y*, int *z*)

Applies the boundary node's condition at a particular location in the mesh.

Parameters

x	The X-axis location of the boundary node (specified in terms of nodes).
y	The Y-axis location of the boundary node (specified in terms of nodes).
z	The Z-axis location of the boundary node (specified in terms of nodes).

Here is the caller graph for this function:

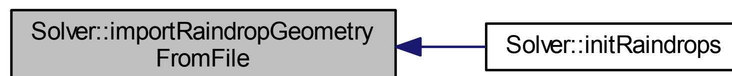
3.12.2.18 void Solver::importRaindropGeometryFromFile (string *fileName*, int *dropNum*)

Read-in a raindrop file and examine for drop-type. Import prototype raindrop geometry information or mesh simulated raindrop using prototype information depending on file type.

Parameters

fileName	The file name of the raindrop file to be imported.
dropNum	The number of the current drop to be imported.

Here is the caller graph for this function:



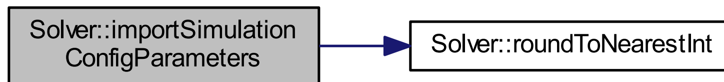
3.12.2.19 void Solver::importSimulationConfigParameters (int *numProcs*, int *rankProc*)

Import simulation parameters from the config.txt file.

Parameters

numProcs	The number of processors being used for program execution.

Here is the call graph for this function:



3.12.2.20 void Solver::initMeshNodesProperties (void)

Initialise the mesh node properties for the entire mesh. The properties are set based on the imported values from the config.txt file.

3.12.2.21 void Solver::initMPIChannels (bool *shouldWrapAround*)

Initialise the MPI communication channels for the 'Connect' process.

Parameters

shouldWrap-Around	A flag to decide if a perodic boundary condition should be applied.

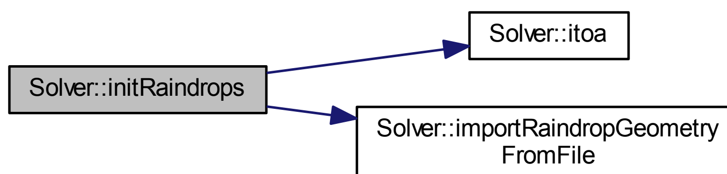
3.12.2.22 void Solver::initRaindrops (void)

Initialise all the raindrops into the mesh Import all the prototype raindrops into the solver, then initialise the raindrops to simulate based off imported location information.

Parameters

rankProc	The 'rank' number of the current processor being used for program execution.

Here is the call graph for this function:



3.12.2.23 string Solver::itoa (int *i*)

Create a string equivalent of an integer.

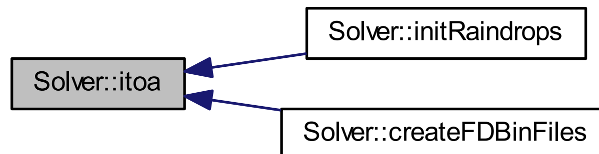
Parameters

i	The integer to convert to a string.

Returns

s The string equivalent of 'i'.

Here is the caller graph for this function:



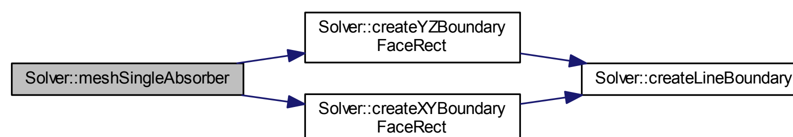
3.12.2.24 void Solver::meshSingleAbsorber (MeshPoint *stPoint*, MeshPoint *ePoint*, int *height*, int *absFace*, float *cond*)

Initialise a single anechoic absorber in the mesh.

Parameters

stPoint	The start point of the base of the absorber (in nodes).
ePoint	The end point of the base of the absorber (in nodes).
height	The height of the absorber (in nodes).
absFace	The particular face of a cuboid object to place the absorber onto.
cond	The boundary condition for the absorber material.

Here is the call graph for this function:



Here is the caller graph for this function:

3.12.2.25 int Solver::numberOfDigits (const int *number*)

Calculates the number of digits present in an integer number.

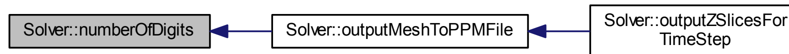
Parameters

number	The number to count the digits of.

Returns

The number of digits present in integer 'number'.

Here is the caller graph for this function:



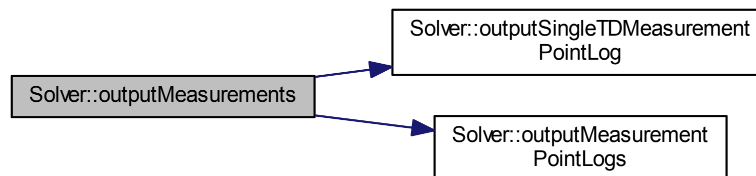
3.12.2.26 void Solver::outputMeasurements (int *measMode*)

Output the mesh measurements Measurements are output dependent on the user-requested measurement mode.

Parameters

measMode	The measurement mode requested by the user.

Here is the call graph for this function:



3.12.2.27 void Solver::outputMeshToPPMFile (int *timeStep,* int *zSlice,* bool *drawBoundaries*)

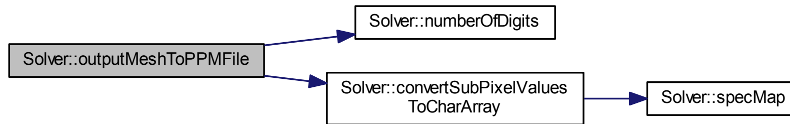
Output a given Z-axis slice to a PPM image file.

Parameters

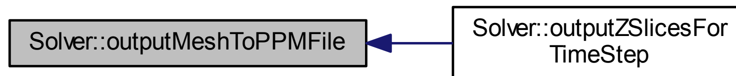
timeStep	The current timestep number.
zSlice	The current Z-axis offset for the image slice.

drawBoundaries	A flag to decide whether to draw boundaries as white pixels.

Here is the call graph for this function:



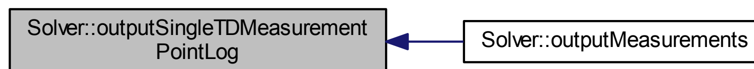
Here is the caller graph for this function:



3.12.2.28 void Solver::outputSingleTDMeasurementPointLog (void)

Output a record of the selected field information for every time-step of a simulation. Each field component that has been requested is written to a log file in plaintext format.

Here is the caller graph for this function:



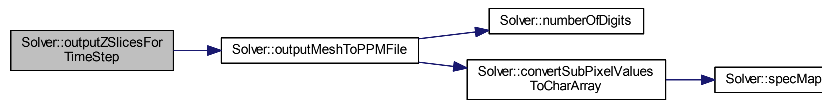
3.12.2.29 void Solver::outputZSlicesForTimeStep (int *timeStep*, bool *drawBoundaries*)

Write out all images for a given timestep. Outputs 1-node thick slices, spanning entire Z-axis range.

Parameters

timeStep	The current timestep number.
drawBoundaries	A flag to decide whether to draw boundaries as white pixels.

Here is the call graph for this function:



3.12.2.30 int Solver::roundToNearestInt (double *number*)

Rounds a double-precision float to the nearest integer.

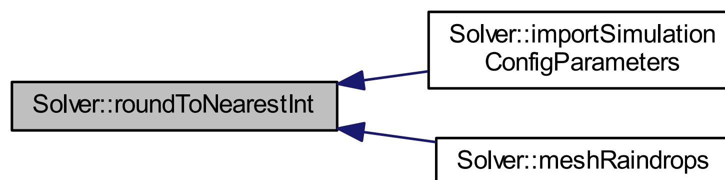
Parameters

number	The number to round.

Returns

The nearest integer to the number 'number'.

Here is the caller graph for this function:



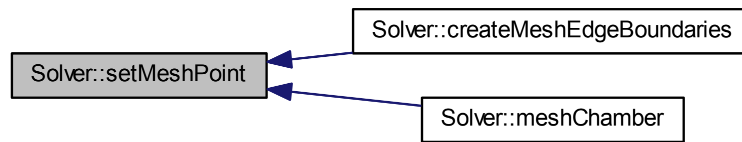
3.12.2.31 void Solver::setMeshPoint (MeshPoint * *p*, int *x*, int *y*, int *z*)

Initialises a MeshPoint object location within the mesh.

Parameters

**p*	A pointer to a MeshPoint object.
x	X-axis location of *p within the mesh (in terms of nodes).
y	Y-axis location of *p within the mesh (in terms of nodes).
z	Z-axis location of *p within the mesh (in terms of nodes).

Here is the caller graph for this function:



3.12.2.32 Pixel Solver::specMap (float *intensity*)

Convert a field intensity value into a set of RGB values along a colourmap. Out of range values are clipped for 8-bit per channel visualisation.

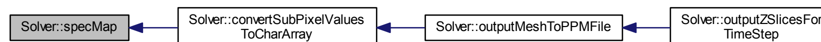
Parameters

intensity	The intensity value to convert.

Returns

p The pixel struct containing the RGB values to visualise.

Here is the caller graph for this function:



3.12.2.33 void Solver::swapInts (int * *a*, int * *b*)

Swap two integers.

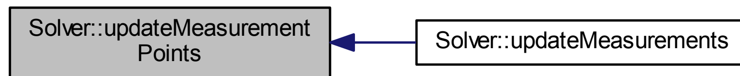
Parameters

**a*	A pointer to one integer.
**b*	A pointer to a second integer.

3.12.2.34 void Solver::updateMeasurementPoints (void)

Update the DFT value for each measurementPoint present. A DFT value is calculated for each frequency specified for each measurementPoint object.

Here is the caller graph for this function:



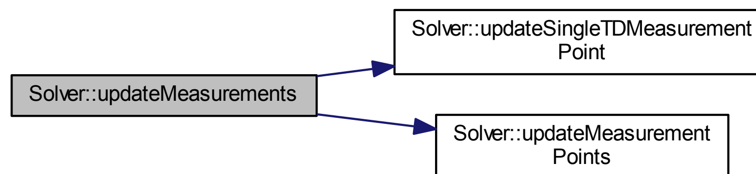
3.12.2.35 void Solver::updateMeasurements (int *measMode*)

Update the mesh measurements Measurements are updated dependent on the user-requested measurement mode.

Parameters

measMode	The measurement mode requested by the user.

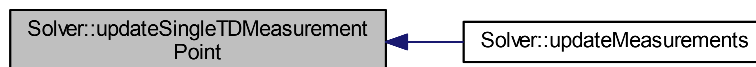
Here is the call graph for this function:



3.12.2.36 void Solver::updateSingleTDMeasurementPoint (void)

Update the single time-domain node. Adds a copy of the node to a <vector> holding a SCNNode for each time-step of the simulation.

Here is the caller graph for this function:



The documentation for this class was generated from the following files:

- 3D-TLM/Solver.h
- 3D-TLM/Solver.cpp

3.13 StubSCNProperties Struct Reference

A struct describing the stub properties for all the SCNs in the mesh.

```
#include <SCNNode.h>
```

Collaboration diagram for StubSCNProperties:

StubSCNProperties

+ Yox
+ Yoy
+ Yoz
+ Zsx
+ Zsy
+ Zsz
+ sigma_e
+ sigma_m
+ mu_r
+ epsilon_r
+ kvx
+ kvy
+ kvz
+ kix
+ kiy
+ kiz

Public Attributes

- float Yox

  X-directed normalised admittance.
- float Yoy

  Y-directed normalised admittance.
- float Yoz

  Z-directed normalised admittance.
- float Zsx

  X-directed short-circuit impedance.
- float Zsy

  Y-directed short-circuit impedance.
- float Zsz

  Z-directed short-circuit impedance.
- float sigma_e

  Sigma-e.
- float **sigma_m**
- float mu_r

  Relative permeability value.
- float epsilon_r

Relative permittivity value.

- float kvx

  X-directed constant for finding e.t.v ∗ sum of voltages.
- float kvy

  Y-directed constant for finding e.t.v ∗ sum of voltages.
- float kvz

  Z-directed constant for finding e.t.v ∗ sum of voltages.
- float kix

  X-directed constant for finding e.t.c ∗ link-line impedance.
- float kiy

  Y-directed constant for finding e.t.c ∗ link-line impedance.
- float kiz

  Z-directed constant for finding e.t.c ∗ link-line impedance.

3.13.1 Detailed Description

A struct describing the stub properties for all the SCNs in the mesh.

The documentation for this struct was generated from the following file:

- 3D-TLM/SCNNode.h

Index

calcMaxTimestep
- Solver, 24

calcMeshCutoff
- Solver, 24

calcStubSCNParams
- Solver, 24

configMeshExcitation
- Solver, 24

configMeshObservationType
- Solver, 25

connectMesh
- Solver, 25

convertSubPixelValuesToCharArray
- Solver, 26

createFDBinFiles
- Solver, 26

createLineBoundary
- Solver, 26

createMeshEdgeBoundaries
- Solver, 27

createSingleTDMeasurementPoint
- Solver, 27

createXYBoundaryFaceRect
- Solver, 28

createXZGaussianPlanewave
- Solver, 28

createXZMeasurementPointsPlane
- Solver, 29

createYZBoundaryFaceRect
- Solver, 29

exciteMeshWithFieldAtCoordinates
- Solver, 30

Field, 5

GaussianPulse, 6
- GaussianPulse, 7
- GaussianPulse, 7
- normalDistWithX, 7
- updatePulse, 7

getFreqStart
- MeasurementPoint, 9

getFreqStep
- MeasurementPoint, 9

getFreqStop
- MeasurementPoint, 10

implementBoundaryNode
- Solver, 30

importRaindropGeometryFromFile
- Solver, 31

importSimulationConfigParameters
- Solver, 31

initMPIChannels
- Solver, 32

initMeshNodesProperties
- Solver, 32

initRaindrops
- Solver, 32

itoa
- Solver, 32

MeasurementPoint, 8
- getFreqStart, 9
- getFreqStep, 9
- getFreqStop, 10
- MeasurementPoint, 9
- MeasurementPoint, 9
- outputNodeListToFile, 10
- recordNodeState, 10

MeshPoint, 10

meshSingleAbsorber
- Solver, 33

normalDistWithX
- GaussianPulse, 7

numberOfDigits
- Solver, 34

outputMeasurements
- Solver, 34

outputMeshToPPMFile
- Solver, 34

outputNodeListToFile
- MeasurementPoint, 10

outputSingleTDMeasurementPointLog
- Solver, 35

outputZSlicesForTimeStep
- Solver, 35

Pixel, 11

PointSource, 12
- PointSource, 12
- PointSource, 12
- updatePointSource, 14

ProtoDrop, 14
- ProtoDrop, 15
- ProtoDrop, 15

Raindrop, 15

recordNodeState
MeasurementPoint, 10
roundToNearestInt
Solver, 36

SCNNode, 18
ScatterVariables, 16
setMeshPoint
Solver, 36
SimulationConstants, 20
Solver, 21
calcMaxTimestep, 24
calcMeshCutoff, 24
calcStubSCNParams, 24
configMeshExcitation, 24
configMeshObservationType, 25
connectMesh, 25
convertSubPixelValuesToCharArray, 26
createFDBinFiles, 26
createLineBoundary, 26
createMeshEdgeBoundaries, 27
createSingleTDMeasurementPoint, 27
createXYBoundaryFaceRect, 28
createXZGaussianPlanewave, 28
createXZMeasurementPointsPlane, 29
createYZBoundaryFaceRect, 29
exciteMeshWithFieldAtCoordinates, 30
implementBoundaryNode, 30
importRaindropGeometryFromFile, 31
importSimulationConfigParameters, 31
initMPIChannels, 32
initMeshNodesProperties, 32
initRaindrops, 32
itoa, 32
meshSingleAbsorber, 33
numberOfDigits, 34
outputMeasurements, 34
outputMeshToPPMFile, 34
outputSingleTDMeasurementPointLog, 35
outputZSlicesForTimeStep, 35
roundToNearestInt, 36
setMeshPoint, 36
specMap, 37
swapInts, 37
updateMeasurementPoints, 37
updateMeasurements, 38
updateSingleTDMeasurementPoint, 38
specMap
Solver, 37
StubSCNProperties, 39
swapInts
Solver, 37

updateMeasurementPoints
Solver, 37
updateMeasurements
Solver, 38
updatePointSource
PointSource, 14
updatePulse
GaussianPulse, 7
updateSingleTDMeasurementPoint
Solver, 38

Appendix C

Mobile TLM solver documentation

Loughborough Wave Lab - iOS

v4.4.1

Generated by Doxygen 1.8.6

Thu Feb 27 2014 10:41:29

Contents

1 Hierarchical Index **1**

1.1 Class Hierarchy 1

2 Class Index **3**

2.1 Class List 3

3 Class Documentation **5**

3.1 BoundaryNode Class Reference 5

3.1.1 Method Documentation 6

3.1.1.1 initBoundaryAtXCoord:YCoordinate:withCondition:isEdge: 6

3.1.1.2 resetNodeBoundaryCondition 6

3.2 ChangeBoundariesViewController Class Reference 7

3.2.1 Method Documentation 8

3.2.1.1 bottomBoundaryConditionChanged: 8

3.2.1.2 drawPulsesOrBoundariesChanged: 8

3.2.1.3 leftBoundaryConditionChanged: 8

3.2.1.4 meshLossSliderValueChanged: 8

3.2.1.5 returnFromChangeBoundariesView: 8

3.2.1.6 rightBoundaryConditionChanged: 8

3.2.1.7 topBoundaryConditionChanged: 9

3.3 ConfigureOptionsViewController Class Reference 9

3.3.1 Method Documentation 10

3.3.1.1 animateBandwidthLabelsWithXOffset: 10

3.3.1.2 changeBoundariesButtonTapped: 10

3.3.1.3 excitationModeValueChanged: 10

3.3.1.4 loadCustomSimulationsButtonTapped: 11

3.3.1.5 returnFromConfigureOptionsView: 11

3.3.1.6 setButtonGradientFills 11

3.3.1.7 simulationModeValueChanged: 11

3.4 <ConfigViewDelegate> Protocol Reference 12

3.5 EditServerAddressViewController Class Reference 13

3.5.1 Method Documentation 14

3.5.1.1 cancelChangeServerAddressTapped: 14
3.5.1.2 goToCEMLStandardWebpage: 14
3.5.1.3 saveChangeServerAddressTapped: 14
3.5.1.4 updateContentsAndDismissEditView 14
3.6 <EditServerAddressViewControllerDelegate> Protocol Reference 15
3.7 InfoBox Class Reference 16
3.7.1 Method Documentation 16
3.7.1.1 addToView:withInfoText:WhilstAnimating: 16
3.8 MBProgressHUD Class Reference 17
3.8.1 Detailed Description 18
3.8.2 Method Documentation 19
3.8.2.1 allHUDsForView: 19
3.8.2.2 hide: 19
3.8.2.3 hide:afterDelay: 20
3.8.2.4 hideAllHUDsForView:animated: 20
3.8.2.5 hideHUDForView:animated: 21
3.8.2.6 HUDForView: 22
3.8.2.7 initWithView: 22
3.8.2.8 initWithWindow: 22
3.8.2.9 show: 23
3.8.2.10 showHUDAddedTo:animated: 23
3.8.2.11 showWhileExecuting:onTarget:withObject:animated: 24
3.8.3 Property Documentation 24
3.8.3.1 animationType 24
3.8.3.2 color 25
3.8.3.3 customView 25
3.8.3.4 delegate 25
3.8.3.5 detailsLabelFont 25
3.8.3.6 detailsLabelText 25
3.8.3.7 dimBackground 25
3.8.3.8 labelFont 25
3.8.3.9 labelText 25
3.8.3.10 margin 25
3.8.3.11 minShowTime 25
3.8.3.12 minSize 26
3.8.3.13 mode 26
3.8.3.14 opacity 26
3.8.3.15 progress 26
3.8.3.16 removeFromSuperViewOnHide 26
3.8.3.17 square 26

3.8.3.18 taskInProgress . . . 26
3.8.3.19 xOffset . . . 26
3.8.3.20 yOffset . . . 26
3.9 MBProgressHUD() Category Reference . . . 27
3.10 <MBProgressHUDDelegate> Protocol Reference . . . 27
3.10.1 Method Documentation . . . 28
3.10.1.1 hudWasHidden: . . . 28
3.11 MBRoundProgressView Class Reference . . . 28
3.11.1 Detailed Description . . . 29
3.11.2 Property Documentation . . . 29
3.11.2.1 backgroundTintColor . . . 29
3.11.2.2 progress . . . 29
3.11.2.3 progressTintColor . . . 29
3.12 MGSplitCornersView Class Reference . . . 30
3.13 MGSplitDividerView Class Reference . . . 31
3.14 MGSplitViewController Class Reference . . . 32
3.15 MGSplitViewController(MGPrivateMethods) Category Reference . . . 33
3.16 <MGSplitViewControllerDelegate> Protocol Reference . . . 34
3.17 PulseObject Class Reference . . . 34
3.17.1 Method Documentation . . . 35
3.17.1.1 initWithXCoord:YCoordinate:iterationCount:excitationMode: . . . 35
3.17.1.2 initWithXCoord:YCoordinate:iterationCount:excitationMode:wavelength: . . . 35
3.17.1.3 normalDistWithX:mean:stdDev: . . . 35
3.17.1.4 repeatPulse . . . 36
3.18 <SimListDelegate> Protocol Reference . . . 37
3.19 Simulation Class Reference . . . 37
3.20 SimulationListViewController Class Reference . . . 38
3.20.1 Method Documentation . . . 39
3.20.1.1 cancelCustomSimTapped: . . . 39
3.20.1.2 createSimulationFromSelectionAtIndexPath: . . . 40
3.20.1.3 editServerAddressTapped: . . . 40
3.20.1.4 loadSimulationList . . . 40
3.20.1.5 parseCEMLFile . . . 40
3.21 SMXMLDocument Class Reference . . . 41
3.22 SMXMLElement Class Reference . . . 43
3.23 TLMiPhoneAppDelegate Class Reference . . . 44
3.23.1 Method Documentation . . . 46
3.23.1.1 checkMeshBuffersForMalloc . . . 46
3.23.1.2 clearBuffers . . . 46
3.23.1.3 clearTLMArray . . . 47

3.23.1.4 fetchCurrentGlobalIterationCounterValue 47
3.23.1.5 resetScreenAndMeshDefaults 47
3.23.1.6 savePreviousGlobalIterationCounterValue 48
3.24 TLMiPhoneViewController Class Reference 48
3.24.1 Method Documentation 50
3.24.1.1 clearScreenButtonPressed: 50
3.24.1.2 drawToScreen: 50
3.24.1.3 drawToScreenNEON: 51
3.24.1.4 manageTimers 52
3.24.1.5 pauseRelevantTimer 52
3.24.1.6 playPauseButtonPressed: 52
3.24.1.7 renderSimulationFrame 53
3.24.1.8 resetMeshContents 53
3.24.1.9 resetModeTitle 54
3.24.1.10 showConfigureOptionsView: 54
3.25 TLMSolver Class Reference 55
3.25.1 Method Documentation 57
3.25.1.1 clearBuffers 57
3.25.1.2 clearPixelArray 57
3.25.1.3 computeBoundaries 57
3.25.1.4 computePulses 57
3.25.1.5 configureCurrentExcitationMode 58
3.25.1.6 createDoubleSlitWave 58
3.25.1.7 createRippleTankWave 58
3.25.1.8 createStraightLineBoundaryWithCondition:andStartX:andStartY:andEndX:andEndY:isEdge: 58
3.25.1.9 createWaveguide 59
3.25.1.10 getPixelArray 59
3.25.1.11 implementBoundaryConditionAtNode: 59
3.25.1.12 init 60
3.25.1.13 initSpecMapArrayWithCustomContrast: 60
3.25.1.14 injectEnergyValue:atX:andY: 61
3.25.1.15 meshCopyBack 61
3.25.1.16 meshSave 61
3.25.1.17 normalDistWithX:mean:stdDev: 61
3.25.1.18 removeEdgeBoundaries 61
3.25.1.19 scatterAndConnect 61
3.25.1.20 scatterAndConnectNEON 62
3.25.1.21 scatterAndConnectNEONWithYStart:andYEnd: 62
3.25.1.22 scatterAndConnectWithYStart:andYEnd: 62

3.25.1.23 seedNewImpulseWithXCoordinate:YCoordinate:withPhaseOffset: 62
3.25.1.24 setupEdgeBoundaries 63
3.25.1.25 updateLocalScatterAndConnectVariables 63

Index **64**

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

MBProgressHUD() . . . 27
MGSplitViewController(MGPrivateMethods) . . . 33
<MGSplitViewControllerDelegate> . . . 34
NSObject
    BoundaryNode . . . 5
    InfoBox . . . 16
    PulseObject . . . 34
    SMXMLDocument . . . 41
    SMXMLElement . . . 43
    TLMiPhoneAppDelegate . . . 44
    TLMSolver . . . 55
<NSObject>
    <ConfigViewDelegate> . . . 12
        TLMiPhoneViewController . . . 48
    <SimListDelegate> . . . 37
        ConfigureOptionsViewController . . . 9
<NSXMLParserDelegate>
    SMXMLDocument . . . 41
    SMXMLElement . . . 43
Simulation . . . 37
<UIApplicationDelegate>
    TLMiPhoneAppDelegate . . . 44
<UIPopoverControllerDelegate>
    ConfigureOptionsViewController . . . 9
    MGSplitViewController . . . 32
    TLMiPhoneViewController . . . 48
UITableViewController
    SimulationListViewController . . . 38
UIView
    MBProgressHUD . . . 17
    MBRoundProgressView . . . 28
    MGSplitCornersView . . . 30
    MGSplitDividerView . . . 31
UIViewController
    ChangeBoundariesViewController . . . 7
    ConfigureOptionsViewController . . . 9
    EditServerAddressViewController . . . 13
    MGSplitViewController . . . 32

TLMiPhoneViewController 48
<UIViewControllerNSObject>
<EditServerAddressViewControllerDelegate> 15
SimulationListViewController 38
<UIViewNSObject>
<MBProgressHUDDelegate> 27

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

BoundaryNode . . . 5
ChangeBoundariesViewController . . . 7
ConfigureOptionsViewController . . . 9
<ConfigViewDelegate> . . . 12
EditServerAddressViewController . . . 13
<EditServerAddressViewControllerDelegate> . . . 15
InfoBox . . . 16
MBProgressHUD . . . 17
MBProgressHUD() . . . 27
<MBProgressHUDDelegate> . . . 27
MBRoundProgressView . . . 28
MGSplitCornersView . . . 30
MGSplitDividerView . . . 31
MGSplitViewController . . . 32
MGSplitViewController(MGPrivateMethods) . . . 33
<MGSplitViewControllerDelegate> . . . 34
PulseObject . . . 34
<SimListDelegate> . . . 37
Simulation . . . 37
SimulationListViewController . . . 38
SMXMLDocument . . . 41
SMXMLElement . . . 43
TLMiPhoneAppDelegate . . . 44
TLMiPhoneViewController . . . 48
TLMSolver . . . 55

Chapter 3

Class Documentation

3.1 BoundaryNode Class Reference

Inheritance diagram for BoundaryNode:

NSObject

BoundaryNode

Collaboration diagram for BoundaryNode:

Instance Methods

- (id) - initBoundaryAtXCoord:YCoordinate:withCondition:isEdge:
- (void) - resetNodeBoundaryCondition

Protected Attributes

- TLMiPhoneAppDelegate ∗ **appDelegate**
- NSInteger **meshEdgeType**

Properties

- NSInteger **boundaryXCoord**
- NSInteger **boundaryYCoord**
- float **boundaryCondition**

3.1.1 Method Documentation

3.1.1.1 - (id) initBoundaryAtXCoord: (NSInteger) *x* YCoordinate:(NSInteger) *y* withCondition:(BOOL) *cond* isEdge:(NSInteger) *meshEdge*

Initialise a BoundaryNode object.

Parameters

x	The x-axis mesh co-ordinate value.
y	The y-axis mesh co-ordinate value.
cond	The boundary condition (from BMODE enum in Defs.h).
meshEdge	The mesh edge type (from BTYPE enum in Defs.h).

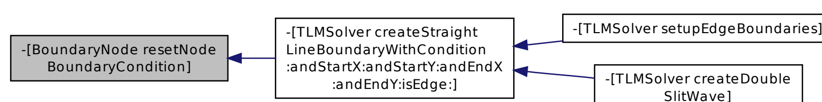
Returns

The initialised BoundaryNode object.

3.1.1.2 - (void) resetNodeBoundaryCondition

Resets the boundary condition based on the current mesh edge type.

Here is the caller graph for this function:



The documentation for this class was generated from the following files:

- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/BoundaryNode.h
- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/BoundaryNode.m

3.2 ChangeBoundariesViewController Class Reference

Inheritance diagram for ChangeBoundariesViewController:

Collaboration diagram for ChangeBoundariesViewController:

Instance Methods

- (IBAction) - topBoundaryConditionChanged:
- (IBAction) - bottomBoundaryConditionChanged:
- (IBAction) - leftBoundaryConditionChanged:
- (IBAction) - rightBoundaryConditionChanged:
- (IBAction) - drawPulsesOrBoundariesChanged:
- (IBAction) - meshLossSliderValueChanged:
- (IBAction) - returnFromChangeBoundariesView:

Protected Attributes

- TLMiPhoneAppDelegate ∗ **appDelegate**

Properties

- IBOutlet UISegmentedControl ∗ **topBoundarySegmentedControl**
- IBOutlet UISegmentedControl ∗ **bottomBoundarySegmentedControl**

- IBOutlet UISegmentedControl ∗ **leftBoundarySegmentedControl**
- IBOutlet UISegmentedControl ∗ **rightBoundarySegmentedControl**
- IBOutlet UISegmentedControl ∗ **drawPulsesOrBoundariesSegmentedControl**
- IBOutlet UISlider ∗ **meshLossSlider**
- IBOutlet UIButton ∗ **doneButton**

3.2.1 Method Documentation

3.2.1.1 - (IBAction) bottomBoundaryConditionChanged: (id) *sender*

IBAction to handle when the bottom boundary condition has changed.

Parameters

sender	IBOutlet representing the UISegmentedControl for the Bottom Boundary condition.

3.2.1.2 - (IBAction) drawPulsesOrBoundariesChanged: (id) *sender*

IBAction to handle when the Draw Pulses or Boundaries option has changed.

Parameters

sender	IBOutlet representing the UISegmentedControl for the Draw Pulses or Boundaries option.

3.2.1.3 - (IBAction) leftBoundaryConditionChanged: (id) *sender*

IBAction to handle when the left boundary condition has changed.

Parameters

sender	IBOutlet representing the UISegmentedControl for the Left Boundary condition.

3.2.1.4 - (IBAction) meshLossSliderValueChanged: (id) *sender*

IBAction to handle when the Mesh Loss slider value has changed.

Parameters

sender	IBOutlet reprensting the UISlider for Mesh Loss.

3.2.1.5 - (IBAction) returnFromChangeBoundariesView: (id) *sender*

Dismiss the Change Boundaries view.

Parameters

sender	IBOutlet representing the Done button.

3.2.1.6 - (IBAction) rightBoundaryConditionChanged: (id) *sender*

IBAction to handle when the right boundary condition has changed.

Parameters

sender	IBOutlet representing the UISegmentedControl for the Right Boundary condition.

3.2.1.7 - (IBAction) topBoundaryConditionChanged: (id) *sender*

IBAction to handle when the top boundary condition has changed.

Parameters

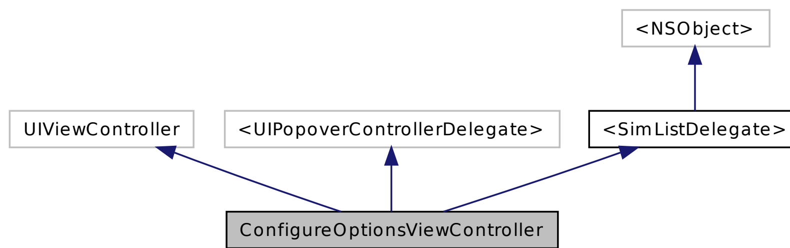
sender	IBOutlet representing the UISegmentedControl for the Top Boundary condition.

The documentation for this class was generated from the following files:

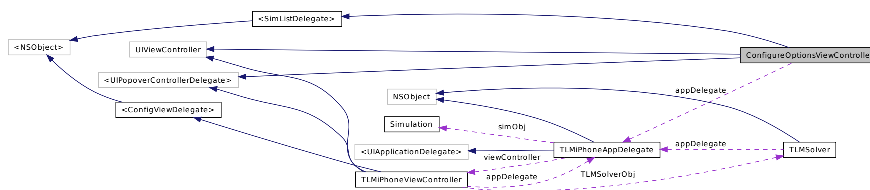
- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/ChangeBoundariesViewController.-h
- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/ChangeBoundariesViewController.-m

3.3 ConfigureOptionsViewController Class Reference

Inheritance diagram for ConfigureOptionsViewController:



Collaboration diagram for ConfigureOptionsViewController:



Instance Methods

- (IBAction) - returnFromConfigureOptionsView:
- (IBAction) - changeBoundariesButtonTapped:
- (IBAction) - loadCustomSimulationButtonTapped:
- (IBAction) - simulationModeValueChanged:

- (IBAction) - excitationModeValueChanged:
- (void) - setButtonGradientFills
- (void) - animateBandwidthLabelsWithXOffset:

Protected Attributes

- TLMiPhoneAppDelegate ∗ **appDelegate**

Properties

- id< ConfigViewDelegate > **delegate**
- IBOutlet UIButton ∗ **doneButton**
- IBOutlet UIButton ∗ **changeBoundariesButton**
- IBOutlet UIButton ∗ **loadCustomSimulationsButton**
- IBOutlet UISegmentedControl ∗ **modeSelectionSegmentControl**
- IBOutlet UISegmentedControl ∗ **excitationModeSegmentControl**
- IBOutlet UITextField ∗ **bandwidthTextField**
- IBOutlet UITextField ∗ **plainWaveWidthTextField**
- IBOutlet UILabel ∗ **plainWaveWidthLabel**
- IBOutlet UILabel ∗ **defaultLabel**
- IBOutlet UILabel ∗ **copyrightLabel**
- UIPopoverController ∗ **popOver**

3.3.1 Method Documentation

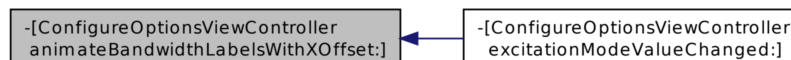
3.3.1.1 - (void) animateBandwidthLabelsWithXOffset: (NSInteger) *xOffset*

Animate the presentation/hiding of two configuration UITextFields.

Parameters

xOffset	The x-axis offset value for movement.

Here is the caller graph for this function:



3.3.1.2 - (IBAction) changeBoundariesButtonTapped: (id) *sender*

Present the Change Boundaries view.

Parameters

sender	IBOutlet representing the Change Boundaries button.

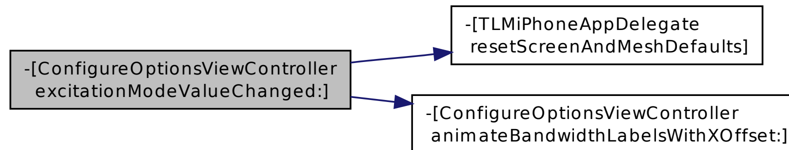
3.3.1.3 - (IBAction) excitationModeValueChanged: (id) *sender*

Update the current excitation mode.

Parameters

sender	IBOutlet representing the current excitation mode.

Here is the call graph for this function:



3.3.1.4 - (IBAction) loadCustomSimulationsButtonTapped: (id) *sender*

Present the Load Custom Simulations view.

Parameters

sender	IBOutlet representing the Load Custom Simulations button.

3.3.1.5 - (IBAction) returnFromConfigureOptionsView: (id) *sender*

Update simulation parameters when returning from the Configuration Options view to the main simulation view.

Parameters

sender	IBOutlet representing the Done button.

3.3.1.6 - (void) setButtonGradientFills

Configure the gradient fills for all the UIButton objects in the view.

3.3.1.7 - (IBAction) simulationModeValueChanged: (id) *sender*

Update the current simulation mode.

Parameters

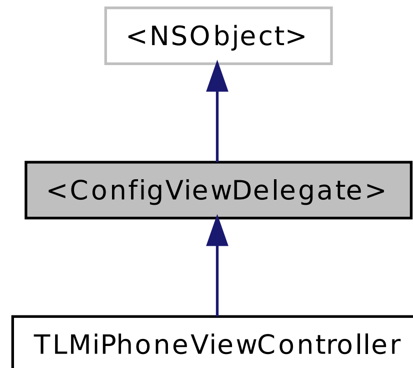
sender	IBOutlet representing the current simulation mode button.

The documentation for this class was generated from the following files:

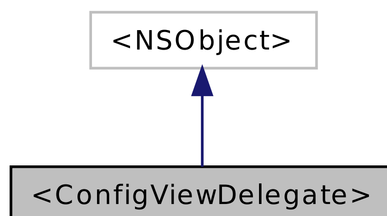
- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/ConfigureOptionsViewController.h
- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/ConfigureOptionsViewController.m

3.4 <ConfigViewDelegate> Protocol Reference

Inheritance diagram for <ConfigViewDelegate>:



Collaboration diagram for <ConfigViewDelegate>:



Instance Methods

- (void) - **swapMeshBuffersForModeTransition**
- (void) - **dismissPopOverView**
- (void) - **dismissConfigViewController**
- (void) - **updateLocalScatterAndConnectVariablesFromConfigView**

The documentation for this protocol was generated from the following file:

- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/ConfigureOptionsViewController.h

3.5 EditServerAddressViewController Class Reference

Inheritance diagram for EditServerAddressViewController:

UIViewController

EditServerAddressViewController

Collaboration diagram for EditServerAddressViewController:

<NSObject>
UIViewController
<UIPopoverControllerDelegate>
<ConfigViewDelegate>
NSObject
<UIApplicationDelegate>
Simulation
TLMiPhoneViewController
simObj
appDelegate
viewController
TLMiPhoneAppDelegate
TLMSolverObj
appDelegate
appDelegate
EditServerAddressViewController
TLMSolver

Instance Methods

- (IBAction) - cancelChangeServerAddressTapped:
- (IBAction) - saveChangeServerAddressTapped:
- (IBAction) - goToCEMLStandardWebpage:
- (void) - updateContentsAndDismissEditView

Protected Attributes

- TLMiPhoneAppDelegate $*$ **appDelegate**

Properties

- IBOutlet UIToolbar $*$ **CEMLStandardToolbar**
- IBOutlet UIBarButtonItem $*$ **CEMLStandardButton**
- IBOutlet UIBarButtonItem $*$ **cancelButton**
- IBOutlet UIBarButtonItem $*$ **saveButton**
- IBOutlet UITextField $*$ **serverAddressTextField**

- id
< EditServerAddressViewControllerDelegate > **delegate**

3.5.1 Method Documentation

3.5.1.1 - (IBAction) cancelChangeServerAddressTapped: (id) *sender*

IBAction to handle when the Cancel button has been pressed.

Parameters

sender	IBOutlet representing the Cancel button.

3.5.1.2 - (IBAction) goToCEMLStandardWebpage: (id) *sender*

IBAction for handling when the Goto CEML Standard button has been pressed.

Parameters

sender	IBOutlet representing the Goto CEML Standard button.

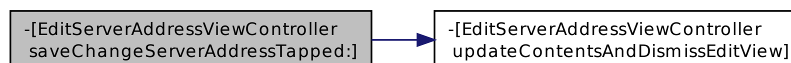
3.5.1.3 - (IBAction) saveChangeServerAddressTapped: (id) *sender*

IBAction for handling when the Save Server Address button has been pressed.

Parameters

sender	IBOutlet representing the Save button.

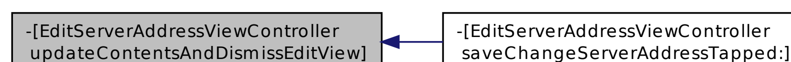
Here is the call graph for this function:



3.5.1.4 - (void) updateContentsAndDismissEditView

Update the server address URL and dismiss the Edit Server view.

Here is the caller graph for this function:



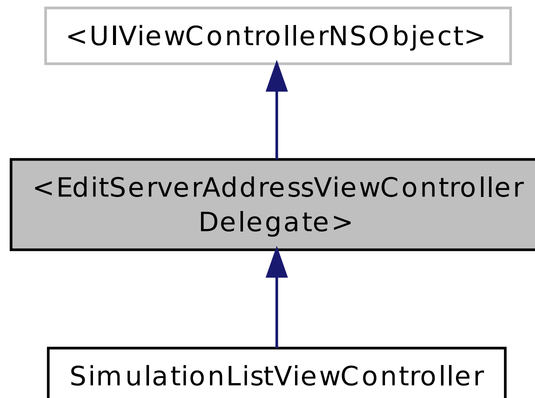
The documentation for this class was generated from the following files:

- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/EditServerAddressViewController.h

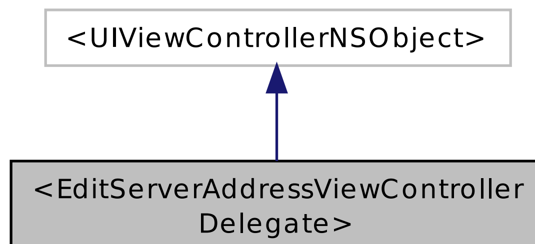
- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/EditServerAddressViewController.-m

3.6 <EditServerAddressViewControllerDelegate> Protocol Reference

Inheritance diagram for <EditServerAddressViewControllerDelegate>:



Collaboration diagram for <EditServerAddressViewControllerDelegate>:



Instance Methods

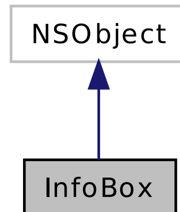
- (void) - **refreshXMLSimulationListTable**

The documentation for this protocol was generated from the following file:

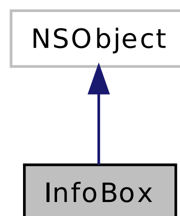
- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/EditServerAddressViewController.h

3.7 InfoBox Class Reference

Inheritance diagram for InfoBox:



Collaboration diagram for InfoBox:



Class Methods

- (id) + addToView:withInfoText:WhilstAnimating:

3.7.1 Method Documentation

3.7.1.1 + (id) addToView: (UIView∗) *view* withInfoText:(NSString∗) *infoText* WhilstAnimating:(BOOL) *animate*

Class method for adding an InfoBox object to a view.

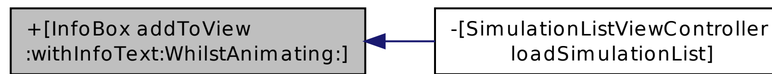
Parameters

view	The view to display the InfoBox within.
infoText	The information text to display within the InfoBox.
animate	A Boolean flag to determine if presentation of the InfoBox should be animated or not.

Returns

The initialised InfoBox object.

Here is the caller graph for this function:



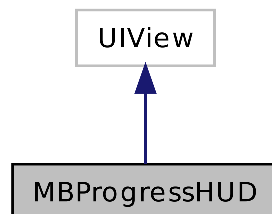
The documentation for this class was generated from the following files:

- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/InfoBox.h
- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/InfoBox.m

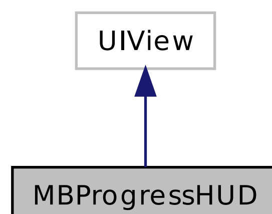
3.8 MBProgressHUD Class Reference

`#import <MBProgressHUD.h>`

Inheritance diagram for MBProgressHUD:



Collaboration diagram for MBProgressHUD:



Instance Methods

- (id) - initWithWindow:
- (id) - initWithView:
- (void) - show:
- (void) - hide:
- (void) - hide:afterDelay:
- (void) - showWhileExecuting:onTarget:withObject:animated:

Class Methods

- (MBProgressHUD *) + showHUDAddedTo:animated:
- (BOOL) + hideHUDForView:animated:
- (NSUInteger) + hideAllHUDsForView:animated:
- (MBProgressHUD *) + HUDForView:
- (NSArray *) + allHUDsForView:

Properties

- MBProgressHUDMode mode
- MBProgressHUDAnimation animationType
- UIView * customView
- id< MBProgressHUDDelegate > delegate
- NSString * labelText
- NSString * detailsLabelText
- float opacity
- UIColor * color
- float xOffset
- float yOffset
- float margin
- BOOL dimBackground
- float **graceTime**
- float minShowTime
- BOOL taskInProgress
- BOOL removeFromSuperViewOnHide
- UIFont * labelFont
- UIFont * detailsLabelFont
- float progress
- CGSize minSize
- BOOL square

3.8.1 Detailed Description

Displays a simple HUD window containing a progress indicator and two optional labels for short messages.

This is a simple drop-in class for displaying a progress HUD view similar to Apple's private UIProgressHUD class. The MBProgressHUD window spans over the entire space given to it by the initWithFrame constructor and catches all user input on this region, thereby preventing the user operations on components below the view. The HUD itself is drawn centered as a rounded semi-transparent view which resizes depending on the user specified content.

This view supports four modes of operation:

- MBProgressHUDModeIndeterminate - shows a UIActivityIndicatorView
- MBProgressHUDModeDeterminate - shows a custom round progress indicator

- MBProgressHUDModeAnnularDeterminate - shows a custom annular progress indicator
- MBProgressHUDModeCustomView - shows an arbitrary, user specified view (

  See Also

  customView)

  All three modes can have optional labels assigned:

- If the labelText property is set and non-empty then a label containing the provided content is placed below the indicator view.
- If also the detailsLabelText property is set then another label is placed below the first label.

3.8.2 Method Documentation

3.8.2.1 + (NSArray *) allHUDsForView: (UIView *) *view*

Finds all HUD subviews and returns them.

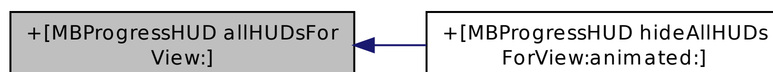
Parameters

view	The view that is going to be searched.

Returns

All found HUD views (array of MBProgressHUD objects).

Here is the caller graph for this function:



3.8.2.2 - (void) hide: (BOOL) *animated*

Hide the HUD. This still calls the hudWasHidden: delegate. This is the counterpart of the show: method. Use it to hide the HUD when your task completes.

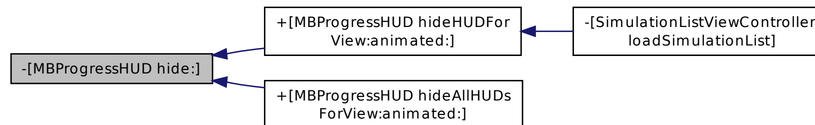
Parameters

animated	If set to YES the HUD will disappear using the current animationType. If set to NO the HUD will not use animations while disappearing.

See Also

animationType

Here is the caller graph for this function:



3.8.2.3 - (void) hide: (BOOL) *animated* afterDelay:(NSTimeInterval) *delay*

Hide the HUD after a delay. This still calls the hudWasHidden: delegate. This is the counterpart of the show: method. Use it to hide the HUD when your task completes.

Parameters

animated	If set to YES the HUD will disappear using the current animationType. If set to NO the HUD will not use animations while disappearing.
delay	Delay in seconds until the HUD is hidden.

See Also

animationType

3.8.2.4 + (NSUInteger) hideAllHUDsForView: (UIView ∗) *view* animated:(BOOL) *animated*

Finds all the HUD subviews and hides them.

Parameters

view	The view that is going to be searched for HUD subviews.
animated	If set to YES the HUDs will disappear using the current animationType. If set to NO the HUDs will not use animations while disappearing.

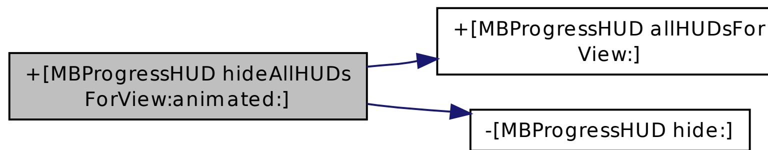
Returns

the number of HUDs found and removed.

See Also

hideAllHUDForView:animated:
animationType

Here is the call graph for this function:



3.8.2.5 + (BOOL) hideHUDForView: (UIView ∗) *view* animated:(BOOL) *animated*

Finds the top-most HUD subview and hides it. The counterpart to this method is showHUDAddedTo:animated:.

Parameters

view	The view that is going to be searched for a HUD subview.
animated	If set to YES the HUD will disappear using the current animationType. If set to NO the HUD will not use animations while disappearing.

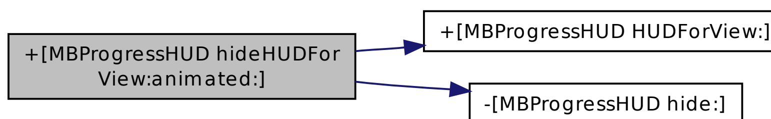
Returns

YES if a HUD was found and removed, NO otherwise.

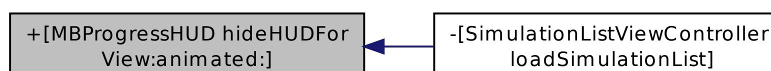
See Also

+ showHUDAddedTo:animated:
animationType

Here is the call graph for this function:



Here is the caller graph for this function:



3.8.2.6 + (MBProgressHUD ∗) HUDForView: (UIView ∗) *view*

Finds the top-most HUD subview and returns it.

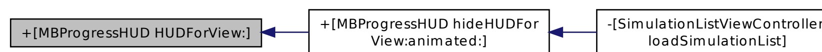
Parameters

view	The view that is going to be searched.

Returns

A reference to the last HUD subview discovered.

Here is the caller graph for this function:



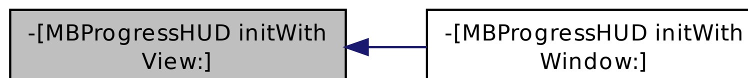
3.8.2.7 - (id) initWithView: (UIView ∗) *view*

A convenience constructor that initializes the HUD with the view's bounds. Calls the designated constructor with view.bounds as the parameter

Parameters

view	The view instance that will provide the bounds for the HUD. Should be the same instance as the HUD's superview (i.e., the view that the HUD will be added to).

Here is the caller graph for this function:



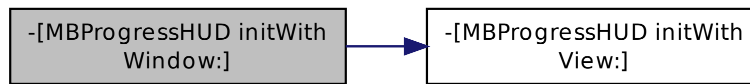
3.8.2.8 - (id) initWithWindow: (UIWindow ∗) *window*

A convenience constructor that initializes the HUD with the window's bounds. Calls the designated constructor with window.bounds as the parameter.

Parameters

window	The window instance that will provide the bounds for the HUD. Should be the same instance as the HUD's superview (i.e., the window that the HUD will be added to).

Here is the call graph for this function:



3.8.2.9 - (void) show: (BOOL) *animated*

Display the HUD. You need to make sure that the main thread completes its run loop soon after this method call so the user interface can be updated. Call this method when your task is already set-up to be executed in a new thread (e.g., when using something like NSOperation or calling an asynchronous call like NSURLRequest).

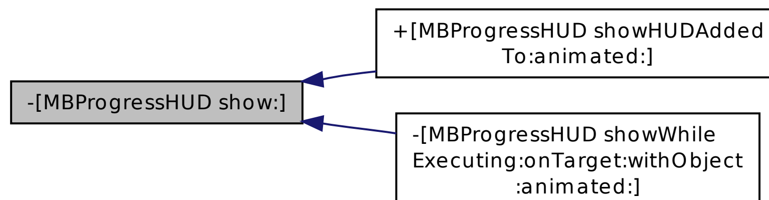
Parameters

animated	If set to YES the HUD will appear using the current animationType. If set to NO the HUD will not use animations while appearing.

See Also

animationType

Here is the caller graph for this function:



3.8.2.10 + (MBProgressHUD ∗) showHUDAddedTo: (UIView ∗) *view* animated:(BOOL) *animated*

Creates a new HUD, adds it to provided view and shows it. The counterpart to this method is hideHUDForView- :animated:.

Parameters

view	The view that the HUD will be added to
animated	If set to YES the HUD will appear using the current animationType. If set to NO the HUD will not use animations while appearing.

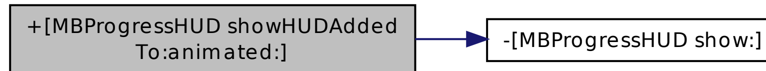
Returns

A reference to the created HUD.

See Also

+ hideHUDForView:animated:
animationType

Here is the call graph for this function:



3.8.2.11 - (void) showWhileExecuting: (SEL) *method* onTarget:(id) *target* withObject:(id) *object* animated:(BOOL) *animated*

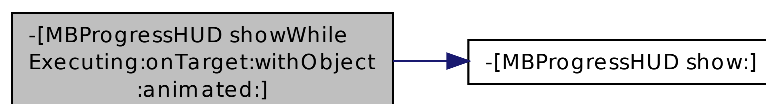
Shows the HUD while a background task is executing in a new thread, then hides the HUD.

This method also takes care of autorelease pools so your method does not have to be concerned with setting up a pool.

Parameters

method	The method to be executed while the HUD is shown. This method will be executed in a new thread.
target	The object that the target method belongs to.
object	An optional object to be passed to the method.
animated	If set to YES the HUD will (dis)appear using the current animationType. If set to NO the HUD will not use animations while (dis)appearing.

Here is the call graph for this function:



3.8.3 Property Documentation

3.8.3.1 - (MBProgressHUDAnimation) animationType [read], [write], [atomic], [assign]

The animation type that should be used when the HUD is shown and hidden.

See Also

MBProgressHUDAnimation

3.8.3.2 - (UIColor*) color [read], [write], [atomic], [assign]

The color of the HUD window. Defaults to black. If this property is set, color is set using this UIColor and the opacity property is not used. using retain because performing copy on UIColor base colors (like [UIColor greenColor]) cause problems with the copyZone.

3.8.3.3 - (UIView*) customView [read], [write], [atomic], [assign]

The UIView (e.g., a UIImageView) to be shown when the HUD is in MBProgressHUDModeCustomView. For best results use a 37 by 37 pixel view (so the bounds match the built in indicator bounds).

3.8.3.4 - (id<MBProgressHUDDelegate>) delegate [read], [write], [atomic], [assign]

The HUD delegate object.

See Also

MBProgressHUDDelegate

3.8.3.5 - (UIFont*) detailsLabelFont [read], [write], [atomic], [assign]

Font to be used for the details label. Set this property if the default is not adequate.

3.8.3.6 - (NSString*) detailsLabelText [read], [write], [atomic], [copy]

An optional details message displayed below the labelText message. This message is displayed only if the labelText property is also set and is different from an empty string (""). The details text can span multiple lines.

3.8.3.7 - (BOOL) dimBackground [read], [write], [atomic], [assign]

Cover the HUD background view with a radial gradient.

3.8.3.8 - (UIFont*) labelFont [read], [write], [atomic], [assign]

Font to be used for the main label. Set this property if the default is not adequate.

3.8.3.9 - (NSString*) labelText [read], [write], [atomic], [copy]

An optional short message to be displayed below the activity indicator. The HUD is automatically resized to fit the entire text. If the text is too long it will get clipped by displaying "..." at the end. If left unchanged or set to "", then no message is displayed.

3.8.3.10 - (float) margin [read], [write], [atomic], [assign]

The amount of space between the HUD edge and the HUD elements (labels, indicators or custom views). Defaults to 20.0

3.8.3.11 - (float) minShowTime [read], [write], [atomic], [assign]

The minimum time (in seconds) that the HUD is shown. This avoids the problem of the HUD being shown and than instantly hidden. Defaults to 0 (no minimum show time).

3.8.3.12 - (CGSize) minSize [read], [write], [atomic], [assign]

The minimum size of the HUD bezel. Defaults to CGSizeZero (no minimum size).

3.8.3.13 - (MBProgressHUDMode) mode [read], [write], [atomic], [assign]

MBProgressHUD operation mode. The default is MBProgressHUDModeIndeterminate.

See Also

MBProgressHUDMode

3.8.3.14 - (float) opacity [read], [write], [atomic], [assign]

The opacity of the HUD window. Defaults to 0.8 (80% opacity).

3.8.3.15 - (float) progress [read], [write], [atomic], [assign]

The progress of the progress indicator, from 0.0 to 1.0. Defaults to 0.0.

3.8.3.16 - (BOOL) removeFromSuperViewOnHide [read], [write], [atomic], [assign]

Removes the HUD from its parent view when hidden. Defaults to NO.

3.8.3.17 - (BOOL) square [read], [write], [atomic], [assign]

Force the HUD dimensions to be equal if possible.

3.8.3.18 - (BOOL) taskInProgress [read], [write], [atomic], [assign]

Indicates that the executed operation is in progress. Needed for correct graceTime operation. If you don't set a graceTime (different than 0.0) this does nothing. This property is automatically set when using showWhileExecuting:onTarget:withObject:animated:. When threading is done outside of the HUD (i.e., when the show: and hide: methods are used directly), you need to set this property when your task starts and completes in order to have normal graceTime functionality.

3.8.3.19 - (float) xOffset [read], [write], [atomic], [assign]

The x-axis offset of the HUD relative to the centre of the superview.

3.8.3.20 - (float) yOffset [read], [write], [atomic], [assign]

The y-ayis offset of the HUD relative to the centre of the superview.

The documentation for this class was generated from the following files:

- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/MBProgressHUD.h
- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/MBProgressHUD.m

3.9 MBProgressHUD() Category Reference

Instance Methods

- (void) - **setupLabels**
- (void) - **registerForKVO**
- (void) - **unregisterFromKVO**
- (NSArray ∗) - **observableKeypaths**
- (void) - **registerForNotifications**
- (void) - **unregisterFromNotifications**
- (void) - **updateUIForKeypath:**
- (void) - **hideUsingAnimation:**
- (void) - **showUsingAnimation:**
- (void) - **done**
- (void) - **updateIndicators**
- (void) - **handleGraceTimer:**
- (void) - **handleMinShowTimer:**
- (void) - **setTransformForCurrentOrientation:**
- (void) - **cleanUp**
- (void) - **launchExecution**
- (void) - **deviceOrientationDidChange:**
- (void) - **hideDelayed:**

Properties

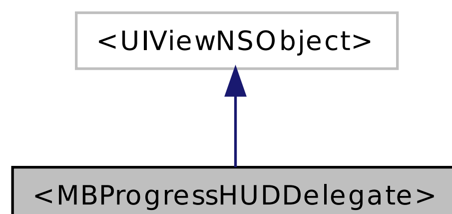
- UIView ∗ **indicator**
- NSTimer ∗ **graceTimer**
- NSTimer ∗ **minShowTimer**
- NSDate ∗ **showStarted**
- CGSize **size**

The documentation for this category was generated from the following file:

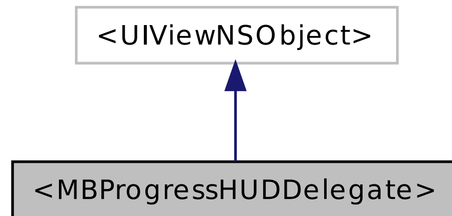
- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/MBProgressHUD.m

3.10 <MBProgressHUDDelegate> Protocol Reference

Inheritance diagram for <MBProgressHUDDelegate>:



Collaboration diagram for <MBProgressHUDDelegate>:



Instance Methods

- (void) - hudWasHidden:

3.10.1 Method Documentation

3.10.1.1 - (void) hudWasHidden: (MBProgressHUD *) *hud* [optional]

Called after the HUD was fully hidden from the screen.

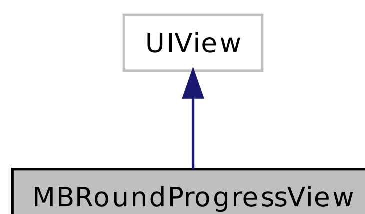
The documentation for this protocol was generated from the following file:

- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/MBProgressHUD.h

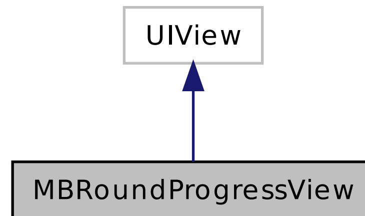
3.11 MBRoundProgressView Class Reference

```
#import <MBProgressHUD.h>
```

Inheritance diagram for MBRoundProgressView:



Collaboration diagram for MBRoundProgressView:



Properties

- float progress
- UIColor * progressTintColor
- UIColor * backgroundTintColor
- BOOL **annular**

3.11.1 Detailed Description

A progress view for showing definite progress by filling up a circle (pie chart).

3.11.2 Property Documentation

3.11.2.1 - (UIColor*) backgroundTintColor [read], [write], [nonatomic], [assign]

Indicator background (non-progress) color. Defaults to translucent white (alpha 0.1)

3.11.2.2 - (float) progress [read], [write], [nonatomic], [assign]

Progress (0.0 to 1.0)

3.11.2.3 - (UIColor*) progressTintColor [read], [write], [nonatomic], [assign]

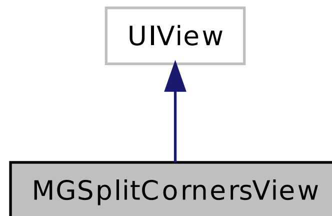
Indicator progress color. Defaults to white [UIColor whiteColor]

The documentation for this class was generated from the following files:

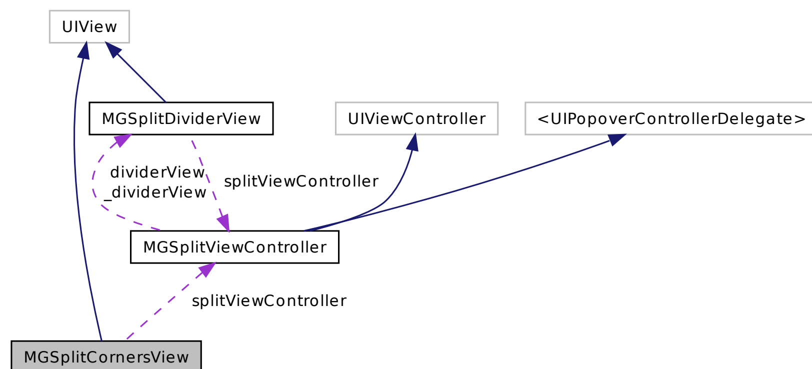
- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/MBProgressHUD.h
- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/MBProgressHUD.m

3.12 MGSplitCornersView Class Reference

Inheritance diagram for MGSplitCornersView:



Collaboration diagram for MGSplitCornersView:



Properties

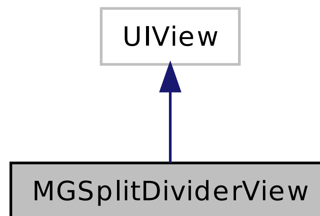
- float **cornerRadius**
- MGSplitViewController ∗ **splitViewController**
- MGCornersPosition **cornersPosition**
- UIColor ∗ **cornerBackgroundColor**

The documentation for this class was generated from the following file:

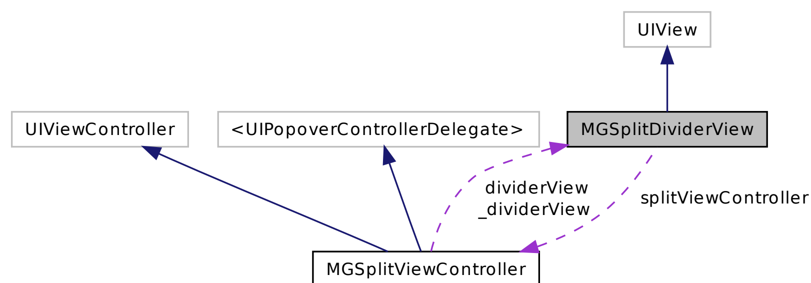
- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/MGSplitCornersView.h

3.13 MGSplitDividerView Class Reference

Inheritance diagram for MGSplitDividerView:



Collaboration diagram for MGSplitDividerView:



Instance Methods

- (void) - **drawGripThumbInRect:**

Properties

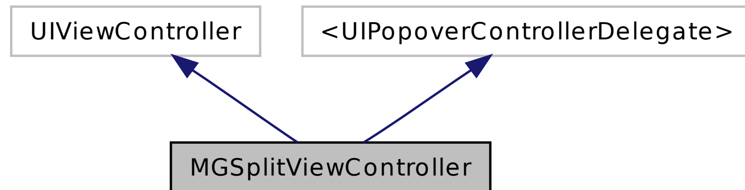
- MGSplitViewController ∗ **splitViewController**
- BOOL **allowsDragging**

The documentation for this class was generated from the following files:

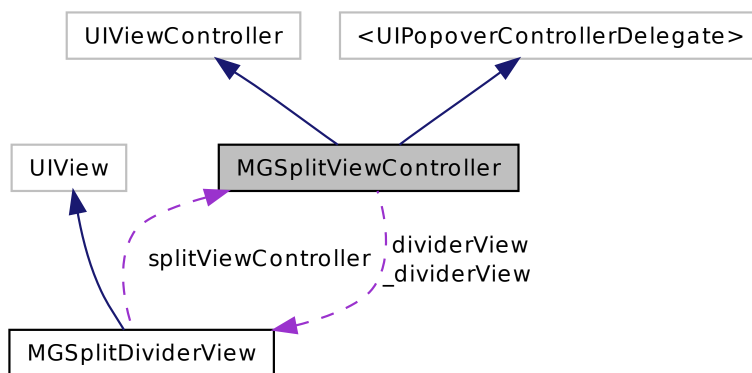
- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/MGSplitDividerView.h
- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/MGSplitDividerView.m

3.14 MGSplitViewController Class Reference

Inheritance diagram for MGSplitViewController:



Collaboration diagram for MGSplitViewController:



Instance Methods

- (IBAction) - **toggleSplitOrientation:**
- (IBAction) - **toggleMasterBeforeDetail:**
- (IBAction) - **toggleMasterView:**
- (IBAction) - **showMasterPopover:**
- (void) - **notePopoverDismissed**
- (BOOL) - **isShowingMaster**
- (void) - **setSplitPosition:animated:**
- (void) - **setDividerStyle:animated:**
- (NSArray ∗) - **cornerViews**

Protected Attributes

- BOOL **_showsMasterInPortrait**
- BOOL **_showsMasterInLandscape**

- float _**splitWidth**
- id _**delegate**
- BOOL _**vertical**
- BOOL _**masterBeforeDetail**
- NSMutableArray ∗ _**viewControllers**
- UIBarButtonItem ∗ _**barButtonItem**
- UIPopoverController ∗ _**hiddenPopoverController**
- MGSplitDividerView ∗ _**dividerView**
- NSArray ∗ _**cornerViews**
- float _**splitPosition**
- BOOL _**reconfigurePopup**
- MGSplitViewDividerStyle _**dividerStyle**

Properties

- IBOutlet id
  < MGSplitViewControllerDelegate > **delegate**
- BOOL **showsMasterInPortrait**
- BOOL **showsMasterInLandscape**
- BOOL **vertical**
- BOOL **masterBeforeDetail**
- float **splitPosition**
- float **splitWidth**
- BOOL **allowsDraggingDivider**
- NSArray ∗ **viewControllers**
- IBOutlet UIViewController ∗ **masterViewController**
- IBOutlet UIViewController ∗ **detailViewController**
- MGSplitDividerView ∗ **dividerView**
- MGSplitViewDividerStyle **dividerStyle**
- BOOL **landscape**

The documentation for this class was generated from the following files:

- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/MGSplitViewController.h
- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/MGSplitViewController.m

3.15 MGSplitViewController(MGPrivateMethods) Category Reference

Instance Methods

- (void) - **setup**
- (CGSize) - **splitViewSizeForOrientation:**
- (void) - **layoutSubviews**
- (void) - **layoutSubviewsWithAnimation:**
- (void) - **layoutSubviewsForInterfaceOrientation:withAnimation:**
- (BOOL) - **shouldShowMasterForInterfaceOrientation:**
- (BOOL) - **shouldShowMaster**
- (NSString ∗) - **nameOfInterfaceOrientation:**
- (void) - **reconfigureForMasterInPopover:**

The documentation for this category was generated from the following file:

- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/MGSplitViewController.m

3.16 <MGSplitViewControllerDelegate> Protocol Reference

Instance Methods

- (void) - **splitViewController:willHideViewController:withBarButtonItem:forPopoverController:**
- (void) - **splitViewController:willShowViewController:invalidatingBarButtonItem:**
- (void) - **splitViewController:popoverController:willPresentViewController:**
- (void) - **splitViewController:willChangeSplitOrientationToVertical:**
- (void) - **splitViewController:willMoveSplitToPosition:**
- (float) - **splitViewController:constrainSplitPosition:splitViewSize:**

The documentation for this protocol was generated from the following file:

- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/MGSplitViewController.h

3.17 PulseObject Class Reference

Inheritance diagram for PulseObject:

Collaboration diagram for PulseObject:

Instance Methods

- (id) - initWithXCoord:YCoordinate:iterationCount:excitationMode:
- (id) - initWithXCoord:YCoordinate:iterationCount:excitationMode:wavelength:
- (float) - normalDistWithX:mean:stdDev:
- (void) - repeatPulse

Protected Attributes

- TLMiPhoneAppDelegate ∗ **appDelegate**
- NSInteger **iterationCountAtSeed**
- int **pulseXCoord**
- int **pulseYCoord**
- float **zNormalDistributionValue**
- float **sinusoidValue**
- float **scalingValue**
- NSInteger **pulseExcitationMode**
- NSInteger **pulseBandwidth**
- BOOL **pulseCompleted**

3.17.1 Method Documentation

3.17.1.1 - (id) initWithXCoord: (int) *pulseXPos* YCoordinate:(int) *pulseYPos* iterationCount:(NSInteger) *count* excitationMode:(NSInteger) *excitation*

Initialises a PulseObject object.

Parameters

pulseXPos	The x-axis mesh co-ordinate for the pulse point source.
pulseYPos	The y-axis mesh co-ordinate for the pulse point source.
count	The number of time-steps that the pulse point source should persist for.
excitation	The excitation mode for the pulse point source (from the MODE enum in Defs.h).

Returns

The initialised PulseObject object.

3.17.1.2 - (id) initWithXCoord: (int) *pulseXPos* YCoordinate:(int) *pulseYPos* iterationCount:(NSInteger) *count* excitationMode:(NSInteger) *excitation* wavelength:(NSInteger) *wavelength*

Initialises a PulseObject object.

Parameters

pulseXPos	The x-axis mesh co-ordinate for the pulse point source.
pulseYPos	The y-axis mesh co-ordinate for the pulse point source.
count	The number of time-steps that the pulse point source should persist for.
excitation	The excitation mode for the pulse point source (from the MODE enum in Defs.h).
wavelength	The wavelength of the pulse point source.

Returns

The initialised PulseObject object.

3.17.1.3 - (float) normalDistWithX: (float) *x* mean:(float) *mean* stdDev:(float) *stdDev*

Calculates the Z-score for a given normal distribution.

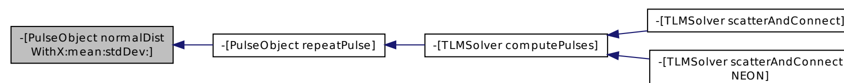
Parameters

x	The x value.
mean	The mean of the normal distribution.
stdDev	The standard deviation of the normal distribution.

Returns

The Z-Score value.

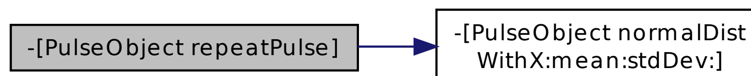
Here is the caller graph for this function:



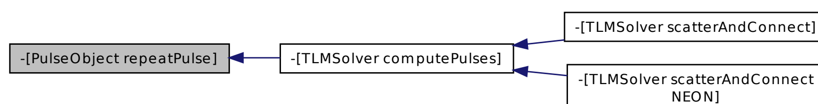
3.17.1.4 - (void) repeatPulse

Compute the next excitation value for the pulse point source, and inject it into the TLM mesh.

Here is the call graph for this function:



Here is the caller graph for this function:

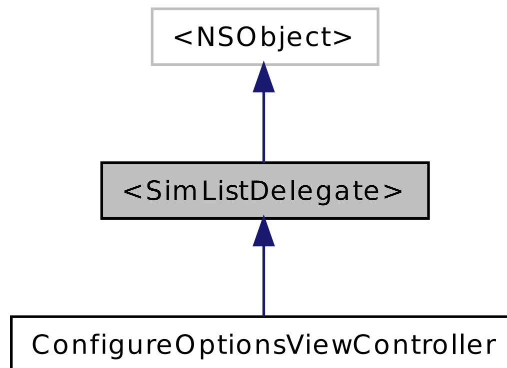


The documentation for this class was generated from the following files:

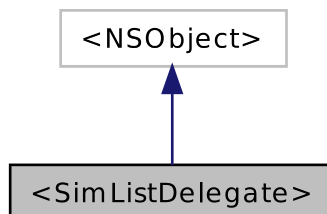
- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/PulseObject.h
- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/PulseObject.m

3.18 <SimListDelegate> Protocol Reference

Inheritance diagram for <SimListDelegate>:



Collaboration diagram for <SimListDelegate>:



Instance Methods

- (void) - **dismissModalSplitViewController**
- (void) - **dismissConfigViewControllerFromListViewController**

The documentation for this protocol was generated from the following file:

- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/SimulationListViewController.h

3.19 Simulation Class Reference

The documentation for this class was generated from the following file:

- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/Simulation.m

3.20 SimulationListViewController Class Reference

Inheritance diagram for SimulationListViewController:

Collaboration diagram for SimulationListViewController:

Instance Methods

- (void) - cancelCustomSimTapped:
- (void) - editServerAddressTapped:
- (void) - loadSimulationList
- (NSError ∗) - parseCEMLFile
- (BOOL) - createSimulationFromSelectionAtIndexPath:

Protected Attributes

- TLMiPhoneAppDelegate ∗ **appDelegate**
- BOOL **didTapACustomSimulation**
- NSMutableArray ∗ **simulations**

Properties

- id< SimListDelegate > **delegate**

3.20.1 Method Documentation

3.20.1.1 - (void) cancelCustomSimTapped: (id) *sender*

Dismiss the XML Simulation List view controller

Parameters

sender	IBOutlet representing the Cancel button.

3.20.1.2 - (BOOL) createSimulationFromSelectionAtIndexPath: (NSIndexPath∗) *path*

Configure TLM mesh and boundary conditions based on a selected simulation from the table view.

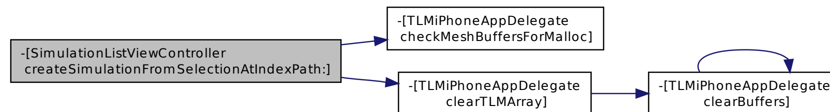
Parameters

path	The NSIndexPath of the selected table row.

Returns

A Boolean value to determine successful execution.

Here is the call graph for this function:



3.20.1.3 - (void) editServerAddressTapped: (id) *sender*

Present the Edit Server Address view controller.

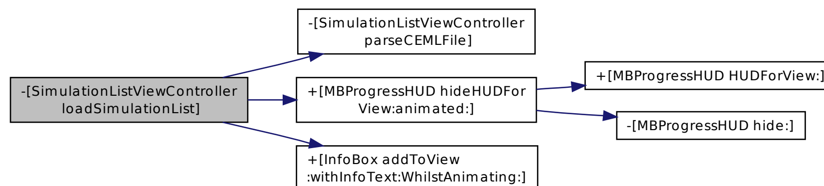
Parameters

sender	IBOutlet representing the Edit Server Address button.

3.20.1.4 - (void) loadSimulationList

Load the full simulation list and present it in the table view.

Here is the call graph for this function:



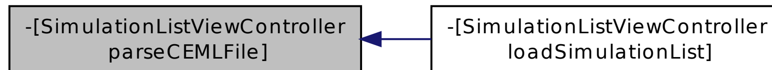
3.20.1.5 - (NSError ∗) parseCEMLFile

Parse the CEML XML file at the current server URL.

Returns

An NSError object (nil if no error).

Here is the caller graph for this function:

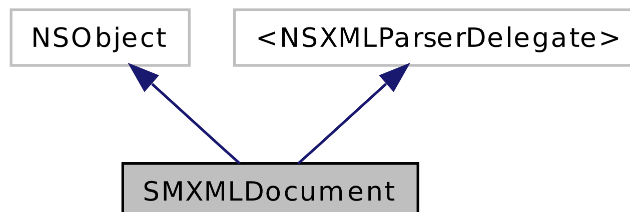


The documentation for this class was generated from the following files:

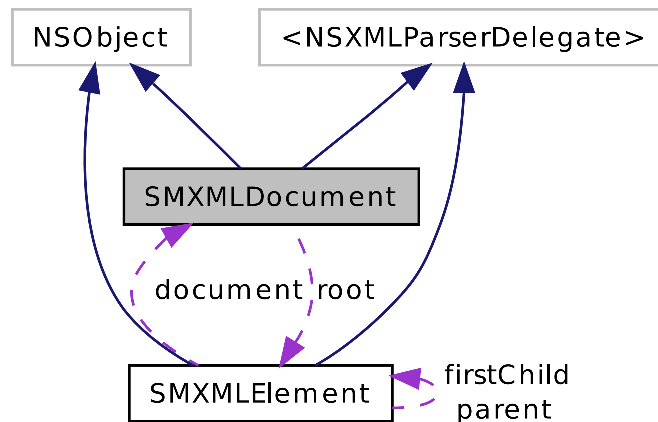
- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/SimulationListViewController.h
- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/SimulationListViewController.m

3.21 SMXMLDocument Class Reference

Inheritance diagram for SMXMLDocument:



Collaboration diagram for SMXMLDocument:



Instance Methods

- (id) - **initWithData:error:**

Class Methods

- (SMXMLDocument ∗) + **documentWithData:error:**

Properties

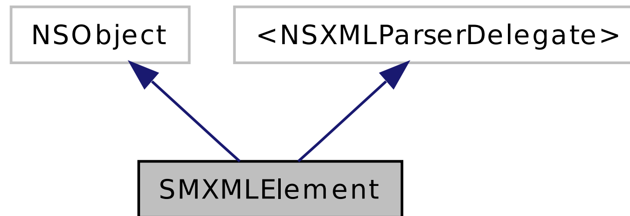
- SMXMLElement ∗ **root**
- NSError ∗ **error**

The documentation for this class was generated from the following files:

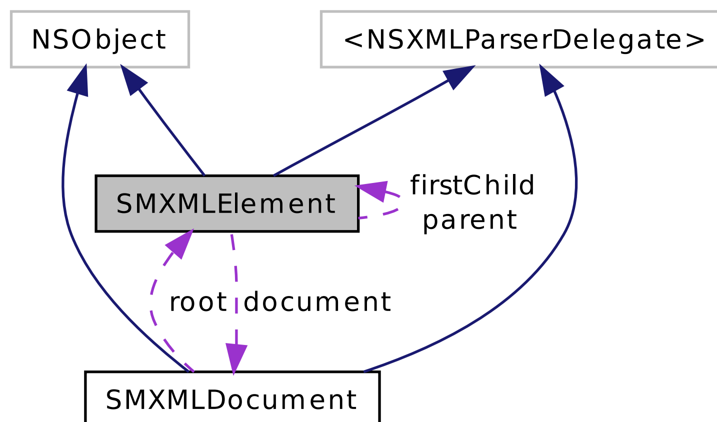
- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/SMXMLDocument.h
- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/SMXMLDocument.m

3.22 SMXMLElement Class Reference

Inheritance diagram for SMXMLElement:



Collaboration diagram for SMXMLElement:



Instance Methods

- (id) - **initWithDocument:**
- (SMXMLElement ∗) - **childNamed:**
- (NSArray ∗) - **childrenNamed:**
- (SMXMLElement ∗) - **childWithAttribute:value:**
- (NSString ∗) - **attributeNamed:**
- (SMXMLElement ∗) - **descendantWithPath:**
- (NSString ∗) - **valueWithPath:**

Properties

- SMXMLDocument ∗ **document**

- SMXMLElement ∗ **parent**
- NSString ∗ **name**
- NSString ∗ **value**
- NSArray ∗ **children**
- SMXMLElement ∗ **firstChild**
- NSDictionary ∗ **attributes**

The documentation for this class was generated from the following files:

- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/SMXMLDocument.h
- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/SMXMLDocument.m

3.23 TLMiPhoneAppDelegate Class Reference

Inheritance diagram for TLMiPhoneAppDelegate:

Collaboration diagram for TLMiPhoneAppDelegate:

Instance Methods

- (void) - savePreviousGlobalIterationCounterValue
- (void) - fetchCurrentGlobalIterationCounterValue
- (void) - resetScreenAndMeshDefaults
- (void) - clearTLMArray
- (void) - clearBuffers
- (void) - checkMeshBuffersForMalloc

Protected Attributes

- int **counterCopy** [6]

Properties

- IBOutlet UIWindow ∗ **window**
- IBOutlet TLMiPhoneViewController ∗ **viewController**
- float ∗ **GaussianMesh**
- float ∗ **SinusoidalMesh**
- float ∗ **RippleTankMesh**
- float ∗ **WaveGuideMesh**
- float ∗ **DoubleSlitMesh**
- float ∗ **CustomSimMesh**
- Simulation ∗ **simObj**
- NSInteger **globalIterationCounter**
- NSInteger **currentSimulationMode**
- NSInteger **currentExcitationMode**
- NSInteger **previousExcitationMode**
- NSInteger **gaussianPulseBandwidthInTimesteps**
- NSInteger **sinusoidalWaveBandwidthInTimesteps**
- NSInteger **plainWaveWidthInNodes**
- NSInteger **rippleTankWaveBandwidthInTimesteps**
- NSInteger **waveguideBandwidthInTimesteps**
- NSInteger **doubleSlitWaveBandwidthInTimesteps**
- NSInteger **xScreenConstraint**
- NSInteger **yScreenConstraint**
- NSInteger **xScreenWidth**
- NSInteger **yScreenWidth**
- NSInteger **rowLen**
- NSInteger **columnLen**
- NSInteger **nodesX**
- NSInteger **nodesY**
- NSNumber ∗ **meshLossFactor**
- NSNumber ∗ **leftBoundaryCondition**
- NSNumber ∗ **rightBoundaryCondition**
- NSNumber ∗ **topBoundaryCondition**
- NSNumber ∗ **bottomBoundaryCondition**
- NSMutableArray ∗ **gaussianPulseArray**
- NSMutableArray ∗ **sinusoidalPulseArray**
- NSMutableArray ∗ **ripplePulseArray**
- NSMutableArray ∗ **waveGuidePulseArray**
- NSMutableArray ∗ **doubleSlitPulseArray**
- NSMutableArray ∗ **customSimulationPulseArray**
- NSMutableArray ∗ **gaussianBoundaryArray**
- NSMutableArray ∗ **sinusoidalBoundaryArray**
- NSMutableArray ∗ **rippleBoundaryArray**
- NSMutableArray ∗ **waveGuideBoundaryArray**
- NSMutableArray ∗ **doubleSlitBoundaryArray**
- NSMutableArray ∗ **customSimulationBoundaryArray**
- NSMutableArray ∗ **customSimulationCircleBoundaryArray**
- NSMutableArray ∗ **topBoundaryArray**
- NSMutableArray ∗ **bottomBoundaryArray**
- NSMutableArray ∗ **leftBoundaryArray**

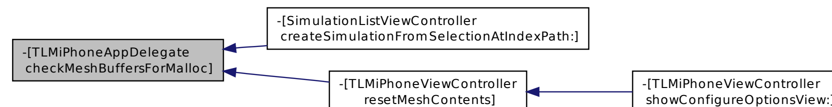
- NSMutableArray ∗ **rightBoundaryArray**
- BOOL **firstRipple**
- BOOL **firstWaveguide**
- BOOL **firstDoubleSlit**
- BOOL **firstGaussianTouch**
- BOOL **firstSinusoidTouch**
- BOOL **drawPulsesOrBoundaries**
- BOOL **firstShowOfChangeMeshInstructions**
- NSInteger **currentDevice**
- CFAbsoluteTime **startTime**

3.23.1 Method Documentation

3.23.1.1 - (void) checkMeshBuffersForMalloc

Lazily allocate the TLM mesh array for the current excitation mode (i.e. if the memory for the array has not already been allocated, then allocate it now).

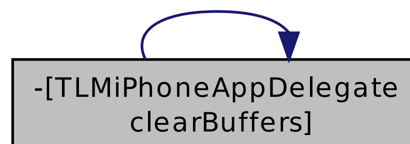
Here is the caller graph for this function:



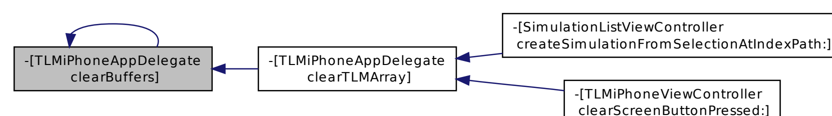
3.23.1.2 - (void) clearBuffers

Clear the two mesh buffers in the TLMSolver object of all impulse values in all ports.

Here is the call graph for this function:



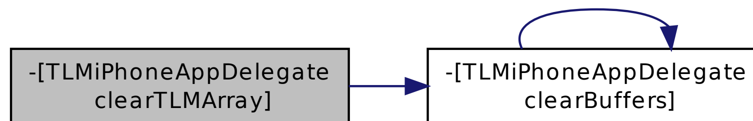
Here is the caller graph for this function:



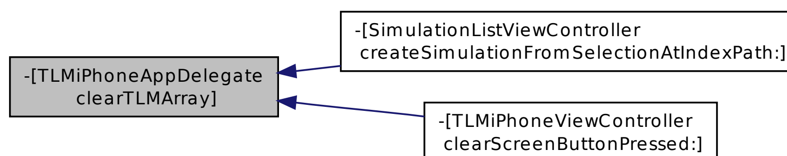
3.23.1.3 - (void) clearTLMArray

Clear the TLM mesh for the current excitation mode of all impulse values in all ports.

Here is the call graph for this function:



Here is the caller graph for this function:



3.23.1.4 - (void) fetchCurrentGlobalIterationCounterValue

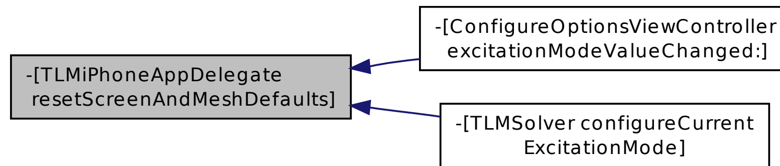
Fetches the previously saved value for the global iteration count for the excitation mode being transitioned into.

Here is the caller graph for this function:

3.23.1.5 - (void) resetScreenAndMeshDefaults

Resets the variables controlling the screen and mesh geometry to their default values.

Here is the caller graph for this function:



3.23.1.6 - (void) savePreviousGlobalIterationCounterValue

Saves a copy of the global iteration count value for the excitation mode being transitioned away from.

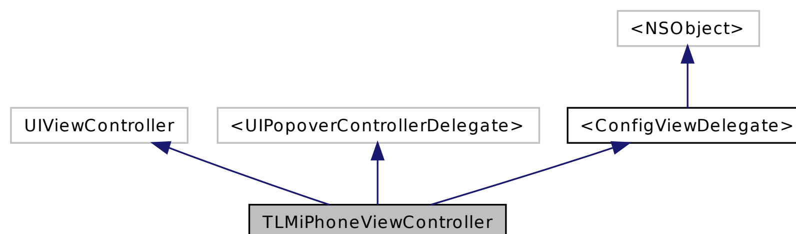
Here is the caller graph for this function:

The documentation for this class was generated from the following files:

- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/TLMiPhoneAppDelegate.h
- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/TLMiPhoneAppDelegate.m

3.24 TLMiPhoneViewController Class Reference

Inheritance diagram for TLMiPhoneViewController:



Collaboration diagram for TLMiPhoneViewController:

Instance Methods

- (void) - drawToScreen:
- (void) - drawToScreenNEON:
- (void) - renderSimulationFrame
- (void) - **endOfDrawLoopHousekeeping**
- (void) - resetModeTitle
- (void) - resetMeshContents
- (void) - manageTimers
- (void) - pauseRelevantTimer
- (IBAction) - showConfigureOptionsView:
- (IBAction) - playPauseButtonPressed:
- (IBAction) - clearScreenButtonPressed:

Protected Attributes

- TLMiPhoneAppDelegate ∗ **appDelegate**
- CGPoint **TLMSeedPoint**
- BOOL **swipeInProgressFlag**
- BOOL **syncSourcesModeFlag**
- BOOL **drawBoundariesFlag**
- NSTimer ∗ **executionGaussianTimer**
- NSTimer ∗ **executionSinusoidalTimer**
- NSTimer ∗ **executionRippleTankTimer**
- NSTimer ∗ **executionWaveGuideTimer**
- NSTimer ∗ **executionDoubleSlitTimer**
- NSTimer ∗ **executionCustomSimulationTimer**
- CFAbsoluteTime **startTime**
- CGFloat **initialPinchDistance**
- CGFloat **phaseChangeValue**
- UIImage ∗ **image**
- CGDataProviderRef **provider**
- CGImageRef **newImageRef**
- CGColorSpaceRef **colorspace**
- CGBitmapInfo **bitmapInfo**
- size_t **width**
- size_t **height**
- size_t **bitsPerComponent**
- size_t **bitsPerPixel**
- size_t **bytesPerRow**
- BOOL **previouslyShownInstructions**
- UILongPressGestureRecognizer ∗ **boundariesOrPulsesLongPressRecogniser**

Properties

- IBOutlet UIImageView ∗ **TLMImageView**
- IBOutlet UIImageView ∗ **tutorialImageView**
- TLMSolver ∗ **TLMSolverObj**
- IBOutlet UIButton ∗ **configureOptionsButton**
- IBOutlet UIButton ∗ **playPauseButton**
- IBOutlet UIButton ∗ **clearScreenButton**
- IBOutlet UILabel ∗ **excitationModeLabel**
- IBOutlet UILabel ∗ **multipleSourcesModeLabel**
- IBOutlet UILabel ∗ **longPressModeLabel**
- IBOutlet UILabel ∗ **fpsLabel**
- UIPopoverController ∗ **popOver**

3.24.1 Method Documentation

3.24.1.1 - (IBAction) clearScreenButtonPressed: (id) *sender*

IBAction triggered when the Clear Screen button is pressed.

Parameters

sender	IBOutlet represented by the Clear Screen button.

Here is the call graph for this function:

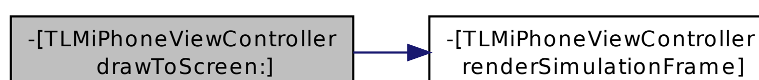
3.24.1.2 - (void) drawToScreen: (NSTimer∗) *timer*

Render the current simulation time-step to the screen, then execute scatter & connect for subsequent time-step.

Parameters

timer	The current NSTimer controlling simulation execution.

Here is the call graph for this function:



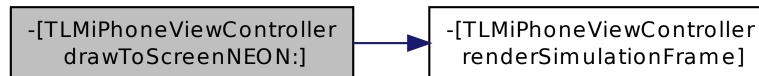
3.24.1.3 - (void) drawToScreenNEON: (NSTimer∗) *timer*

Render the current simulation time-step to the screen, then execute scatter & connect for subsequent time-step (using ARM NEON acceleration).

Parameters

timer	The current NSTimer controlling simulation execution.

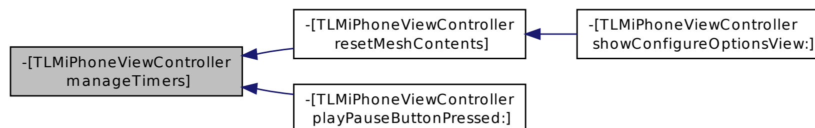
Here is the call graph for this function:



3.24.1.4 - (void) manageTimers

Manager the various NSTimer objects depending on the current excitation mode.

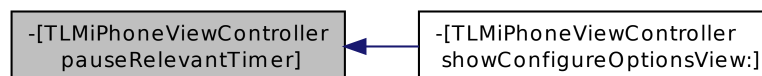
Here is the caller graph for this function:



3.24.1.5 - (void) pauseRelevantTimer

Pause the relevant NSTimer object depending on the current excitation mode.

Here is the caller graph for this function:



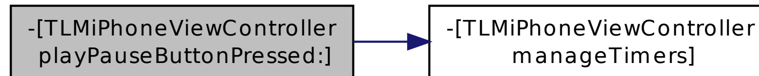
3.24.1.6 - (IBAction) playPauseButtonPressed: (id) *sender*

IBAction for when the Play/Pause button has been pressed by the user.

Parameters

sender	The IBOutlet representing the Play/Pause button.

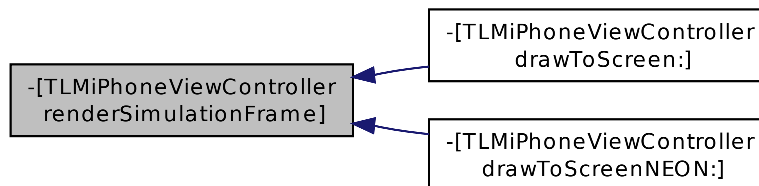
Here is the call graph for this function:



3.24.1.7 - (void) renderSimulationFrame

Render the current time-step to a UIImage and display on screen.

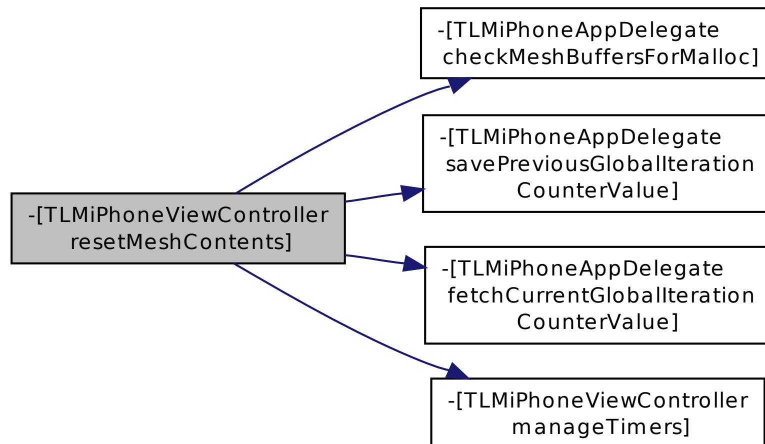
Here is the caller graph for this function:



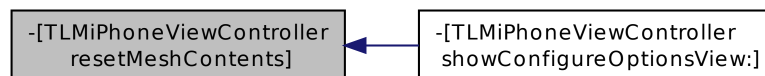
3.24.1.8 - (void) resetMeshContents

Reset the TLM mesh contents based on the current excitation mode.

Here is the call graph for this function:



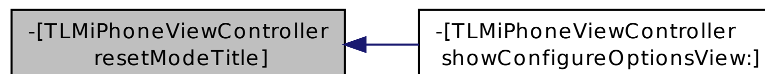
Here is the caller graph for this function:



3.24.1.9 - (void) resetModeTitle

Reset the mode title to reflect the current excitation mode.

Here is the caller graph for this function:



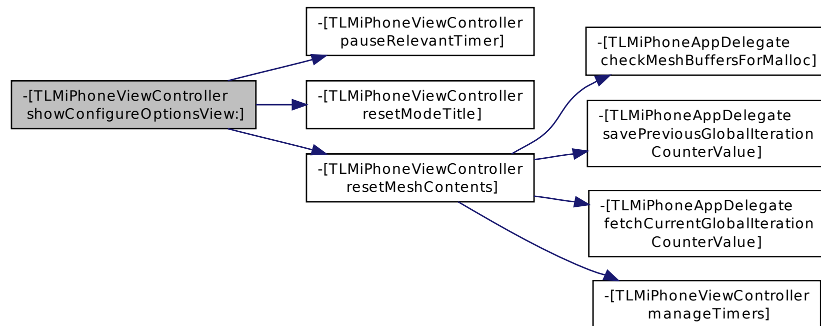
3.24.1.10 - (IBAction) showConfigureOptionsView: (id) *sender*

Display the Configuration Options view to the user.

Parameters

sender	The IBOutlet representing the Configuration Options button.

Here is the call graph for this function:

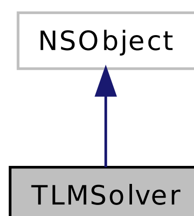


The documentation for this class was generated from the following files:

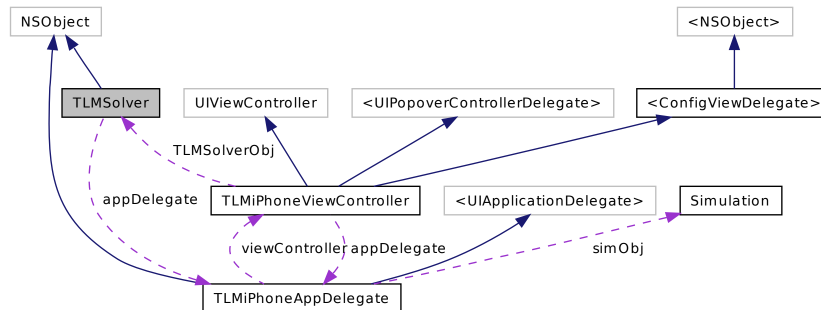
- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/TLMiPhoneViewController.h
- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/TLMiPhoneViewController.m

3.25 TLMSolver Class Reference

Inheritance diagram for TLMSolver:



Collaboration diagram for TLMSolver:



Instance Methods

- (id) - init
- (void) - seedNewImpulseWithXCoordinate:YCoordinate:withPhaseOffset:
- (void) - computeBoundaries
- (void) - computePulses
- (void) - implementBoundaryConditionAtNode:
- (void) - setupEdgeBoundaries
- (void) - removeEdgeBoundaries
- (void) - createStraightLineBoundaryWithCondition:andStartX:andStartY:andEndX:andEndY:isEdge:
- (void) - createRippleTankWave
- (void) - createWaveguide
- (void) - createDoubleSlitWave
- (void) - injectEnergyValue:atX:andY:
- (void) - updateLocalScatterAndConnectVariables
- (void) - scatterAndConnect
- (void) - scatterAndConnectWithYStart:andYEnd:
- (void) - scatterAndConnectNEON
- (void) - scatterAndConnectNEONWithYStart:andYEnd:
- (void) - configureCurrentExcitationMode
- (signed char ∗) - getPixelArray
- (void) - clearPixelArray
- (void) - initSpecMapArrayWithCustomContrast:
- (void) - meshCopyBack
- (void) - meshSave
- (void) - clearBuffers
- (float) - normalDistWithX:mean:stdDev:

Protected Attributes

- TLMiPhoneAppDelegate ∗ **appDelegate**

Properties

- NSInteger **iterationCountAtSeed**
- float **contrastValue**

3.25.1 Method Documentation

3.25.1.1 - (void) clearBuffers

Remove all the pulse voltage information in the main TLM mesh buffer arrays.

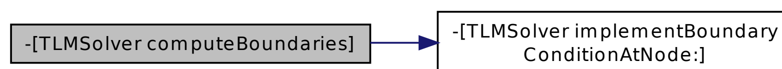
3.25.1.2 - (void) clearPixelArray

Remove all RGB information from the pixels array.

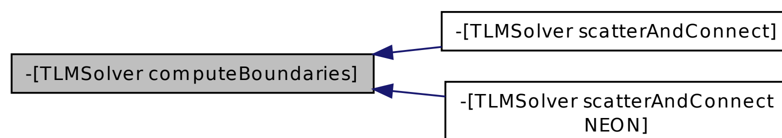
3.25.1.3 - (void) computeBoundaries

Execute boundary node calculations for the current time-step and current excitation mode.

Here is the call graph for this function:



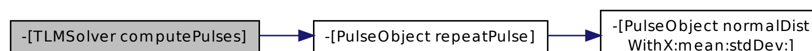
Here is the caller graph for this function:



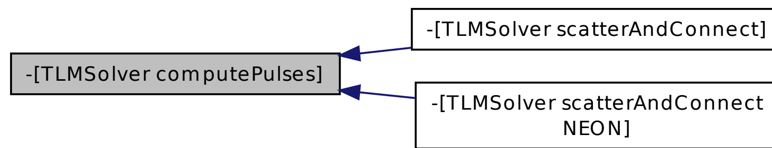
3.25.1.4 - (void) computePulses

Execute pulse point source calculations for the current time-step and current excitation mode.

Here is the call graph for this function:



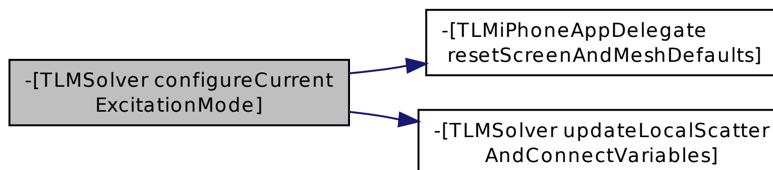
Here is the caller graph for this function:



3.25.1.5 - (void) configureCurrentExcitationMode

Ready the mesh for simulation when the current excitation mode of the TLMSolver object has changed.

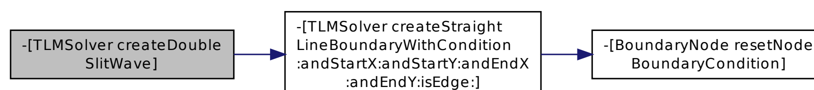
Here is the call graph for this function:



3.25.1.6 - (void) createDoubleSlitWave

Configure the Double Slit simulation.

Here is the call graph for this function:



3.25.1.7 - (void) createRippleTankWave

Configure the Ripple Tank simulation.

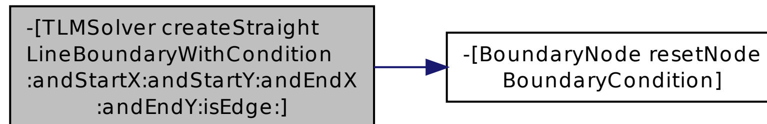
3.25.1.8 - (void) createStraightLineBoundaryWithCondition: (BOOL) *cond* andStartX:(NSInteger) *stX* andStartY:(NSInteger) *stY* andEndX:(NSInteger) *eX* andEndY:(NSInteger) *eY* isEdge:(NSInteger) *meshEdge*

Configure a straight-line of boundary nodes, with a given boundary condition.

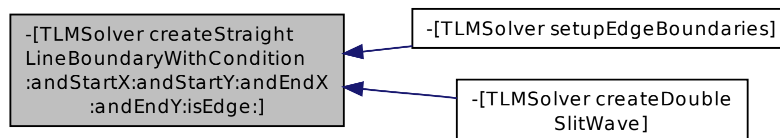
Parameters

cond	The boundary condition.
stX	The x-axis co-ordinate for the start of the straight-line boundary.
stY	The y-axis co-ordinate for the start of the straight-line boundary.
eX	The x-axis co-ordinate for the end of the straight-line boundary.
eY	The y-axis co-ordinate for the end of the straight-line boundary.
meshEdge	The mesh edge type.

Here is the call graph for this function:



Here is the caller graph for this function:



3.25.1.9 - (void) createWaveguide

Configure the Waveguide simulation.

3.25.1.10 - (signed char ∗) getPixelArray

Return the pixel char array ready for pushing to the device screen.

Returns

The 1D array of RGB values for screen pixels.

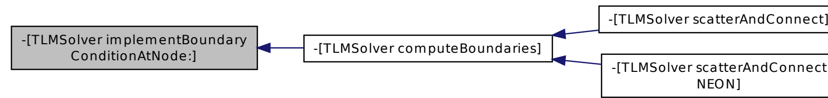
3.25.1.11 - (void) implementBoundaryConditionAtNode: (BoundaryNode∗) *bNode*

TLMSolver object instance method for reversing scatter & connect and implementing boundary condition for a given node.

Parameters

bNode	The BoundaryNode object to work with.

Here is the caller graph for this function:



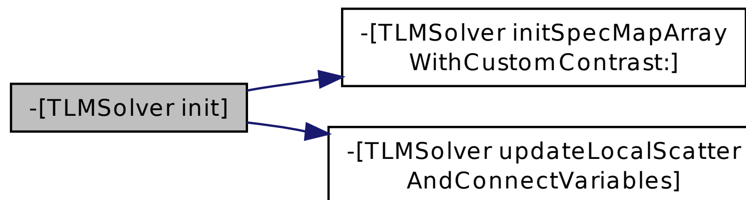
3.25.1.12 - (id) init

Initialise a TLMSolver object.

Returns

The initialised TLMSolver object.

Here is the call graph for this function:



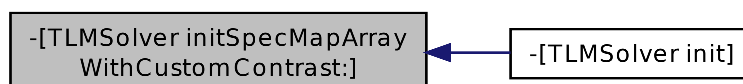
3.25.1.13 - (void) initSpecMapArrayWithCustomContrast: (BOOL) *customContrast*

Initialise the array holding the spectrum colourmap values for intensity to RGB pixel value conversions.

Parameters

customContrast	A Boolean flag to determine if the custom contrast value should be utilised.

Here is the caller graph for this function:



3.25.1.14 - (void) injectEnergyValue: (float) *energy* atX:(int) *x* andY:(int) *y*

Inject a specific pulse value into the TLM mesh at specific point.

Parameters

energy	The pulse value to inject.
x	The x-axis mesh co-ordinate.
y	The y-axis mesh co-ordinate.

3.25.1.15 - (void) meshCopyBack

Copy the previously-saved TLM mesh information back to the TLMSolver object's main TLM mesh array for the current excitation mode.

3.25.1.16 - (void) meshSave

Save a copy of the TLMSolver object's TLM mesh into the relevant storage array for the current excitation mode.

3.25.1.17 - (float) normalDistWithX: (float) *x* mean:(float) *mean* stdDev:(float) *stdDev*

Calculate a Z-score for a given normal distribution.

Parameters

x	The 'x' value.
mean	The mean of the normal distribution.
stdDev	The standard deviation of the normal distribution.

Returns

The computed Z-score value.

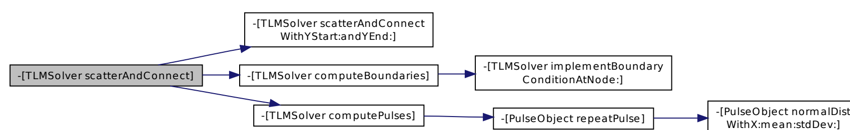
3.25.1.18 - (void) removeEdgeBoundaries

Remove all boundary nodes along TLM mesh edges.

3.25.1.19 - (void) scatterAndConnect

Execute the scatter & connect process on the TLM mesh for a single time-step (with no ARM NEON acceleration).

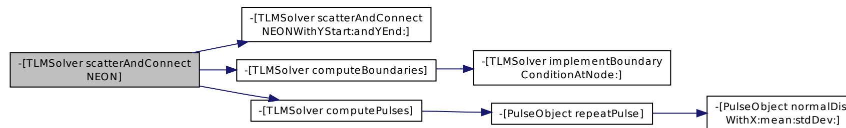
Here is the call graph for this function:



3.25.1.20 - (void) scatterAndConnectNEON

Execute the scatter & connect process on the TLM mesh for a single time-step (with ARM NEON acceleration).

Here is the call graph for this function:



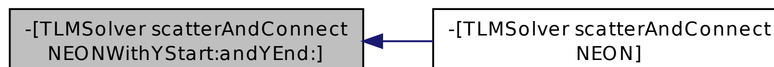
3.25.1.21 - (void) scatterAndConnectNEONWithYStart: (int) *yStart* andYEnd:(int) *yEnd*

Scatter & connect a single y-axis row of the TLM mesh (with ARM NEON acceleration).

Parameters

yStart	The y-axis start position.
yEnd	The y-axis end position.

Here is the caller graph for this function:



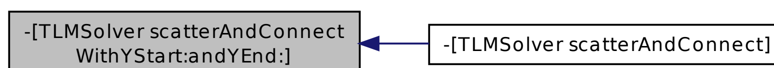
3.25.1.22 - (void) scatterAndConnectWithYStart: (int) *yStart* andYEnd:(int) *yEnd*

Scatter & connect a single y-axis row of the TLM mesh (with no ARM NEON acceleration).

Parameters

yStart	The y-axis start position.
yEnd	The y-axis end position.

Here is the caller graph for this function:



3.25.1.23 - (void) seedNewImpulseWithXCoordinate: (int) *touchXPos* YCoordinate:(int) *touchYPos* withPhaseOffset:(int) *phase*

Configure a new pulse point source (in response to a user-triggered touch event).

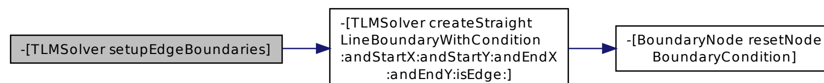
Parameters

touchXPos	The x-axis screen co-ordinate of the touch.
touchYPos	The y-axis screen co-ordinate of the touch.
phase	The phase offset value.

3.25.1.24 - (void) setupEdgeBoundaries

Configure the boundary conditions for each edge of the TLM mesh.

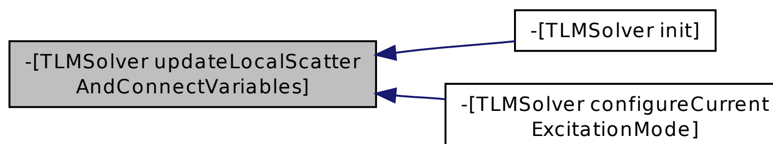
Here is the call graph for this function:



3.25.1.25 - (void) updateLocalScatterAndConnectVariables

Update the local copies of variables used during the scatter & connect process.

Here is the caller graph for this function:



The documentation for this class was generated from the following files:

- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/TLMSolver.h
- /Users/eeseadmin/GitHub Repos/Loughborough-Wave-Lab-iOS/Classes/TLMSolver.m

Index

<ConfigViewDelegate>, 12
<EditServerAddressViewControllerDelegate>, 15
<MBProgressHUDDelegate>, 27
<MGSplitViewControllerDelegate>, 34
<SimListDelegate>, 37

addToView:withInfoText:WhilstAnimating:
    InfoBox, 16
allHUDsForView:
    MBProgressHUD, 19
animateBandwidthLabelsWithXOffset:
    ConfigureOptionsViewController, 10
animationType
    MBProgressHUD, 24

backgroundTintColor
    MBRoundProgressView, 29
bottomBoundaryConditionChanged:
    ChangeBoundariesViewController, 8
BoundaryNode, 5
    initBoundaryAtXCoord:YCoordinate:withCondition-:isEdge:, 6
    resetNodeBoundaryCondition, 6

cancelChangeServerAddressTapped:
    EditServerAddressViewController, 14
cancelCustomSimTapped:
    SimulationListViewController, 39
changeBoundariesButtonTapped:
    ConfigureOptionsViewController, 10
ChangeBoundariesViewController, 7
    bottomBoundaryConditionChanged:, 8
    drawPulsesOrBoundariesChanged:, 8
    leftBoundaryConditionChanged:, 8
    meshLossSliderValueChanged:, 8
    returnFromChangeBoundariesView:, 8
    rightBoundaryConditionChanged:, 8
    topBoundaryConditionChanged:, 9
checkMeshBuffersForMalloc
    TLMiPhoneAppDelegate, 46
clearBuffers
    TLMiPhoneAppDelegate, 46
    TLMSolver, 57
clearPixelArray
    TLMSolver, 57
clearScreenButtonPressed:
    TLMiPhoneViewController, 50
clearTLMArray
    TLMiPhoneAppDelegate, 47
color
    MBProgressHUD, 24
computeBoundaries
    TLMSolver, 57
computePulses
    TLMSolver, 57
configureCurrentExcitationMode
    TLMSolver, 58
ConfigureOptionsViewController, 9
    animateBandwidthLabelsWithXOffset:, 10
    changeBoundariesButtonTapped:, 10
    excitationModeValueChanged:, 10
    loadCustomSimulationsButtonTapped:, 11
    returnFromConfigureOptionsView:, 11
    setButtonGradientFills, 11
    simulationModeValueChanged:, 11
createDoubleSlitWave
    TLMSolver, 58
createRippleTankWave
    TLMSolver, 58
createSimulationFromSelectionAtIndexPath:
    SimulationListViewController, 40
createWaveguide
    TLMSolver, 59
customView
    MBProgressHUD, 25

delegate
    MBProgressHUD, 25
detailsLabelFont
    MBProgressHUD, 25
detailsLabelText
    MBProgressHUD, 25
dimBackground
    MBProgressHUD, 25
drawPulsesOrBoundariesChanged:
    ChangeBoundariesViewController, 8
drawToScreen:
    TLMiPhoneViewController, 50
drawToScreenNEON:
    TLMiPhoneViewController, 50

editServerAddressTapped:
    SimulationListViewController, 40
EditServerAddressViewController, 13
    cancelChangeServerAddressTapped:, 14
    goToCEMLStandardWebpage:, 14
    saveChangeServerAddressTapped:, 14
    updateContentsAndDismissEditView, 14
excitationModeValueChanged:
    ConfigureOptionsViewController, 10

fetchCurrentGlobalIterationCounterValue
TLMiPhoneAppDelegate, 47

getPixelArray
TLMSolver, 59
goToCEMLStandardWebpage:
EditServerAddressViewController, 14

HUDForView:
MBProgressHUD, 22
hide:
MBProgressHUD, 19
hide:afterDelay:
MBProgressHUD, 20
hideAllHUDsForView:animated:
MBProgressHUD, 20
hideHUDForView:animated:
MBProgressHUD, 21
hudWasHidden:
MBProgressHUDDelegate-p, 28

implementBoundaryConditionAtNode:
TLMSolver, 59
InfoBox, 16
addToView:withInfoText:WhilstAnimating:, 16
init
TLMSolver, 60
initBoundaryAtXCoord:YCoordinate:withCondition:is-Edge:
BoundaryNode, 6
initSpecMapArrayWithCustomContrast:
TLMSolver, 60
initWithView:
MBProgressHUD, 22
initWithWindow:
MBProgressHUD, 22
initWithXCoord:YCoordinate:iterationCount:excitation-Mode:
PulseObject, 35
initWithXCoord:YCoordinate:iterationCount:excitation-Mode:wavelength:
PulseObject, 35
injectEnergyValue:atX:andY:
TLMSolver, 60

labelFont
MBProgressHUD, 25
labelText
MBProgressHUD, 25
leftBoundaryConditionChanged:
ChangeBoundariesViewController, 8
loadCustomSimulationsButtonTapped:
ConfigureOptionsViewController, 11
loadSimulationList
SimulationListViewController, 40

MBProgressHUD, 17
allHUDsForView:, 19
animationType, 24
color, 24
customView, 25
delegate, 25
detailsLabelFont, 25
detailsLabelText, 25
dimBackground, 25
HUDForView:, 22
hide:, 19
hide:afterDelay:, 20
hideAllHUDsForView:animated:, 20
hideHUDForView:animated:, 21
initWithView:, 22
initWithWindow:, 22
labelFont, 25
labelText, 25
margin, 25
minShowTime, 25
minSize, 25
mode, 26
opacity, 26
progress, 26
removeFromSuperViewOnHide, 26
show:, 23
showHUDAddedTo:animated:, 23
showWhileExecuting:onTarget:withObject:animated-:, 24
square, 26
taskInProgress, 26
xOffset, 26
yOffset, 26
MBProgressHUD(), 27
MBProgressHUDDelegate-p
hudWasHidden:, 28
MBRoundProgressView, 28
backgroundTintColor, 29
progress, 29
progressTintColor, 29
MGSplitCornersView, 30
MGSplitDividerView, 31
MGSplitViewController, 32
MGSplitViewController(MGPrivateMethods), 33
manageTimers
TLMiPhoneViewController, 52
margin
MBProgressHUD, 25
meshCopyBack
TLMSolver, 61
meshLossSliderValueChanged:
ChangeBoundariesViewController, 8
meshSave
TLMSolver, 61
minShowTime
MBProgressHUD, 25
minSize
MBProgressHUD, 25
mode
MBProgressHUD, 26

normalDistWithX:mean:stdDev:

PulseObject, 35
TLMSolver, 61

opacity
MBProgressHUD, 26

parseCEMLFile
SimulationListViewController, 40
pauseRelevantTimer
TLMiPhoneViewController, 52
playPauseButtonPressed:
TLMiPhoneViewController, 52
progress
MBProgressHUD, 26
MBRoundProgressView, 29
progressTintColor
MBRoundProgressView, 29
PulseObject, 34
initWithXCoord:YCoordinate:iterationCount-:excitationMode:, 35
initWithXCoord:YCoordinate:iterationCount-:excitationMode:wavelength:, 35
normalDistWithX:mean:stdDev:, 35
repeatPulse, 36

removeEdgeBoundaries
TLMSolver, 61
removeFromSuperViewOnHide
MBProgressHUD, 26
renderSimulationFrame
TLMiPhoneViewController, 53
repeatPulse
PulseObject, 36
resetMeshContents
TLMiPhoneViewController, 53
resetModeTitle
TLMiPhoneViewController, 54
resetNodeBoundaryCondition
BoundaryNode, 6
resetScreenAndMeshDefaults
TLMiPhoneAppDelegate, 47
returnFromChangeBoundariesView:
ChangeBoundariesViewController, 8
returnFromConfigureOptionsView:
ConfigureOptionsViewController, 11
rightBoundaryConditionChanged:
ChangeBoundariesViewController, 8

SMXMLDocument, 41
SMXMLElement, 43
saveChangeServerAddressTapped:
EditServerAddressViewController, 14
savePreviousGlobalIterationCounterValue
TLMiPhoneAppDelegate, 48
scatterAndConnect
TLMSolver, 61
scatterAndConnectNEON
TLMSolver, 61
scatterAndConnectNEONWithYStart:andYEnd:
TLMSolver, 62
scatterAndConnectWithYStart:andYEnd:
TLMSolver, 62
seedNewImpulseWithXCoordinate:YCoordinate:with-PhaseOffset:
TLMSolver, 62
setButtonGradientFills
ConfigureOptionsViewController, 11
setupEdgeBoundaries
TLMSolver, 63
show:
MBProgressHUD, 23
showConfigureOptionsView:
TLMiPhoneViewController, 54
showHUDAddedTo:animated:
MBProgressHUD, 23
showWhileExecuting:onTarget:withObject:animated:
MBProgressHUD, 24
Simulation, 37
SimulationListViewController, 38
cancelCustomSimTapped:, 39
createSimulationFromSelectionAtIndexPath:, 40
editServerAddressTapped:, 40
loadSimulationList, 40
parseCEMLFile, 40
simulationModeValueChanged:
ConfigureOptionsViewController, 11
square
MBProgressHUD, 26

TLMSolver, 55
clearBuffers, 57
clearPixelArray, 57
computeBoundaries, 57
computePulses, 57
configureCurrentExcitationMode, 58
createDoubleSlitWave, 58
createRippleTankWave, 58
createWaveguide, 59
getPixelArray, 59
implementBoundaryConditionAtNode:, 59
init, 60
initSpecMapArrayWithCustomContrast:, 60
injectEnergyValue:atX:andY:, 60
meshCopyBack, 61
meshSave, 61
normalDistWithX:mean:stdDev:, 61
removeEdgeBoundaries, 61
scatterAndConnect, 61
scatterAndConnectNEON, 61
scatterAndConnectNEONWithYStart:andYEnd:, 62
scatterAndConnectWithYStart:andYEnd:, 62
seedNewImpulseWithXCoordinate:YCoordinate-:withPhaseOffset:, 62
setupEdgeBoundaries, 63
updateLocalScatterAndConnectVariables, 63
TLMiPhoneAppDelegate, 44
checkMeshBuffersForMalloc, 46
clearBuffers, 46

clearTLMArray, 47
fetchCurrentGlobalIterationCounterValue, 47
resetScreenAndMeshDefaults, 47
savePreviousGlobalIterationCounterValue, 48
TLMiPhoneViewController, 48
clearScreenButtonPressed:, 50
drawToScreen:, 50
drawToScreenNEON:, 50
manageTimers, 52
pauseRelevantTimer, 52
playPauseButtonPressed:, 52
renderSimulationFrame, 53
resetMeshContents, 53
resetModeTitle, 54
showConfigureOptionsView:, 54
taskInProgress
MBProgressHUD, 26
topBoundaryConditionChanged:
ChangeBoundariesViewController, 9

updateContentsAndDismissEditView
EditServerAddressViewController, 14
updateLocalScatterAndConnectVariables
TLMSolver, 63

xOffset
MBProgressHUD, 26

yOffset
MBProgressHUD, 26

Appendix D

iOS mobile solver analytics data (28/01/2013 - 24/02/2014)

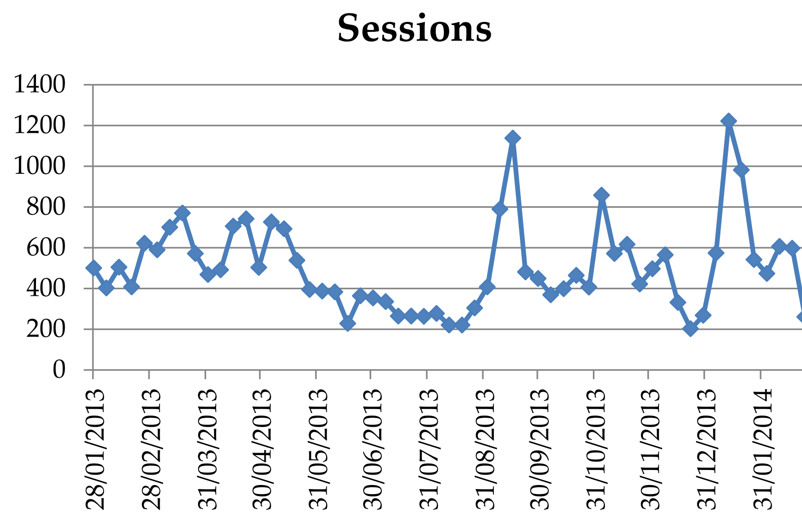


Figure D.1: Weekly user sessions figures for the iOS version of the mobile solver application.

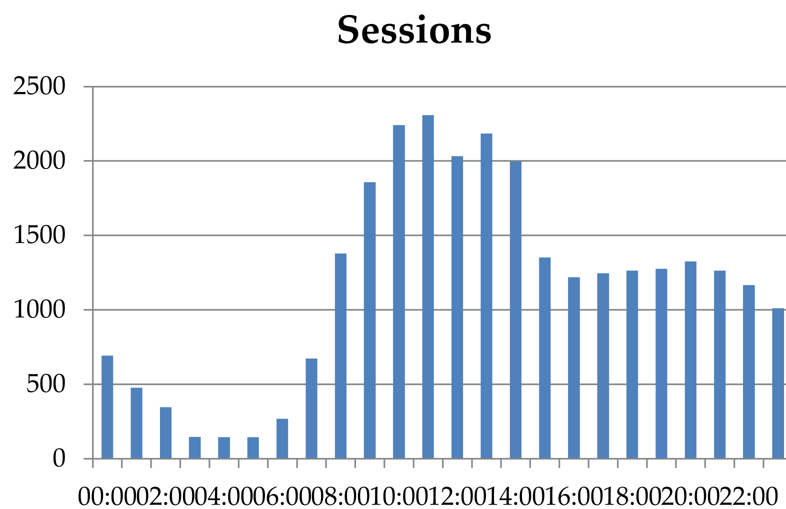


Figure D.2: User sessions observed over the course of any given 24 hour period for the iOS version of the solver application.

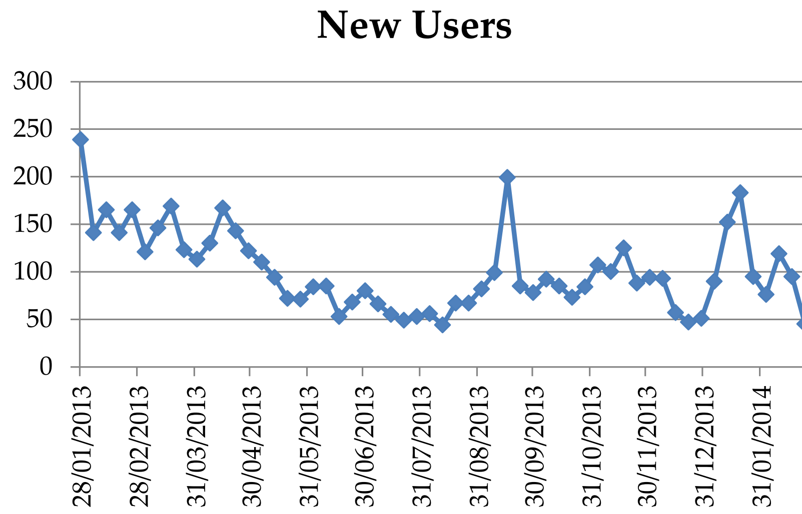


Figure D.3: Weekly new user sessions figures for the iOS version of the mobile solver application.

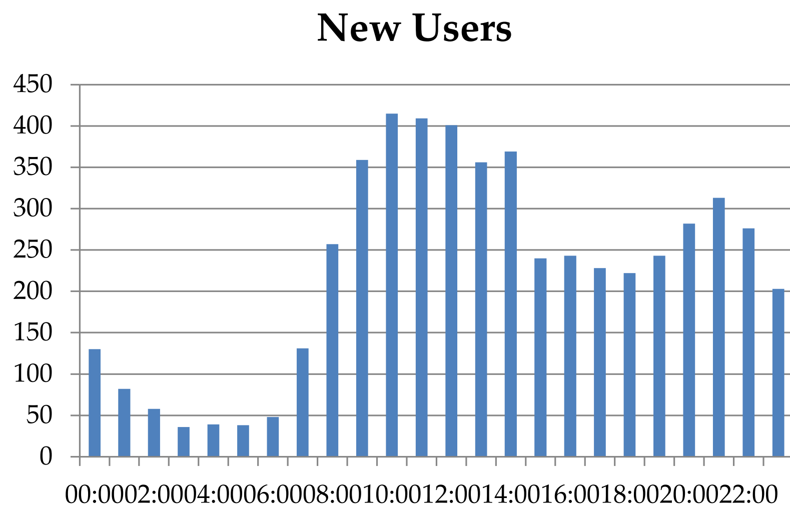


Figure D.4: New user sessions observed over the course of any given 24 hour period for the iOS version of the solver application.

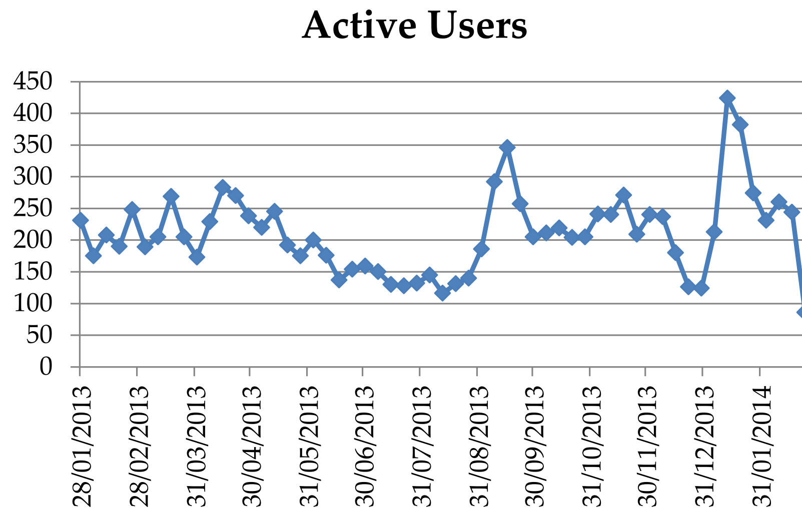


Figure D.5: Weekly active user sessions figures for the iOS version of the mobile solver application.

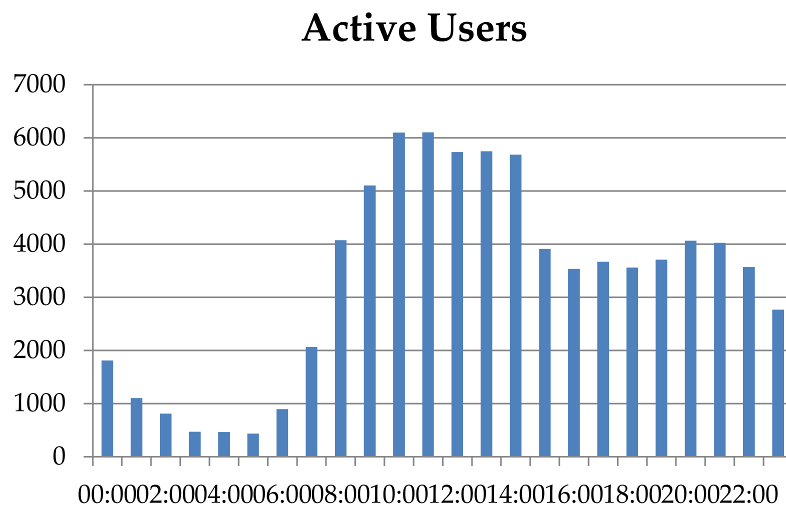


Figure D.6: Active user sessions observed over the course of any given 24 hour period for the iOS version of the solver application.

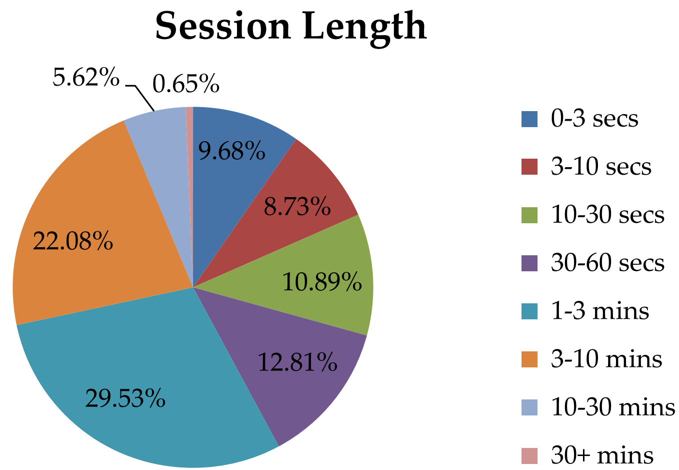


Figure D.7: Session length usage data across all users for the iOS version of the solver application.

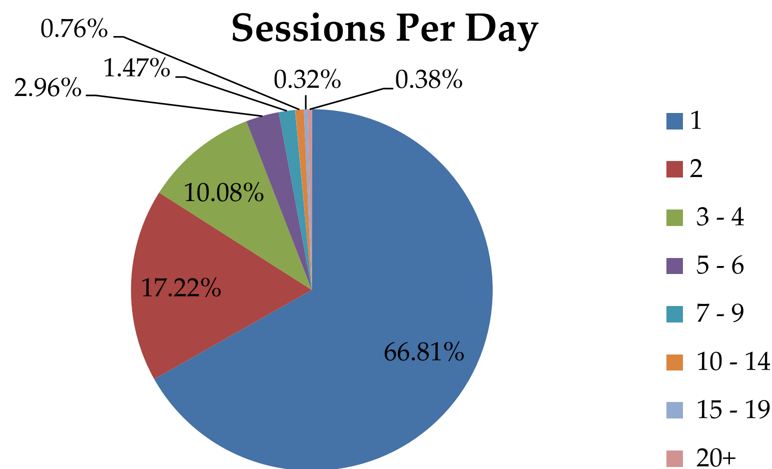


Figure D.8: Session per day usage data across all users for the iOS version of the solver application.

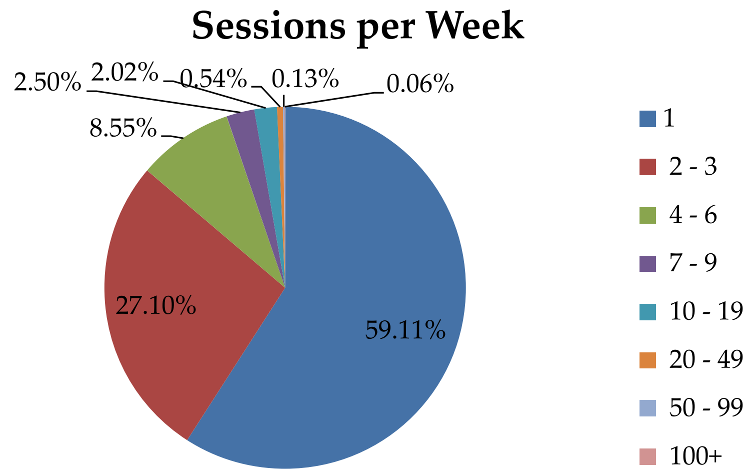


Figure D.9: Session per week usage data across all users for the iOS version of the solver application.

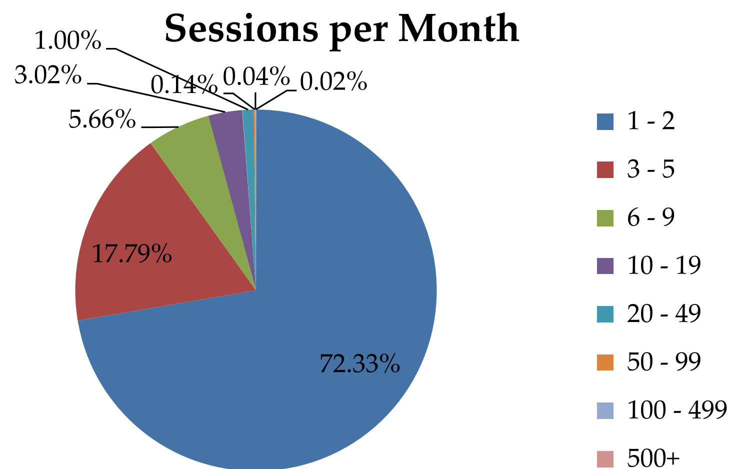


Figure D.10: Session per month usage data across all users for the iOS version of the solver application.

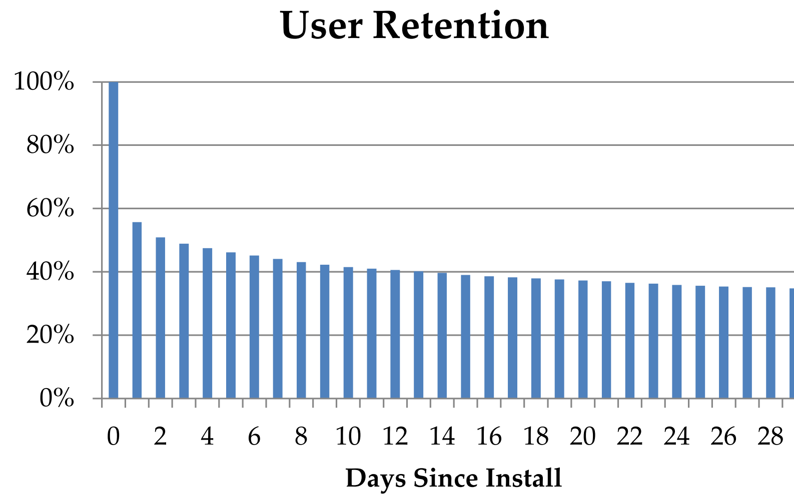


Figure D.11: Usage retention percentages data for the first 30 days after installing the application.

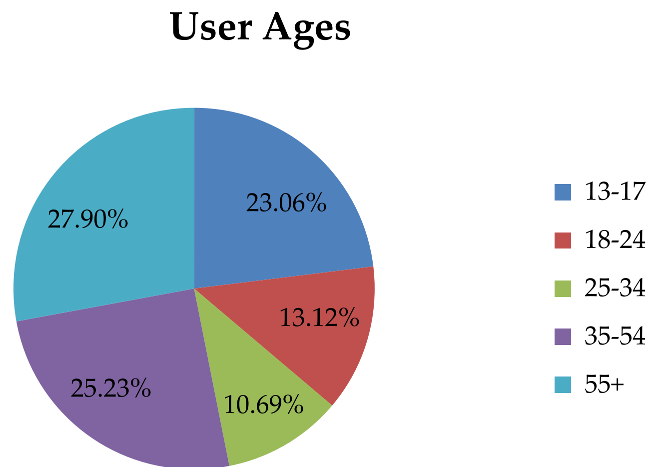


Figure D.12: User age distribution (based on the 20.8% of users where age is known) for the iOS version of the solver application.

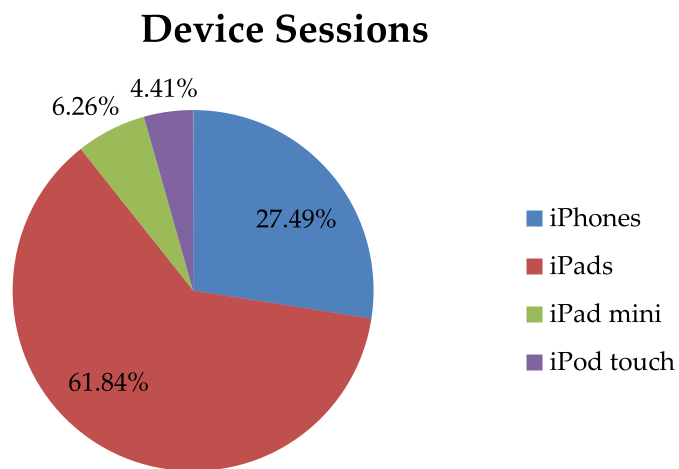


Figure D.13: iOS device family usage percentages across all sessions.