ABSTRACT

NEW COMPUTATIONAL APPROACHES TO THE N-BODY PROBLEM WITH APPLICATIONS TO ELECTRON COOLING OF HEAVY ION BEAMS

Pulukkuttige D. Sumana Abeyratne, Ph.D.
Department of Physics
Northern Illinois University, 2016
Professor Bela Erdelyi, Director

Thomas Jefferson National Accelerator Facility (JLab) has proposed a new Electron-Ion Collider, JLEIC. In this collider, a polarized electron beam and a counter rotating ion beam collide at the interaction point(s). A critical problem for the JLEIC collider is cooling the ion beam to ensure small emittance and to achieve high luminosity. Since electron cooling—a method of cooling ‘hot’ ion beams through Coulomb interactions with ‘cold’ electron beams—is one of the most effective cooling methods, it will be used by JLEIC. However, the most naive way of calculating Coulomb forces through the pair-wise method becomes infeasible even with the most high performing computers since the computational complexity grows $O(N^2)$, where $N$ is the number of particles as large as $10^{11}$.

In this dissertation, we have developed new computational tools and a high performance computer code that allows, for the first time, a particle-based simulation of realistic electron cooling scenarios of heavy ion beams. Our toolset, collectively referred to as the Particles High-Order Adaptive Dynamics (PHAD), contains three specific tools. The first tool, the adaptive multi-level Fast Multipole Method, reduces the computational cost of computing Coulomb forces to only $O(N)$. Our platform supports particles of any complex distribution.
(2D or 3D). The second tool, the Picard iteration-based integrator, resolves close encounters of particles efficiently and accurately. Finally, the third tool, the Strang operator splitting, reduces the runtime while maintaining the accuracy. The high performance code is comprised of these three main components.

Although, the proposed toolset is both precise and fast, completely simulating the electron cooling of the ion beam still takes a long time on a modern computer cluster due to the millions of small time steps that needs to be simulated. In order to overcome this challenge, we have developed an MPI-parallel high performance computer code to speed up our simulations. The results gathered to date are for over one million time steps, which is less than 1% of the cooling time.
NEW COMPUTATIONAL APPROACHES TO THE N-BODY PROBLEM WITH APPLICATIONS TO ELECTRON COOLING OF HEAVY ION BEAMS

BY
PULUKKUTTIGE D. SUMANA ABEYRATNE
© 2016 Pulukkuttige D. Sumana Abeyratne

A DISSERTATION SUBMITTED TO THE GRADUATE SCHOOL IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE DOCTOR OF PHILOSOPHY

DEPARTMENT OF PHYSICS

Dissertation Director:
Professor Bela Erdelyi
ACKNOWLEDGEMENTS

First and foremost I would like to thank my dissertation advisor, Prof. Bela Erdelyi, for giving me this excellent opportunity and having confidence in me to work on this challenging research problem. Thank you for patiently guiding me throughout my research work. Your availability and prompt responses to my questions are highly commendable. I received indispensable advice and guidance for writing research papers and giving presentations. I am thankful for the opportunities given to me to attend numerous conferences. It has been a privilege to have learnt from your expertise.

I would also like to thank Prof. Yasuo Ito for serving as a member of my dissertation committee and other progress review committees. His constructive comments always encouraged me. I wish to thank Prof. Philippe Piot for serving as a member of my dissertation committee and other progress review committees. His suggestions and constructive comments were always very helpful. I would also like to thank Dr. Hamed Alhoori for serving as a member of my dissertation committee.

Most importantly, I would like to thank my husband and my daughter with all my heart. It is no exaggeration to say that without you this work would not have been possible. Both of you were very supportive in all aspects. Thank you for your tolerance, care, and unconditional love for me. I wish to thank my dearest sister for her love and taking care of my parents during my absence. I would like to express my special gratitude to my parents for having raised me into the person I am today and leading by example. Both of you instilled in me the importance of education at a very young age.

My life-long friend, Karthi, deserves special recognition as well. Karthi, without your immense support and encouragement, it would have been impossible to complete this work.
Thank you for listening to me, sharing my ups and downs, and endless support in numerous ways.

I owe my sincere thanks to a few people from the Department of Computer Science for allowing me to use the cluster computer, Gaea. Dr. Caesar Ordonez, thank you for helping me launch my simulations and other technical supports. I would like to thank Dave Ulrick for responding to my emails and launching my applications promptly even in late hours. Thank you John Winans for making time to meet with me to resolve issues in my application. I would also like to thank Dr. Kirk Duffin and Dr. Nick Karonis for their help in numerous occasions.

I am especially grateful to Dr. Shashikant Manikonda for all his help during my visits to the Argonne National Lab. He extended his generous support and had very fruitful discussions with me at the beginning of this project. I am extremely thankful to Dr. Herman Schaumburg for his assistance in certain stages of my work. Also, I wish to thank Dr. He Zhang at Jefferson Lab for providing me with valuable information.

I would like to extend my special gratitude to Prof. Haowen Xi and Prof. Lewis Fulcher from Bowling Green State University, Prof. Lalantha Liyanage from the Open University of Sri Lanka and Prof. Mahawatta from University of Colombo, Sri Lanka for supporting and encouraging me to pursue graduate studies.

I would also like to acknowledge my friend, Chamadarie, for helping and encouraging me, and most importantly for our friendship.

I wish to thank the Department of Energy (DOE) for funding my research work. I would also like to thank IEEE, Northern Illinois University and the Jefferson Lab for providing me with travel grants to attend conferences (PAC, IPAC) and workshops (EIC14, COOL15).

My warm thanks also goes to the members in my research group and the members of the Department of Physics, especially my professors.
DEDICATION

This dissertation is dedicated to my parents.

Thank you Amma and Thaththa, I am forever grateful to you!
## CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>List of Tables</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>List of Figures</td>
<td>xii</td>
</tr>
<tr>
<td></td>
<td>List of Appendices</td>
<td>xviii</td>
</tr>
<tr>
<td></td>
<td>Chapter</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>Research Objective</td>
<td>4</td>
</tr>
<tr>
<td>1.2</td>
<td>Research Approach</td>
<td>5</td>
</tr>
<tr>
<td>1.3</td>
<td>Dissertation Organization</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>PROPOSED ELECTRON ION COLLIDER</td>
<td>8</td>
</tr>
<tr>
<td>2.1</td>
<td>Review of Beam Physics Concepts</td>
<td>8</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Emittance</td>
<td>8</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Betatron Function</td>
<td>9</td>
</tr>
<tr>
<td>2.1.3</td>
<td>Luminosity</td>
<td>12</td>
</tr>
<tr>
<td>2.2</td>
<td>Luminosity Limitations</td>
<td>13</td>
</tr>
<tr>
<td>2.3</td>
<td>Coulomb Scattering</td>
<td>14</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Coulomb Collisions</td>
<td>15</td>
</tr>
<tr>
<td>2.4</td>
<td>Electron Cooling</td>
<td>15</td>
</tr>
<tr>
<td>2.4.1</td>
<td>Approximate Derivation of the Cooling Force</td>
<td>18</td>
</tr>
<tr>
<td>2.5</td>
<td>Two New Technologies in MEIC</td>
<td>22</td>
</tr>
<tr>
<td>2.5.1</td>
<td>Energy Recovery Linac (ERL) and Circulator Ring (CR)</td>
<td>22</td>
</tr>
<tr>
<td>Chapter</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>3</td>
<td>FAST MULTIPOLE METHOD</td>
<td>24</td>
</tr>
<tr>
<td>3.1</td>
<td>Overview</td>
<td>24</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Different FMM Algorithms</td>
<td>26</td>
</tr>
<tr>
<td>3.1.2</td>
<td>The Novel Adaptive Multi-level FMM Algorithm</td>
<td>27</td>
</tr>
<tr>
<td>3.1.2.1</td>
<td>Terminology used in the FMM Algorithm</td>
<td>27</td>
</tr>
<tr>
<td>3.2</td>
<td>Basics of Differential Algebra</td>
<td>31</td>
</tr>
<tr>
<td>3.2.1</td>
<td>FMM in Differential Algebra</td>
<td>35</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Implementation of FMM using DA vectors</td>
<td>35</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Data Structure of the Novel FMM Algorithm</td>
<td>43</td>
</tr>
<tr>
<td>3.2.3.1</td>
<td>Universal Index $(n, l)$</td>
<td>44</td>
</tr>
<tr>
<td>3.2.3.2</td>
<td>Techniques of Bit Interleaving and Deinterleaving</td>
<td>49</td>
</tr>
<tr>
<td>3.2.3.3</td>
<td>The $D$-tree</td>
<td>58</td>
</tr>
<tr>
<td>3.2.3.4</td>
<td>The $C$-forest</td>
<td>59</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Maximum Level Reached</td>
<td>61</td>
</tr>
<tr>
<td>3.2.5</td>
<td>Parameter Optimization</td>
<td>63</td>
</tr>
<tr>
<td>3.2.5.1</td>
<td>Optimization of $q$</td>
<td>63</td>
</tr>
<tr>
<td>3.2.5.2</td>
<td>FMM order, $p$</td>
<td>66</td>
</tr>
<tr>
<td>3.2.5.3</td>
<td>Scaling with the error, $\varepsilon$</td>
<td>67</td>
</tr>
<tr>
<td>3.2.6</td>
<td>Runtime</td>
<td>69</td>
</tr>
<tr>
<td>3.2.7</td>
<td>Runtime Memory Footprint</td>
<td>71</td>
</tr>
<tr>
<td>3.2.8</td>
<td>Potential Measured with the New code</td>
<td>73</td>
</tr>
<tr>
<td>4</td>
<td>PARTICLES’ HIGH-ORDER ADAPTIVE DYNAMICS (PHAD)</td>
<td>82</td>
</tr>
<tr>
<td>4.1</td>
<td>Transition from FMM to Particles High-order Adaptive Dynamics (PHAD)</td>
<td>82</td>
</tr>
<tr>
<td>Chapter</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>4.2 N-body Problem</td>
<td>82</td>
<td></td>
</tr>
<tr>
<td>4.2.1 Electromagnetic N-body Problem</td>
<td>83</td>
<td></td>
</tr>
<tr>
<td>4.2.2 Fundamental Equations in Beam Dynamics</td>
<td>84</td>
<td></td>
</tr>
<tr>
<td>4.2.3 Lorentz Transformation</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>4.3 6N Differential Equations</td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>4.3.1 Scaling of Equations</td>
<td>95</td>
<td></td>
</tr>
<tr>
<td>4.3.1.1 Scaling of Equations with Time</td>
<td>96</td>
<td></td>
</tr>
<tr>
<td>4.3.1.2 Scaling of Equations with Momentum</td>
<td>97</td>
<td></td>
</tr>
<tr>
<td>4.3.2 Picard Integrator</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>4.3.3 Strang Operator Splitting</td>
<td>104</td>
<td></td>
</tr>
<tr>
<td>4.3.3.1 Preparation of Equations for Strang Splitting</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>4.3.4 Three Integrators in PHAD</td>
<td>109</td>
<td></td>
</tr>
<tr>
<td>4.3.4.1 Regular Long Range Integrator and the Picard Integrator</td>
<td>109</td>
<td></td>
</tr>
<tr>
<td>4.3.4.2 Modified Long Range Integrator</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>4.3.4.3 The FMM order in PHAD</td>
<td>117</td>
<td></td>
</tr>
<tr>
<td>4.3.4.4 Time Scales and Time Step Sizes</td>
<td>119</td>
<td></td>
</tr>
<tr>
<td>5 CODE DEVELOPMENT AND BENCHMARKING</td>
<td>122</td>
<td></td>
</tr>
<tr>
<td>5.1 Data Structuring</td>
<td>122</td>
<td></td>
</tr>
<tr>
<td>5.2 Coulomb Force Calculation</td>
<td>124</td>
<td></td>
</tr>
<tr>
<td>5.3 The N-body Code</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>5.3.1 Benchmarking the PHAD code against the N-body code</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>5.3.1.1 Particle Trajectories</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>5.3.1.2 Position and Momentum Plots</td>
<td>127</td>
<td></td>
</tr>
<tr>
<td>Chapter</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>5.4 MPI Based Parallelization of PHAD</td>
<td>132</td>
<td></td>
</tr>
<tr>
<td>5.4.1 Parallel Implementation</td>
<td>135</td>
<td></td>
</tr>
<tr>
<td>5.4.2 Parallelizing Data Structuring with Intel Cilk Plus</td>
<td>139</td>
<td></td>
</tr>
<tr>
<td>5.4.3 Parallelizable Procedures in the PHAD</td>
<td>140</td>
<td></td>
</tr>
<tr>
<td>5.4.4 Serial Procedures in the PHAD</td>
<td>141</td>
<td></td>
</tr>
<tr>
<td>5.4.5 Cluster Information</td>
<td>142</td>
<td></td>
</tr>
<tr>
<td>6 SIMULATION DETAILS, RESULTS AND APPLICATIONS</td>
<td>144</td>
<td></td>
</tr>
<tr>
<td>6.1 Energy Recovery Linac in JLEIC</td>
<td>144</td>
<td></td>
</tr>
<tr>
<td>6.2 Particle-based Proof of Principle Numerical Simulation of Electron Cooling</td>
<td>145</td>
<td></td>
</tr>
<tr>
<td>6.2.1 Parameter Estimation</td>
<td>145</td>
<td></td>
</tr>
<tr>
<td>6.2.1.1 Booster Ring</td>
<td>146</td>
<td></td>
</tr>
<tr>
<td>6.2.1.2 Collider Ring</td>
<td>146</td>
<td></td>
</tr>
<tr>
<td>6.2.1.3 Simulation Details</td>
<td>148</td>
<td></td>
</tr>
<tr>
<td>6.2.2 Selection of Optimum Parameters</td>
<td>148</td>
<td></td>
</tr>
<tr>
<td>6.2.3 Particle Limit $q$</td>
<td>149</td>
<td></td>
</tr>
<tr>
<td>6.2.4 Time Step For the Picard Integrator</td>
<td>149</td>
<td></td>
</tr>
<tr>
<td>6.2.5 Picard Order</td>
<td>151</td>
<td></td>
</tr>
<tr>
<td>6.2.6 FMM Order</td>
<td>153</td>
<td></td>
</tr>
<tr>
<td>6.2.7 Simulation 1: Proton Beam with Small Transverse Emittances</td>
<td>153</td>
<td></td>
</tr>
<tr>
<td>6.2.7.1 Emittance Calculation</td>
<td>154</td>
<td></td>
</tr>
<tr>
<td>6.2.8 Problems Discovered and Remedies</td>
<td>156</td>
<td></td>
</tr>
<tr>
<td>6.2.8.1 Electron Bunch Refreshing</td>
<td>156</td>
<td></td>
</tr>
<tr>
<td>6.2.8.2 Softening Parameter</td>
<td>157</td>
<td></td>
</tr>
<tr>
<td>Chapter</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>6.2.9</td>
<td>157</td>
<td></td>
</tr>
<tr>
<td>Different Initial Particle Configurations.</td>
<td>6.2.9.1 Protons and Electrons are Evenly Distributed</td>
<td>157</td>
</tr>
<tr>
<td>6.2.9.2 Proton Spread in 0.5 cm Radius</td>
<td>159</td>
<td></td>
</tr>
<tr>
<td>6.2.9.3 Proton Spread in 1cm Radius</td>
<td>159</td>
<td></td>
</tr>
<tr>
<td>6.2.10 Simulation 2: Protons with 0.1% Momentum</td>
<td>160</td>
<td></td>
</tr>
<tr>
<td>6.2.11 Longitudinal Magnetic Field</td>
<td>161</td>
<td></td>
</tr>
<tr>
<td>6.2.12 Simulation 3: Proton Beam with 0.1% Transverse Emittances and High Magnetic field</td>
<td>162</td>
<td></td>
</tr>
<tr>
<td>7 CONCLUSIONS</td>
<td>164</td>
<td></td>
</tr>
<tr>
<td>Bibliography</td>
<td>167</td>
<td></td>
</tr>
<tr>
<td>Appendices</td>
<td>172</td>
<td></td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>COSY INFINITY output.</td>
<td>34</td>
</tr>
<tr>
<td>3.2</td>
<td>The neighbor box set of box 50.</td>
<td>57</td>
</tr>
<tr>
<td>3.3</td>
<td>Distribution of targets among the $D$-leaf nodes.</td>
<td>60</td>
</tr>
<tr>
<td>3.4</td>
<td>Distribution of sources among the tree leaf nodes.</td>
<td>60</td>
</tr>
<tr>
<td>3.5</td>
<td>Optimum $q$ of uniformly distributed data sets of different sizes.</td>
<td>64</td>
</tr>
<tr>
<td>3.6</td>
<td>Relative error variation with the FMM order.</td>
<td>68</td>
</tr>
<tr>
<td>3.7</td>
<td>Runtime measured varying the FMM order with 1DM2L and 3DM2L for 800k particles. For higher orders ($&gt;5$) 3DM2L is faster.</td>
<td>71</td>
</tr>
<tr>
<td>3.8</td>
<td>Variation of peak runtime-memory with particle number, $N$.</td>
<td>73</td>
</tr>
<tr>
<td>3.9</td>
<td>Comparison of FMM runtime with direct runtime for different particle numbers, $N$, and different FMM orders.</td>
<td>80</td>
</tr>
<tr>
<td>4.1</td>
<td>Scaled quantities are denoted with the hat notation. $m$ and $q$ represent the mass and the charge of a proton, respectively.</td>
<td>95</td>
</tr>
<tr>
<td>4.2</td>
<td>The accuracy of the PHAD improves as the number of time steps increases by decreasing the time step size.</td>
<td>118</td>
</tr>
<tr>
<td>5.1</td>
<td>Information saved in an OctreeNode class object.</td>
<td>124</td>
</tr>
<tr>
<td>5.2</td>
<td>The description of three codes.</td>
<td>125</td>
</tr>
<tr>
<td>5.3</td>
<td>Parameters used in the $N$-body code and the PHAD code for the small time step size, 0.001 meters.</td>
<td>129</td>
</tr>
<tr>
<td>5.4</td>
<td>Parameter used in the $N$-body code and the PHAD code for the large time step size, 2 meters.</td>
<td>130</td>
</tr>
<tr>
<td>Table</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>5.5</td>
<td>Time taken by each part of the FMM.</td>
<td>133</td>
</tr>
<tr>
<td>5.6</td>
<td>Runtime measured for 1000 time steps by varying the number of nodes and the processors per node (ppn).</td>
<td>138</td>
</tr>
<tr>
<td>5.7</td>
<td>Time taken by each procedure for one time step in the serial version of the PHAD. Picard order is 7. The FMM order is 5. (N=1100).</td>
<td>141</td>
</tr>
<tr>
<td>5.8</td>
<td>Gaea Cluster’s Compute Node Specification.</td>
<td>143</td>
</tr>
<tr>
<td>5.9</td>
<td>ARC Cluster’s Compute Node Specification.</td>
<td>143</td>
</tr>
<tr>
<td>6.1</td>
<td>Parameters used in the simulation.</td>
<td>149</td>
</tr>
<tr>
<td>6.2</td>
<td>The runtime for different FMM orders and Picard orders.</td>
<td>152</td>
</tr>
<tr>
<td>6.3</td>
<td>Parameters used for Simulation 3.</td>
<td>162</td>
</tr>
</tbody>
</table>
**LIST OF FIGURES**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The proposed Jefferson Lab Electron Ion Collider.</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Upgrading of CEBAF to 12GeV.</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>Two-dimensional elliptical phase space of a beam.</td>
<td>11</td>
</tr>
<tr>
<td>2.2</td>
<td>Variation of the betatron function with the longitudinal displacement.</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>A single electron-ion collision. The ion moves with velocity $\vec{v}_i$ and scatters from an electron. The impact parameter is $b$.</td>
<td>20</td>
</tr>
<tr>
<td>2.4</td>
<td>Energy recovery linac (ERL) and the Circulator Cooler Ring (CCR) of JLEIC.</td>
<td>22</td>
</tr>
<tr>
<td>3.1</td>
<td>cell-particle or tree method: All particles in the box A can be represented by the single pseudo-particle in red.</td>
<td>26</td>
</tr>
<tr>
<td>3.2</td>
<td>particle-cell or cluster method: Interactions between all particles in box A with the pseudo-particle in the box B are calculated first.</td>
<td>27</td>
</tr>
<tr>
<td>3.3</td>
<td>cell-cell or fast multipole Method: The pseudo-particle in A interacts with the pseudo-particle in B.</td>
<td>28</td>
</tr>
<tr>
<td>3.4</td>
<td>At level 3, the total number of boxes (nodes) is 64. “BOX” is well-separated from boxes which have centers $\sqrt{2}b$ ($b$ is the side length of BOX) distance away from the center of BOX. The parent of BOX is the larger box containing BOX, 4, 5 and 6. The interaction list of BOX is shown in blue. Its neighbor boxes are shown in light gray.</td>
<td>29</td>
</tr>
<tr>
<td>3.5</td>
<td>Multipole expansion of the particle at $S$ is valid at points ($T$) only in the region outside the white boxes.</td>
<td>37</td>
</tr>
<tr>
<td>3.6</td>
<td>2D analog of the 3D computational domain depicting the upward pass and downward pass.</td>
<td>38</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>----------------------------------------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>3.7</td>
<td>Multipole-to-Local translation at level 2 and level 3 in a 2D layout. Interaction list boxes are labeled as ‘i’.</td>
<td>40</td>
</tr>
<tr>
<td>3.8</td>
<td>Local-to-Local translation takes place from parent to child node. The parent box (gray) translates the multipole expansion to the child box (blue) as shown by the white arrow.</td>
<td>41</td>
</tr>
<tr>
<td>3.9</td>
<td>Space-filling pattern with Morton order.</td>
<td>43</td>
</tr>
<tr>
<td>3.10</td>
<td>Hierarchical box numbering in 2D.</td>
<td>44</td>
</tr>
<tr>
<td>3.11</td>
<td>Index string for different boxes.</td>
<td>45</td>
</tr>
<tr>
<td>3.12</td>
<td>Box index 5 represents different boxes at different levels. The parent box number of the box 5 is always 1 and it is independent of the level. The box 14 at level 2 has 4 children (box numbers 56, 57, 58, 59) at level 3.</td>
<td>46</td>
</tr>
<tr>
<td>3.13</td>
<td>Given the $x, y, z$ coordinates of the particle, the box number at each level can be uniquely defined using bit-interleaving and bit-deinterleaving technique.</td>
<td>49</td>
</tr>
<tr>
<td>3.14</td>
<td>Bit interleaving to find the box number.</td>
<td>51</td>
</tr>
<tr>
<td>3.15</td>
<td>Box number at level 2 is 6.</td>
<td>52</td>
</tr>
<tr>
<td>3.16</td>
<td>Bit deinterleaving.</td>
<td>54</td>
</tr>
<tr>
<td>3.17</td>
<td>Neighbor boxes of the box 50 are shown in blue.</td>
<td>55</td>
</tr>
<tr>
<td>3.18</td>
<td>Highrachical subdivision of a root box containing 7 targets (empty blue squares) and 7 sources (filled blue squares). The $q$ value is 2. The maximum level reached is 4.</td>
<td>58</td>
</tr>
<tr>
<td>3.19</td>
<td>$D$-tree.</td>
<td>59</td>
</tr>
<tr>
<td>3.20</td>
<td>$C$-forest: There are five trees in the $C$-forest.</td>
<td>61</td>
</tr>
<tr>
<td>3.21</td>
<td>Number of trees in the $C$-forest due to two different particle distributions ($N=10^6$) as a function of $q$. When $q=1$, the number of trees is 56,622 and 61,700 in Gaussian and uniform distribution, respectively. Also, when $q=10^6$ the number of trees is 53 and 64 in Gaussian and uniform distribution, respectively.</td>
<td>62</td>
</tr>
<tr>
<td>Figure</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>3.22 Comparison of highest level reached in the two distributions ((N=10^6)) for different (q). For any given (q), the Gaussian distribution reached a higher level compared to that of the normal distribution.</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>3.23 Variation of the runtime with different (q) values of two different distributions ((N=100k)) at FMM order=2. The optimum (q) value gives the minimum runtime for data structuring.</td>
<td>65</td>
<td></td>
</tr>
<tr>
<td>3.24 Optimum (q) value for different size of data sets. FMM order=8.</td>
<td>66</td>
<td></td>
</tr>
<tr>
<td>3.25 Non-linear relationship between the runtime and the FMM order, (p), for Gaussian distribution of (N=300k).</td>
<td>67</td>
<td></td>
</tr>
<tr>
<td>3.26 Relative error of the potential calculated with different FMM orders for 1000 charged particles in a Gaussian distribution.</td>
<td>68</td>
<td></td>
</tr>
<tr>
<td>3.27 Variation of the relative error calculated between the average potential measured in the pairwise method and the FMM with the FMM order, (p).</td>
<td>69</td>
<td></td>
</tr>
<tr>
<td>3.28 Data structuring runtime and full FMM (data structuring and Coulomb potential calculation) runtime linearly scale with the number of particles, (N), in both uniform and Gaussian distributions. In each case, the (q) value is taken as 0.1(N).</td>
<td>70</td>
<td></td>
</tr>
<tr>
<td>3.29 Comparison of time taken by the direct method or pairwise calculation (red solid line) with those of full FMM at different FMM orders. When (N) is small, in order to calculate the potentials with high accuracy, the pairwise calculation is the best method.</td>
<td>70</td>
<td></td>
</tr>
<tr>
<td>3.30 Variation of the runtime with the FMM order for 1DM2L and 3DM2L in Gaussian distribution of (N=800k).</td>
<td>72</td>
<td></td>
</tr>
<tr>
<td>3.31 Runtime memory (measured using Valgrind 3.7.0) scales with the number of particles, (N).</td>
<td>72</td>
<td></td>
</tr>
<tr>
<td>3.32 Target-source configuration. Sources are derived from a single Gaussian distribution.</td>
<td>74</td>
<td></td>
</tr>
<tr>
<td>3.33 Single Gaussian potential and contour plots.</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>3.34 Target-source configuration. Sources are derived from a uniform distribution.</td>
<td>76</td>
<td></td>
</tr>
<tr>
<td>3.35 Uniform cube potential and contour plots.</td>
<td>77</td>
<td></td>
</tr>
</tbody>
</table>
Figure | Page
---|---
3.36 Target-source configuration. Sources are derived from a nine-Gaussian distribution. | 78
3.37 Nine Gaussian potential and contour plots. | 79
3.38 3D Contour plot. The FMM order is 2. | 80
3.39 2D Contour plots of the potential at different \( z \) values. | 81
4.1 The laboratory frame (unprimed) and the beam frame (primed) with \( z \) axes aligned. The particle beam moves at the velocity \( v\hat{\mathbf{z}} = \beta c\hat{\mathbf{z}} \). | 87
4.2 After applying the long range integrator, the source particles (small blue squares) in the neighborhood (blue boxes) and outside the neighborhood of the target (red square) are shown in (a). The Picard integrator changes the location of the source particles. By applying the modified integrator we can move the displaced sources back to the previous locations in order to keep the neighborhood unchanged. | 115
4.3 As the FMM order increases, the accuracy of PHAD increases. | 118
4.4 As the number of time steps increases (while decreasing the time step size), the accuracy of PHAD increases. | 119
4.5 Each Picard time step \( \Delta t_n \) is further divided into even smaller time steps of size \( \Delta (\Delta t_n) \). The total time \( T \) has \( n \) number of time steps. The red arrow indicates the direction of the time progression. | 120
5.1 Comparison of particle trajectories produced by: (a) the PHAD code with FMM order 6 and (b) the \( N \)-body code. The time step size is 0.5. The total time span is 200 units. | 126
5.2 The absolute error calculated for the \( x \) values measured in the PHAD and the \( N \)-body code. The FMM order used in the PHAD is 6. The time step size is 0.5. | 127
5.3 The absolute error calculated for the \( x \) values for FMM orders 2 and 12. The total number of time steps is 200 with time step size is 0.025 meters. | 128
5.4 \( z - x \) plot and the momentum in \( x - y \) plane \( (p_x - p_y) \) for the \( N \)-body code and the PHAD. All points in both plots overlap indicating that the results are in good agreement. The time step size is 0.001 meters. | 128
Figure

6.7 The $x - y$ cross section of the proton beam at the time step 132,000. The time step size is 0.004m. Particle 26 and 54 are outliers. The other 98 protons are at the top-right corner of the plot. This shows that 0.004 is a too large time step size........................................ 153

6.8 Schematic diagram of the particle bunch. 100 protons (red squares) and 1000 electrons (blue dots). ......................................................... 154

6.9 Emittance ($\epsilon_x$) of protons after $10^6$ time steps. The dashed black line is a reference line. The red line is the trend line to represent the behavior of horizontal emittance. ........................................... 155

6.10 Emittance ($\epsilon_y$) of protons after $10^6$ time steps. The dashed black line is a reference line. The red line is the trend line to represent the behavior of the vertical emittance. ........................................... 156

6.11 Cross section of the proton and the electron beam. Both types of particles are uniformly distributed on a disk of 1cm radius in the $x - y$ plane. 100 protons are in red and 1000 electrons are in blue..................... 158

6.12 Evenly distributed protons and electrons. Emittances of protons at the time step 132,000. The green and black dashed lines are reference lines, for $x$ and $y$ emittance values......................................................... 158

6.13 Maxima of emittance curves of protons in the transverse direction. ..... 159

6.14 Emittances of electrons at the time step 100,000 (with softening and electron bunch refreshing). The longitudinal magnetic field is 1 T......................... 160

6.15 Emittances of protons at the time step 100,000 (with softening and electron bunch refreshing). The longitudinal magnetic field is 1 T......................... 161

6.16 Emittances of protons for 1,005,000 time steps. (with softening and electron bunch refreshing). The longitudinal magnetic field is 1.5 T............. 163

6.17 Maxima of the vertical and the horizontal emittance plots. The negative slope on both plots indicates that the transverse emittance decreases with time.......................................................... 163

A.1 Rotation of the coordinate system to perform 3DM2L to 1DM2L translation. 173
# LIST OF APPENDICES

<table>
<thead>
<tr>
<th>Appendix</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Multipole-to-Local Translation in 1D and 3D</td>
</tr>
<tr>
<td>B</td>
<td>Algorithms</td>
</tr>
</tbody>
</table>
CHAPTER 1
INTRODUCTION

Thomas Jefferson Lab Electron Ion Collider, JLEIC (formerly named MEIC), proposed by the Jefferson Lab (JLab) will open the next Quantum Chromo-Dynamics (QCD) frontier for the nuclear science and atomic physics research (Fig. 1.1) [1, 2]. This has been identified as the highest priority in new facility construction by the federal Nuclear Science Advisory Committee (NSAC). A fundamental and conclusive goal of nuclear physics is understanding QCD, the theory that describes the strong interactions between quarks mediated by exchange of gluons, which is the ‘glue’ or the strong force that bind quarks inside nucleons. Understanding QCD is made possible by allowing a polarized electron beam to collide with a polarized proton beam or a polarized light ion beam at ultra-high luminosity over a broad range of energy. The JLEIC will be able to smash an electron with a moving proton or an ion. This process will enable physicists to probe the internal structure of the nucleus and its constituents or nucleons. Therefore, the JLEIC will pave the way for broader experiments to study sea quarks and gluons.

The first stage of the JLEIC is the Medium Electron Ion Collider (MEIC), and it will then be upgraded to the JLEIC. The collider will have a ring-ring configuration with figure-8 shape \(^1\). The upgraded Continuous Electron Beam Accelerator Facility (CEBAF) shown in Fig. 1.2 will provide polarized electrons \(^1\) while the ion complex will provide polarized ions or protons. The energy ranges of the electron beam and the proton beam for MEIC are 3-12 GeV and 25-100 GeV, respectively. It is for any other ion species varies as 12-40 GeV.

\(^1\)https://conferences.lbl.gov/event/56/session/1/contribution/9/material/slides/0.pptx
per nucleon. In the next stage for the JLEIC, the energy values of the electrons, protons, and ions are 20 GeV, 250 GeV and 100 GeV per nucleon, respectively.

As the ion or the proton beam travels around the ring, it will become ‘hot’ and expand in size. It is crucial to cool the ion or the proton beam to reduce its cross section and thereby to gain high luminosities required for these experiments. Luminosity is defined by

$$L = \frac{f_{\text{rev}} n_b N_1 N_2}{4\pi \sigma_x \sigma_y},$$

where $\sigma_x(y)$ is the horizontal (vertical) standard deviation of the spot size of the beam, $f_{\text{rev}}$ is the frequency of collision, $N_{1,2}$ is the number of particles in beam 1 and 2, $n_b$ is the number of bunches in each beam.

The JLEIC will provide a unique facility to generate high quality beams with high luminosity and small spot size, which is the main requirement for high precision measurements. The goal is to achieve ultra-high peak luminosity of $10^{34}$ cm$^{-2}$s$^{-1}$ in JLEIC. Achieving high luminosity is hindered due to several factors and the transverse emittance growth is one of them. The transverse emittance is directly proportional to the spot size or the transverse
cross section of the beam. Many factors such as collision between the particles in the beam, nonlinear forces due to fluctuations in the beam density profile, and instabilities arising due to external and beam generated electromagnetic fields have been identified as contributing to the emittance growth. Beam cooling techniques can be used to suppress the emittance growth and improve the lifetime of the beam.

There are different types of cooling techniques such as electron cooling, stochastic cooling, ionization cooling, and radiation cooling. Electron cooling is a method of cooling high energy ion beams via Coulomb interactions with low energy electron beams. This cooling technique has been identified as the most appropriate cooling technique for the MEIC [3, 4]. Electron cooling technique was first proposed by G. I. Budker in 1966 at the Institute of Nuclear Physics (INP) in Novosibirsk.
1.1 Research Objective

Simulating electron cooling involves calculating the Coulomb forces between the particles of the ‘cold’ electron beam and the particles of the ‘hot’ ion beam. If the forces are calculated precisely in the naive way—particle-to-particle direct evaluation—the computational complexity scales quadratically ($O(N^2)$) with number of particles, $N$. Even with today’s high performance computers, the direct calculation is prohibitively expensive and it becomes an intrinsic challenge for a large number of particles. Therefore, our research objective is to pursue new, faster methods to calculate interactions between particles. The main challenges we face are reducing the large computational time, handling large size of data sets, and calculating forces accurately. Electron cooling is a single point failure of the JLEIC. If the uncertainty is off by a factor of 2, the entire design will fail. It is an expensive machine with $1B price tag. Therefore, highly accurate and efficient method of Coulomb force calculation is required to reinforce the confidence and ensure the success of the project.

There are several codes developed for electron cooling: BETACOOL [5, 6], SimCool, OOPIC, VORPAL (now called VSIM$^2$). With these methods, electron cooling has been used successfully in some accelerator facilities. The Fermi Lab Recycler uses BETACOOL for numerical simulations of the cooling processes [7]. The Relativistic Heavy Ion Collider (RHIC) at the Brookhaven National Laboratory (BNL) uses BETACOOL and SimCool to simulate the cooling dynamics [8, 9, 10]. These codes find dynamic solutions to differential equations based on certain assumptions. The BETACOOL code uses Particle-In-Cell (PIC) method for calculating the force. The PIC method itself is inefficient to correctly calculate forces because it does not correctly treat the close encounters. However, electron cooling happens due to close encounters of particles and they cannot be undervalued. PIC method

\footnote{https://www.txcorp.com/products/OOPIC_pro/}
calculates only the average friction force between particles. Therefore, the need for developing a cooling process based on realistic distribution of particles has become important.

In addition to precise force calculation, another challenge we encounter is studying the beam dynamics with time advance. Advancing the simulation with large time steps hinder the important physics phenomena and can cause instabilities in the system. On the other hand, too small time steps degrade the efficiency. Therefore, we need to employ a time integrator with an optimized time step size. Also, handling large data sets is cumbersome. With advanced computer tools and high performance computers we can overcome all these challenges.

This dissertation addresses all the above mentioned challenges. We present new computation tools that we developed to calculate forces among a large number of charged particles interacting through Coulomb forces and that reduce the computational time smaller than ever before. This can be viewed as the investigation of many-body effects. Therefore, we can apply these new methods to any type of many-body system. Moreover, we introduce a new high performance computer code that allows, for the first time, a particle-based simulation of realistic electron cooling of heavy ion beams. It is scalable and reduces the runtime. Since it does not depend on assumptions, the results are also highly accurate.

1.2 Research Approach

Identifying the correct data structure for the new algorithm is the most important part of this implementation. There are different well known algorithms to reduce the computational time. Barnes-Hut method reduces the time to $O(N \log N)$, where $N$ is the number of particles in the distribution. Another method is Particle-In-Cell (PIC) method mentioned in 1.1. The computational cost of PIC is in the order of $O(M + N \log N)$, where $M$ is the number of
mesh points and \( M \ll N \) \cite{11, 12}. However, the PIC method cannot model highly non-uniform particle distributions or correlated systems. Therefore, these algorithms improve the efficiency at the loss of accuracy.

The Fast Multipole Method (FMM) is a fast algorithm, which scales linearly with the number of particles and it enables highly accurate evaluation of the potential and fields among a large number of particles using less memory compared to some other fast methods. It has been called one of the top ten algorithms developed in the 20\textsuperscript{th} century \cite{13}. The FMM was first introduced by L. Greengard and V. Rochlin in 1987. It is a numerical technique, which can calculate the pair-wise interactions between particles in a system of particles precisely and efficiently. The FMM itself has been developed with different algorithms. The single level fast multipole method has the computation cost of \( \mathcal{O}(N\log N) \). The drawback of this method is that it works for uniform distributions and cannot model non-uniform distributions like Gaussian distributions.

We developed a new computational tool—adaptive multi-level Fast Multipole Method based on a new algorithm \cite{14, 15}. This can model large systems from uniform or any complex distribution. It is adaptive with respect to source points and the evaluation points, and its runtime is of the order \( \mathcal{O}(N) \). Most importantly, this new code yields highly accurate results. We chose an octree data structure to represent 3D spaces since tree structures naturally represent object hierarchies. As we know that we need to apply this for a large number of particles and different complex distributions, we invested a substantial amount of time on parameter optimization. Details of the implementation and the performance analysis is given in Chapter 3.

Furthermore, with the adaptive Fast Multipole Method, we developed another high performance computer code that allows, for the first time, a particle-based simulation of realistic electron cooling scenarios of heavy ion beams—Particles’ High-order Adaptive Dynamics (PHAD) code. In this code, we used a numerical integrator called Picard Iteration Based
Integrator to propagate particles in time. Also, we used an operator splitting method known as Strang splitting. This is a second order accurate splitting method and it reduces the computational cost. Chapter 4 provides more details of these concepts.

We use this code for the electron cooling simulation of a proton beam of energy 280 MeV. Even though the developed tool produces fast and accurate results, we observed that it requires a longer simulation time to cool the proton beam. In order to overcome this challenge, we developed the MPI-parallel high performance computer code to speed up our simulations.

1.3 Dissertation Organization

The research work is presented in six self-contained chapters. Chapter 2 describes the proposed JLEIC, the definition of the beam temperature of charged particle beams and the luminosity concept. Chapter 3 is entirely devoted to explain the novel fast multipole method and its results. Chapter 4 contains the details of PHAD and the differential equations to be solved. Chapter 5 presents the details of the code development and the implementation of its parallel version. Chapter 6 discusses the results and applicability of this work to the proposed new electron cooler of MEIC at JLab. Finally, Chapter 7 includes the summary of this work and conclusions.
CHAPTER 2
PROPOSED ELECTRON ION COLLIDER

2.1 Review of Beam Physics Concepts

As a preface, very important terms used in beam physics need to be introduced.

2.1.1 Emittance

The volume of the six-dimensional phase space occupied by particles in a beam is called the beam emittance, $\epsilon$. Emittance is an important concept in beam physics, and it is a measure of the beam quality. One can define three two-dimensional emittances, and typically they are independent or weakly coupled. Two of these emittances are transverse emittances while the other one is the longitudinal emittance [16]. The direction of the beam propagation is known as the longitudinal direction, typically $z$ direction, and the other two directions perpendicular to the longitudinal direction ($x$ and $y$) are known as transverse directions. The area $A$ occupied by the beam can be represented by multiplying the numerical value of the 2D emittance by $\pi$. Hence, the area of the beam in the respective phase plane can be denoted by,

$$A = \pi \epsilon.$$  \hspace{1cm} (2.1)

Emittance can be given as the average transverse spread of the beam in the phase space, and it is invariant in the absence of a dissipative forces. Therefore, in the process of beam cooling the emittance can be reduced due to the cooling forces. That means by applying a
dissipative force the phase space density can be increased. The rms emittance of an ensemble of particles in the horizontal direction can be given by

\[ \epsilon_{x,\text{rms}} = \sqrt{\langle x^2 \rangle \langle x'^2 \rangle - \langle xx' \rangle^2}, \]  

(2.2)

where \( x \) is the position and \( x' = \frac{p_x}{p_z} \).  \( p_x \) and \( p_z \) are respectively the horizontal and the longitudinal momentum. The rms emittance in the vertical direction, \( y \), can be expressed by a similar expression. The second central moments can be defined by,

\[
\begin{align*}
\langle x^2 \rangle &= \frac{\sum x^2}{n} - \left( \frac{\sum x}{n} \right)^2 \\
\langle x'^2 \rangle &= \frac{\sum x'^2}{n} - \left( \frac{\sum x'}{n} \right)^2 \\
\langle xx' \rangle &= \frac{\sum xx'}{n} - \frac{\sum x \sum x'}{n^2}.
\end{align*}
\]  

(2.3)

### 2.1.2 Betatron Function

The equation of motion of a single particle in the beam through any beam transport system or an accelerator can be given by a linear second-order differential equation, which is of the general form known as Hill’s equation[17]. The differential equation of the particle’s motion can be written as [16],

\[ u'' + K(s)u = 0, \]  

(2.4)
where \( u \) denotes \( x \) or \( y \). \( K \) is a periodic function of the longitudinal position of the particle, which is the independent variable of the motion, and it nearly resembles the spring constant in the simple harmonic oscillation. The general solution of the equation takes the form,

\[
 u(s) = \sqrt{\epsilon} \sqrt{\beta(s)} \cos(\psi(s) + \delta),
\]

(2.5)

where \( \epsilon \) and \( \delta \) are integration constants. The first and the second derivative of Eq. 2.5 are

\[
 u'(s) = -\sqrt{\epsilon} \frac{\beta'}{2\sqrt{\beta}} \cos(\psi(s) + \delta) - \frac{\sqrt{\epsilon}}{\sqrt{\beta}} \sin(\psi(s) + \delta),
\]

(2.6)

where \( \beta(s) = \beta \).

\[
 u''(s) = -\sqrt{\epsilon} \beta' \beta - \frac{1}{2} \beta'^2 \cos(\psi(s) + \delta) - \frac{\sqrt{\epsilon} \beta'}{\sqrt{\beta}} \sin(\psi(s) + \delta) \psi'(s)
\]

\[
 - \sqrt{\epsilon} \sqrt{\beta} \sin(\psi(s) + \delta) \psi''(s) - \sqrt{\epsilon} \sqrt{\beta} \cos(\psi(s) + \delta) \psi'(s)^2.
\]

(2.7)

By substituting Eq. 2.7 into Eq. 2.4, the following equations can be obtained.

\[
 \frac{1}{2} (\beta \beta'' - \frac{1}{2} \beta'^2) - \beta^2 \psi'(s)^2 + \beta^2 K = 0.
\]

(2.8)

\[
 \beta' \psi'(s) + \beta \psi''(s) = 0.
\]

(2.9)

Eq. 2.9 yields

\[
 \beta(s) \psi'(s) = \text{constant},
\]

(2.10)

and by making the constant to be equal to 1, the betatron function can be given by

\[
 \beta(s) = \frac{1}{\psi'(s)}.
\]

(2.11)
With $\alpha = -\frac{1}{2} \beta'$, the Eq. 2.6 can be recast as

$$u'(s) = -\sqrt{\epsilon} \frac{\alpha}{\sqrt{\beta}} \cos(\psi(s) + \delta) - \frac{\sqrt{\epsilon}}{\sqrt{\beta}} \sin(\psi(s) + \delta). \quad (2.12)$$

By eliminating the trigonometric functions from Eq. 2.5 and 2.12

$$\gamma u(s)^2 + 2\alpha u(s)u'(s) + \beta u'(s)^2 = \epsilon, \quad (2.13)$$

where $\gamma = \frac{(1+\alpha^2)}{\beta}$. This is the well known Courant-Snyder invariant, and it gives the general equation for an ellipse with area $\pi \epsilon$. Therefore, the trajectory of a particle in a periodic lattice can be described by Eq. 2.13. The maximum amplitude of betatron motion $x_{\text{max}}$ is $\sqrt{\epsilon \beta}$ and the maximum divergence $x'_{\text{max}}$ is $\sqrt{\epsilon \gamma}$ as shown in Fig. 2.1.

Figure 2.1: Two-dimensional elliptical phase space of a beam.
Figure 2.2 shows that the beam radius is minimum at the waist. The betatron function $\beta^*$ at the interaction point can be expressed as [18]

$$\beta(s - s_0) = \beta^* + \frac{(s - s_0)^2}{\beta^*}. \quad (2.14)$$

![Diagram of betatron function with beam waist](image)

Figure 2.2: Variation of the betatron function with the longitudinal displacement.

### 2.1.3 Luminosity

Modern accelerators are competing to achieve high performance which is described in terms of luminosity, brightness etc. The luminosity $\mathcal{L}$, also known as the integrated luminosity, is considered the most prominent figure of merit in any accelerator to express its performance. The collision rate $R$ of particle is linked to the collision cross section or the
transverse beam size $\sigma$ via the luminosity as $R = \mathcal{L}(t)\sigma$. The definition of the instantaneous luminosity is,

$$L_{\text{inst}} = \frac{f_{\text{rev}}n_bN_1N_2}{4\pi\sigma_x\sigma_y},$$  \hspace{1cm} (2.15)

where $\sigma_{x,y} = \sqrt{\epsilon_{x,y}\beta^{*}_{x,y}}$, $\epsilon_{x,y}$ are the horizontal and vertical emittances, and $\beta^{*}_{x,y}$ are the horizontal and vertical beta functions at the collision point. $N_{1,2}$ denotes the number of particles (bunch population) in beam 1 and 2 while $n_b$ is the number of bunches in each beam. $f_{\text{rev}}$ is the frequency of revolution.

The instantaneous luminosity is related to the integrated luminosity through the relationship,

$$\mathcal{L} = L_{\text{inst}} = \int_0^t L_{\text{int}} \, dt.$$ \hspace{1cm} (2.16)

According to the Eq. 2.15, there are several parameters that can be manipulated to reach high luminosity. Compression of the beams is required to have a small $\beta^{*}_{x,y}$ at the point of collision. High bunch repetition rate of the two colliding beams and high current allow high luminosity. Clearly, the requirement of either high luminosity or high brightness entails the requirement of small emittance in the transverse planes. Small emittances are always required in high energy accelerators since it gives a small beam size, which is necessary to increases the probability of collisions between two opposing beams.

### 2.2 Luminosity Limitations

The luminosity lifetime in a collider ring decays due to emittance growth as described by Eq. 2.15. There are several factors such as intra-beam scattering, synchrotron radiation
damping, and collisions of the residual gas molecules in the beam pipe contribute to the emittance growth.

### 2.3 Coulomb Scattering

Collisions between the beam particles are known as Coulomb scattering, and there are two types, single Coulomb scattering and multiple Coulomb scattering.

The collision between two charged particle beams at the interaction point generates beam-beam effects. In this process a particle can be lost if it is deflected and travels out of the transverse acceptance or its momentum exceeds the longitudinal acceptance of the collider ring [19]. The collision of particles within a bunch also results single-beam losses. During this type of collision, the transverse momentum is transferred to the longitudinal momentum, and then it is enhanced by the relativistic gamma factor ($\gamma$). Finally, the particle acquires a large momentum in the longitudinal direction. If the particle’s energy exceeds the energy acceptance of the accelerator then both scattered particles will be lost, one particle with enormous energy and the other with too little energy [19], and this is the well known Touscheck scattering. This involves a single Coulomb scattering.

Intra-beam scattering occurs as a result of multiple Coulomb scattering. It involves many small-angle scattering events and leads to a diffusion in all three dimensions. Hence, the beam dimensions change and, as a result, the emittance growth of the beam occurs.

All the above mentioned factors contribute to the emittance growth. A tenuous population of scattered particles may cause enormous emittance growth.
2.3.1 Coulomb Collisions

The interaction of ‘cold’ electron beam with a ‘hot’ ion beam leads to a momentum transfer from ions to electrons. This will be discussed in 2.4.1. The momentum transfer causes emittance reduction of the ion beam. In other words, the ion beam gets cooled. The relationship between the emittance $\epsilon$ and the cooling (or heating) rate is defined by

$$\frac{1}{\tau} = \frac{1}{\epsilon} \frac{d\epsilon}{dt},$$

(2.17)

where $\tau$ is the cooling time. Hence, an accurate calculation of the Coulomb collision is needed to estimate the cooling time.

2.4 Electron Cooling

The phase space volume compression or an increase in the density of a particle beam in accelerators is commonly known as beam cooling. Studying of beam cooling to generate beams with low emittance has become increasingly important branch of beam physics. There are several cooling mechanisms. In radiative cooling, the accelerated charged particles lose energy by emitting radiation proportional to the square of the acceleration. Also, since the power radiated in this method is inversely proportional to the mass of the particle species it turns out to be important only for the lightest charged particles such as electrons. Ionization cooling has a very short cooling time. Therefore, it is the only cooling technique available for muon cooling since the lifetime of muon is very short (2.19 $\mu$s). For cooling of heavy particle beams, however, there are three well-known cooling techniques [4]: electron cooling, stochastic cooling and laser cooling. Intensive beams with low temperature can be cooled
using electron and laser cooling methods. On the other hand, stochastic cooling is efficient for compressing low-intensity with high temperature.

The information gleaned from literature review suggests that electron cooling is the best suited for cooling proton or heavy-ion beams [3, 4]. The fundamental importance of any cooling method is to reduce the temperature and the corresponding emittance. If the temperature of the beam is reduced, the transverse emittances as well as the momentum spread of the beam can be reduced. The velocity distribution of a particle beam is not spherical symmetric. Hence, there exists two local beam temperatures, longitudinal and transverse, and they can be defined as $T_\parallel$ and $T_\perp$, respectively.

\[
T_\parallel = \frac{M}{k_B} \left( \langle v_\parallel^2 \rangle - \langle v_\parallel \rangle^2 \right) \tag{2.18a}
\]

\[
T_\perp = \frac{M}{2k_B} \left( \langle v_\perp^2 \rangle - \langle v_\perp \rangle^2 \right) \tag{2.18b}
\]

where $k_B$ is the Boltzmann constant and $M$ is the mass of ion. Since the average longitudinal velocity is not zero and due to the rotational symmetry, which is the case for most of the practical beams, Eq. 2.18 can be further simplified with the rms longitudinal velocity spread $\langle \Delta v_\parallel^2 \rangle^{1/2}$ and the transverse rms velocity $\langle v_\perp^2 \rangle^{1/2}$ as

\[
T_\parallel = \frac{M}{k_B} \langle \Delta v_\parallel^2 \rangle. \tag{2.19a}
\]

\[
T_\perp = \frac{M}{2k_B} \langle v_\perp^2 \rangle. \tag{2.19b}
\]

The factor 2 in the denominator of Eq. 2.19b is the result of the two degrees of freedom in the transverse coordinates [20].

Electron cooling was first proposed by Prof. G. I. Budker in 1966 [21, 22] and experimentally confirmed at the INP in Novosibirsk in 1974 using a 65 MeV proton beam [23]. In storage rings, electron cooling is performed in a straight section of the ring [21].
intense, monoenergetic and parallel ‘cold’ electron beam is superimposed on the ‘hot’ heavy ion or proton beam \[24\]. The electron has a small momentum spread, and it co-propagates with the ion beam at the same average velocity. The idea is that the ion beam and the accompanying electron beam exchange ‘heat’ through Coulomb collisions between particles, and eventually the both type of particles will lead to temperature relaxation.

\[ \langle v_{\parallel, \text{ion}} \rangle = \langle v_{\parallel, e} \rangle. \]

The electron cooling process continues, as the result of Coulomb collisions, until the ions and electrons relative energies reach the same value. This suggests that \( v_e \approx v_{\text{ion}} \), and the temperature of the electron beam can be determined by

\[ T_e \approx \frac{m_e}{M_{\text{ion}}} T_{\text{ion}}, \tag{2.20} \]

where \( T \) denotes the temperature.

When cooling is achieved, the temperatures of the two beams become equal. Therefore,

\[ \frac{1}{2} m v_e^2 = \frac{1}{2} m v_{\text{ion}}^2. \tag{2.21} \]

Consequently, the angular spread of the ion beam becomes smaller than that of the electrons \[25\].

\[ \theta_{\text{ion}} = \sqrt{\frac{m_e}{M_{\text{ion}}}} \theta_e, \tag{2.22} \]

where \( \theta \) denotes the angular spread.
2.4.1 Approximate Derivation of the Cooling Force

The cooling force can be crudely estimated by considering the binary collision model [26]. As shown in Fig. 2.3, a single ion with charge $Ze$ and velocity $\vec{v}_\text{ion}$ approaches a single electron at rest and collides with it. The Coulomb force between them is

$$F = \frac{1}{4\pi\varepsilon_0} \frac{Ze^2}{x^2 + b^2}, \quad (2.23)$$

where $b$ is the impact parameter, $e$ is the elementary charge, $Z$ is the atomic number of the ion and $x$ is the distance traveled by the ion.

Therefore, the momentum gained by the electron is

$$\Delta p = \frac{1}{4\pi\varepsilon_0} \int \frac{Ze^2}{x^2 + b^2} dt. \quad (2.24)$$

If the limits of integration are taken from negative to positive infinity, the longitudinal component of the force, $F_\parallel$ will vanish. The transverse component of the force

$$F_\perp = \frac{1}{4\pi\varepsilon_0} \int \frac{Ze^2}{x^2 + b^2} \frac{b}{\sqrt{x^2 + b^2}} dt. \quad (2.25)$$

Since $v_\text{ion} = \frac{dx}{dt}$, Eq. 2.24 can be recast as

$$\Delta p = \int_{-\infty}^{+\infty} F_\perp dt = \frac{1}{4\pi\varepsilon_0} \frac{2Ze^2}{bv_\text{ion}}. \quad (2.26)$$

The amount of energy transferred from the ion to the electron is

$$\Delta E(b) = \frac{(\Delta p)^2}{2m_e} = \frac{2Z^2e^4}{(4\pi\varepsilon_0)^2m_e b^2 v_\text{ion}^2}, \quad (2.27)$$
where $m_e$ is the mass of the electron. Assuming that the ions pass through a $dx$ distance in an electron cloud of density $n_e$, the energy loss can be expressed by the *Bethe-Bloch* formula

$$ -\frac{\Delta E(b)}{dx} = 2\pi \int_0^x n_e \Delta E(b) b \, db = \frac{4\pi Z^2 e^4 n_e}{(4\pi \epsilon_0)^2 m_e v_{ion}^2} \int_0^x \frac{db}{b}. \quad (2.28) $$

By introducing the lower and upper limits of the impact parameter as $b_{\text{min}}$ and $b_{\text{max}}$, respectively

$$ \int_0^x \frac{db}{b} = \int_{b_{\text{min}}}^{b_{\text{max}}} \frac{db}{b} = \ln \left( \frac{b_{\text{max}}}{b_{\text{min}}} \right) = L_c. $$

$L_c$ is called the Coulomb logarithm. Typically, $L_c \approx 10$ [19]. The upper integration limit $b_{\text{max}}$ can be taken as the Debye shielding radius $\lambda_D$ defined by

$$ \lambda_D = \sqrt{\frac{k_B T}{4\pi m_e n_e r_e}}, $$

where $r_e$ is the classical electron radius and

$$ r_e = \frac{1}{4\pi \epsilon_0} \frac{e^2}{m_e c^2}. $$

Therefore,

$$ b_{\text{max}} = \lambda_D = \sqrt{\frac{k_B T}{4\pi m_e n_e r_e}} = \sqrt{\frac{\epsilon_0 k_B T e}{n_e e^2}}. \quad (2.29) $$

The minimal impact parameter $b_{\text{min}}$ can be determined by the maximum momentum transfer, which occurs at the head-on collision between the ion and the electron.

$$ \Delta p_{\text{max}} = 2 m_e v_{\text{ion}} $$

$$ b_{\text{min}} = \frac{1}{4\pi \epsilon_0} \frac{Z e^2}{m_e v_{\text{ion}}^2}. \quad (2.30) $$
The electron cooling force in the longitudinal direction can be given by the following standard expression \([9, 26]\), which represents the 3D integral over the velocity space of electron.

\[
F_{\text{cool}} = -\frac{4\pi Z^2 e^4 n_e L_e}{(4\pi\epsilon_0)^2 m_e} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} d^3v_0 f_e(v_0) \frac{\vec{v}_{\text{ion}} - \vec{v}_e}{(\vec{v}_{\text{ion}} - \vec{v}_e)^3},
\]

(2.31)

where \(f_e\) is the electron velocity distribution function, which has the form of an anisotropic Maxwell-Boltzmann distribution.

Figure 2.3: A single electron-ion collision. The ion moves with velocity \(\vec{v}_i\) and scatters from an electron. The impact parameter is \(b\).

Also,

\[
v = x' = \sqrt{\frac{\epsilon}{\beta}}
\]

(2.32)

\[
\epsilon = v^2 \beta.
\]

(2.33)
This yields
\[
\frac{1}{\epsilon} \frac{de}{dt} = \frac{2}{v} \frac{dv}{dt}.
\]
(2.34)

Therefore, using Eq. 2.17, the cooling rate in the electron beam frame can be defined by [3]
\[
\frac{1}{\tau_{\text{cool}}} = \left| \frac{2}{v_{\text{ion}}} \frac{dv_{\text{ion}}}{dt} \right|.
\]
(2.35)

In the laboratory frame, the cooling time can be given by Eq. 2.36 [3, 27].
\[
\tau_{\text{cool}} = \frac{\gamma^2 M_{\text{ion}}}{\eta_{\text{el}}} \frac{1}{m_e Z^2 \ell_{\text{lab}}^2 c^4 \rho L L_c} \begin{cases} 
\frac{1}{4\pi} v_{\text{ion}}^3, & \text{for } v_{\text{ion}} \gg v_{e,\text{rms}}, \\
\frac{3}{2\sqrt{2\pi}} \left(\frac{3 \kappa B T_e}{m_e}\right)^{3/2}, & \text{for } v_{\text{ion}} < v_{e,\text{rms}},
\end{cases}
\]
(2.36)

where the electron beam density in the lab frame is by \(\rho_L\), and \(\eta_{\text{el}}\) denotes the fraction of the cooling section relative to the circumference of the storage ring. According to Eq. 2.36, cooling is increasingly efficient for ions carrying a higher charge. The cooling rate for different types of particles can be expressed by [28, 29],
\[
\frac{1}{\tau_{\text{cool}}} \propto \frac{Z^2}{A},
\]
(2.37)

where \(A\) denotes the atomic mass. On the other hand, highly charged ions are capable of comfortably capturing cooling electrons. The rate of recombination can be expressed as [28]
\[
\frac{1}{\tau_r} \propto Z^2.
\]
(2.38)

The \(v_{\text{ion}}^3\) factor in Eq. 2.36 explains that the cooling time scales as the cubed of the ion beam velocity if the ion beam is ‘hot’ whereas the cooling time of ‘cold’ ion beams depends only on the electron beam temperature. Since the cooling time increases with \(\gamma^2\), the electron cooling is not suitable for high energy particle beams. At Fermilab’s Recycler storage ring,
electron cooling of 8 GeV anti-protons was performed [30]. The cooling time must be smaller than the time taken for intra-beam scattering.

2.5 Two New Technologies in MEIC

Two types of cooling facilities, low energy or DC cooler and the medium energy cooler, are employed in MEIC at various stages. The DC cooler technology is well established and those coolers are currently available to use. According to the conceptual design of the medium energy cooler, it will be based on two new technologies.

2.5.1 Energy Recovery Linac (ERL) and Circulator Ring (CR)

Figure 2.4: Energy recovery linac(ERL) and the Circulator Cooler Ring (CCR) of JLEIC.
The two new technologies are the Energy Recovery Linac and the Circulator Ring as shown in Fig. 2.4. The electron beam needed for cooling must possess good quality with two to three times larger current in comparison to the ion beam being cooled [1]. Since it needs to produce a high current beam, it requires SRF technology. To accelerate the electron beam, the SRF cavity invests a considerable amount of energy, and it is costly. The circulator ring concepts is used to circulate the same electron beam several times in the cooling section until it gets heated. Therefore, the average current of the electron beam can be reduced. As a result, the cost can be reduced by a factor equivalent to the number of circulations. Also, it prevents deleterious effects on the photo injector, which produces electrons, due to overuse and helps extending its effective lifetime.

The heated electron beam must be replaced by a new batch of electrons. The energy range of the ion or proton beam is the ultimate deciding factor of the choice of cooling device. For instance, if the proton beam energy is a few GeV, a single pass of the electron beam trough the cooling section would suffice. On the other hand, if the proton energy is high, the most appropriate mechanism is to use an electron beam from a storage ring. Hence, the temperature of the electron beam can be kept at a constant level by refreshing the heated electron by a fresh electron bunch. By means of the ERL, the energy invested to accelerate the electron beam can recovered by decelerating it before discarding. Therefore, the recovered energy can be subsequently used to accelerate the new batch.
CHAPTER 3

FAST MULTipoLE METHOD

3.1 Overview

The naive method to calculate the interaction among particles is the point-to-point calculation and it entails $O(N^2)$ computational complexity. This is sometimes referred as the direct method or the pair-wise method. For a large number of particles $N$, this method becomes exceedingly unfeasible since it requires an excessively large time period. Therefore, the necessity to seek other techniques arises and the goal is to compute interactions expeditiously. Calculation of Coulomb forces in a charged particle distribution is equivalent to solving the Poisson’s equation

$$\triangle \phi(\vec{r}) = -\frac{\rho(\vec{r})}{\epsilon}, \quad \phi(\infty) = 0,$$

where $\phi$, $\rho$, and $\epsilon$ denote the scalar potential function, the charge density and the electric permittivity, respectively. The solution to this differential equation with free boundary condition is

$$\phi(\vec{r}) = \frac{1}{4\pi\epsilon} \int \frac{\rho(\vec{r}')}{\|\vec{r} - \vec{r}'\|} d\vec{r}'.$$

There are three methods, basis function method, grid-based method, and hierarchical space subdivision method that are commonly used to seek an approximate solution to the Poisson’s equation. In the basis function method \cite{31, 32}, solving of the differential equation begins with the decomposition of the derivative of the function into a series of basis functions.
These derivatives can be integrated to find the solution to the original problem. However, the drawback of this method is finding the appropriate basis functions, especially when the charge distribution is inhomogeneous.

The grid-based method [33] assumes that the particles are deposited on a spatial grid. After solving the Poisson’s equation, the potential at any required location is obtained by interpolation. This method does not work accurately when the charge distribution is inhomogeneous.

In the hierarchical space subdivision method, space partitioning is performed without altering the original charge distribution. Hence, this method yields a solution, which is closer to the solution of the direct method. In this method, the underlying assumption is that the groups of particles are far away enough, then their interaction forces are smooth. Grouping of particles and finding their interactions can be done in three different ways. The first method is the cell-particle interaction, which is known as the tree method (Fig. 3.1). The second method is the particle-cell interaction method called the cluster method (Fig. 3.2). In this method, the particles in the distant cell is replaced by a pseudo-particle at the center of B. In the third method the interaction has a cell-cell interaction, and it is called the fast multipole method (Fig. 3.3).

The fast multipole method (FMM) is a numerically tractable technique developed to evaluate potential fields, and it was ranked as one of the top 10 algorithms in the 20th century [13]. It was discovered by Greengard and Rokhlin in 1987 [34]. The number of operations needed for the FMM is $\mathcal{O}(N)$. For large data sets, the difference in the speed is enormous.
3.1.1 Different FMM Algorithms

The fast multipole algorithm has different versions, viz., single level FMM, multi-level FMM, non-adaptive and adaptive multi-level FMM. Depending on the nature of the data distribution, one has to decide the most suitable FMM algorithm. The computational cost of the single level FMM is $O(N \log N)$. Tree-based algorithms (e.g. Barnes-Hut algorithm) known as tree-codes also need a computational cost of $O(N \log N)$. The hierarchical subdivision of the computational domain employed in the multi-level FMM enables to get rid of the $\log N$ factor and enhance the efficiency. The two shifting operators, multipole-to-multipole and local-to-local, introduced in the multi-level FMM are the contributing factors for this speed up. Since we encounter different types of data structures, the FMM must be appropriately modified to handle them. Therefore, the adaptive multi-level FMM was introduced to treat non-uniform data sets.

Figure 3.1: cell-particle or tree method: All particles in the box A can be represented by the single pseudo-particle in red.
3.1.2 The Novel Adaptive Multi-level FMM Algorithm

In order to implement our new algorithm we used an octree data structure to represent the particle space. Octree data structures are particularly useful in representing 3D spaces because each node can be broken into 8 child nodes, which closely represents partitioning a box at the center along three directions $x, y, z$ into 8 congruent boxes. The tree structure allows boxes to be recursively subdivided into smaller and smaller boxes. The two-dimensional equivalent of octrees are quadtrees.

3.1.2.1 Terminology used in the FMM Algorithm

We use terms commonly used in FMM algorithms.

Well-separated box:
Two boxes on the same level with side length $b$ are said to be well-separated if they are located with $b$ distance apart. Figure 3.4 shows level 3 boxes; all boxes apart from light gray
Figure 3.3: cell-cell or fast multipole Method: The pseudo-particle in A interacts with the pseudo-particle in B.

boxes are well-separated from the gray box (labeled “BOX”).

Parent box:
If any box at level $l$ can be further divided to reach the next level $l + 1$, then it is called a parent box. In Fig. 3.4, the BOX (at level 3) and boxes 4, 5, and 6 (all are at level 3) form a parent box at level 2.

Child box:
If any box at level $l$ belongs to a box at level $l - 1$, the box at level $l$ is called a child box. In 2D (3D), the parent box creates 4 (8) child boxes. If the side length of the parent box is $b$, then the side length of a child box is $\frac{b}{2^l}$, where $l$ is the level of the child box.

Neighbors and the neighborhood:
The neighbors of a given box is the set of boxes that shares at least one common boundary point and is at the same level as that particular box. In 2D there are 8 neighbors. In Fig. 3.4, the neighbors of BOX are shown in light gray. The neighborhood includes neighbors and the BOX itself and may contain up to 9 boxes in 2D and up to 27 boxes in 3D.

Interaction list:
The interaction list (up to 27 or 189 boxes in 2D or 3D, respectively) consists of the children boxes of the neighbors of its parent box, but not its own neighbors. The interaction list of BOX is shown in Fig. 3.4 in blue (27 boxes).

![Diagram](image)

Figure 3.4: At level 3, the total number of boxes (nodes) is 64. “BOX” is well-separated from boxes which have centers $\sqrt{2}b$ ($b$ is the side length of BOX) distance away from the center of BOX. The parent of BOX is the larger box containing BOX, 4, 5 and 6. The interaction list of BOX is shown in blue. Its neighbor boxes are shown in light gray.

In general, all FMM algorithms possess a similar computational structure. They start with a configuration of particles (both targets and sources) enclosed in a cube with side length 1. This is commonly known as the root box. Space partitioning starts with the root box. Targets are also referred to as observation points or evaluation points. In any type of FMM algorithm, there is a parameter that needs to be prescribed at the beginning that will determine the final depth of the tree. In certain algorithms, this parameter is denoted by $s$, and it is the maximum number of sources allowed in any node. The corresponding
parameter in our new algorithm is $q$, and it is the maximum number of sources allowed in the neighborhood of a particular target. The root box is divided into eight equal sub-regions (octants) and construct octrees. This partitioning continues until it meets the prescribed condition $q$ explained above. The original box is called the parent while the subdivisions are known as children. In general, partitioning will result $2^{bl}$ congruent boxes with equal volumes of side length $b/2l$, where $b$ and $l$ are the side length of the root box and the level, respectively. Subdividing of the root box takes place in a hierarchy of congruent boxes while identifying the neighbor boxes and the interaction list boxes of each resulting box.

A systematic categorization of the computational domain, which comprises of all target and the source points, to near and far regions arises as a result of hierarchical subdivision of the domain. The neighborhood boxes belong to the near region and they contribute to the near field. Similarly, the interaction list boxes belong to the far region and the far field arises due to them. The decomposing procedure repeats until it meets the prescribed condition $s$ or $q$.

Once the octree reaches the largest level, we can compute the multipole expansion of each box at the largest level. Then, it performs two passes: upward pass and downward pass. The former is implemented from the largest level to the level 2 while the latter is implemented from level 2 to the largest level. In the upward pass it starts traversing the tree towards the level 2 while gathering multipole expansions of the boxes at each level and reaching their next coarser level. This process is called the multipole-to-multipole (M2M) translation. Once level 2 is reached, the next pass or the downward pass starts. During this process, at each level, the accumulated multipole expansions are transformed to the boxes in the interaction list. This translation is called the multipole-to-local (M2L) translation. This local expansion must be added to the existing local expansion of that box. In the next step, this local expansion must be shifted to the next larger level or to the child node centers. This shifting is known as the local-to-local (L2L) translation. After this step, the local expansion
can be evaluated at each target point. When the downward pass is completed each box at each level possesses its local expansion which is due to the contribution of multipole expansion from the far region. In other words, the long range interactions on each particle are calculated. Then, the point-to-point (P2P) calculation must be performed in order to get the interaction due to close range particles. In our algorithm, this is also called the FMM DIRECT procedure. Finally, the total interaction is calculated by adding the long range interactions and the close range interactions.

3.2 Basics of Differential Algebra

The part of the code that calculates the Coulomb potential or the force is written in COSY INFINITY 9.1. COSY INFINITY is built on the concept of Differential Algebra (DA). We have extensively used transfer maps in our simulations, and Differential Algebra enables to compute the transfer maps efficiently [35]. Therefore, this section is devoted for a brief introduction to the basic concepts of Differential Algebra [36].

The simplest case of differential algebra is the first-order, one-variable structure denoted by $1D_1$. For the ordered pairs $(a_0, a_1), a_0, a_1 \in \mathbb{R}$, in the vector space $\mathbb{R}^2$, the addition and scalar multiplication can be defined as,

$$(a_0, a_1) + (b_0, b_1) = (a_0 + b_0, a_1 + b_1)$$

$$t \cdot (a_0, a_1) = (t \cdot a_0, t \cdot a_1),$$

where $a_0, a_1, b_0, b_1 \in \mathbb{R}$ and $t$ is a scalar. In addition to the two relationships given in Eq. 3.1, the multiplication between two ordered pairs is defined as,

$$(a_0, a_1) \cdot (b_0, b_1) = (a_0 \cdot b_0, a_0 \cdot b_1 + a_1 \cdot b_0).$$
According to this, we can establish the following result, and it is different from the result of the complex number multiplication.

\[(0, 1) \cdot (0, 1) = (0, 0).\]  \hspace{1cm} (3.2)

Therefore, an infinitely small element \(d\) can be defined as,

\[d \overset{\text{def}}{=} (0, 1).\]

and its square becomes zero according to Eq. 3.2. In the differential algebra structure, \(d\) is called the differential unit. Hence, the ordered pair, \((a_0, a_1)\), in \(1D_1\) can easily be written by means of \(d\) as \((a_0 + a_1 \cdot d)\), where \(a_0\) is the real part and \(a_1\) is the differential part.

The most generalized case of the algebra \(1D_1\) is the structure \(nD_v\). The space \(C^n(\mathbb{R}^v)\) comprises a set of \(n\) times differentiable functions of \(v\) variables on \(\mathbb{R}^v\). The collection of equivalence classes is called \(nD_v\). If \(a\) and \(b\) are two functions, where \(a, b \in C^n(\mathbb{R}^v)\), the following arithmetic operations are valid.

\[\begin{align*}
[a]_n + [b]_n &= [a + b]_n, \\
t \cdot [b]_n &= [t \cdot b]_n, \\
[a]_n \cdot [b]_n &= [a \cdot b]_n,
\end{align*}\]

where \(t\) is a scalar.

Furthermore, differential algebra supports integration and differentiation. The derivation operator for the partial derivative with respect to the variable \(v\) is given by \(\partial_v\) and the
corresponding anti derivative is $\partial^{-1}_v$. Therefore, $\partial$ along with the above arithmetic operations, turns algebra into differential algebra (DA). $\partial$ in $1D_1$ is defined by

$$\partial(a_0, a_1) = (0, 1).$$

It is easy to show that

$$\partial [(a_0, a_1) + (b_0, b_1)] = \partial(a_0, a_1) + \partial(b_0, b_1)$$

$$\partial [(a_0, a_1) \cdot (b_0, b_1)] = [\partial(a_0, a_1)] \cdot (b_0, b_1) + (a_0, a_1) \cdot [\partial(b_0, b_1)].$$

Therefore, we can conclude that $\partial$ denotes a derivation and $(1D_1, \partial)$ forms a differential algebra. By the same token, in the generalized case, $(nD_v, \partial_1, ..., \partial_v)$ is a differential algebra, where $\partial_k, k \in (1, v)$ is the derivative.

The multiplicative inverse can be defined as follows, if and only if $a_0$ is nonzero.

$$\begin{align*}
(a_0, a_1)^{-1} &= \left( \frac{1}{a_0}, \frac{-a_1}{a_0^2} \right).
\end{align*}$$

Also, the integration or the anti-derivative of a polynomial in DA can be performed in a very straightforward way by modifying its coefficients. For instance, if $a$ is a DA vector, then its integral $A$ can be given by the following expression.

$$\begin{align*}
a &= (a_0, a_1, a_2, ..., a_n) \\
A &= (c, a_0, \frac{a_1}{2}, \frac{a_2}{3}, ..., \frac{a_n}{n+1}).
\end{align*}$$
where \( c \) is a constant.

**Example**

If

\[
f(x) = 3.5x^6 + 5x^4 - 7.5x^2 + 3x + 12 \quad \text{with } f(0) = 0
\]

\[
\int_0^x f(x)dx = 0.5x^7 + x^5 - 2.5x^3 + 1.5x^2 + 12x + 0 \quad (3.3)
\]

Then the DA vector \( a \) can be defined as

\[
a = (12, 3, -7.5, 0, 5, 0, 3.5).
\]

Therefore, from Eq. 3.3 the integral \( A \) is

\[
A = (0, 12, 1.5, -2.5, 0, 1, 0, 0.5).
\]

The results obtained in COSY INFINITY is shown in Table 3.1.

**Table 3.1: COSY INFINITY output.**

<table>
<thead>
<tr>
<th>I</th>
<th>COEFFICIENT</th>
<th>ORDER</th>
<th>EXPONENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12.00000000000000</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1.500000000000000</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>-2.500000000000000</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1.000000000000000</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>0.500000000000000</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>
3.2.1 FMM in Differential Algebra

Our goal is to calculate the potential due to the ensemble of large number of particles, and we started by partitioning the domain into small cubes or boxes. If the electric potential generated at the position \( \vec{r} \) by a single point charge located at \( \vec{r}_i \) is described by

\[
\phi(r) = \frac{1}{|\vec{r} - \vec{r}_i|},
\]

we can refer to the observer’s location (also known as the target or the evaluation point) as \( \vec{r} \) and the source location as \( \vec{r}_i \), where \( i = 1, 2, ..., N \) and \( N \) is the total number of sources. In Cartesian coordinates, \( \vec{r}=(\vec{x},\vec{y},\vec{z}) \) and \( \vec{r}_i=(\vec{x}_i,\vec{y}_i,\vec{z}_i) \).

For simplicity, we can assume that the center of the box with side length \( b \) coincides with the origin \((0, 0, 0)\). In view of Eq. 3.4, if there are \( n \) sources in the box, in infinite free space, the potential generated at \((x, y, z)\) can be explicitly given by the summation

\[
\phi(x, y, z) = \sum_{1}^{n} \frac{1}{\sqrt{(x_i - x)^2 + (y_i - y)^2 + (z_i - z)^2}}.
\]

3.2.2 Implementation of FMM using DA vectors

We can apply multipole expansion for the potential given in Eq. 3.5, and it gives a series with different powers of DA variables. The DA variables themselves are functions of the coordinates of the observer’s location, \( x, y \) and \( z \). The convergence of the series of expansion can be achieved only if \( r \gg b \), where \( b \) is the side length of a box. In other words, the source and the target boxes must be well-separated from each other as shown in Fig. 3.5, and this requirement is considered in our data structure. The accuracy of the calculated potential
can be enhanced by including the higher order terms of the expansion (higher FMM order).

At this point, we can calculate the multipole expansion of the potential due to the sources in the box with its center at the origin, \((0, 0, 0)\). According to our FMM algorithm, we have to perform this multipole expansion only once for all the boxes at the finest (largest) level. We use COSY INFINITY to calculate the multipole expansion at any arbitrary point and to evaluate them at any given point. Once we have the multipole expansions at the finest level, the rest of the FMM can be implemented by simply translating them as required, and this will be discussed in the next section.

The potential can be expanded by means of DA variables, and these variables can be expressed in terms of the position of the target and distance to the target from the origin (Eq. 3.6).

For ease of illustration, we choose the 2D setup of the computational domain (Fig. 3.6). Let the multipole expansion of the potential at \(\vec{r}\) due to all the sources located at \(\vec{r}_i\) in the box \(i_1\) centered at \((x_0, y_0, z_0)\) be \(\phi_m\). Then,

\[
\phi_m = \phi(x, y, z) = \sum_i^n \frac{1}{\sqrt{(x_i - x)^2 + (y_i - y)^2 + (z_i - z)^2}},
\]

where \(n\) is the total number of sources in the box \(i_1\).

This multipole expansion can be expressed in terms of DA variables, and the DA variables can be explicitly given by the following expressions.

\[
DA(1) = \frac{x}{\sqrt{x^2 + y^2 + z^2}}, \\
DA(2) = \frac{y}{\sqrt{x^2 + y^2 + z^2}}, \\
DA(3) = \frac{z}{\sqrt{x^2 + y^2 + z^2}}.
\]
Figure 3.5: Multipole expansion of the particle at $S$ is valid at points $(T)$ only in the region outside the white boxes.

We can translate this multipole expansion to another box $i_2$ with the center $(x'_s, y'_s, z'_s)$, and this box will represent a new frame. The observer’s location with respect to the new frame is $(x', y', z')$, and the new box $i_2$ must also be well-separated from the observer. We can define the translation operator as the map, $T_{mm}$, which comprises of DA variables, and it shifts the center of the box with sources from $(x_0, y_0, z_0)$ to $(x'_s, y'_s, z'_s)$. Therefore

$$T_{mm} : \mathbb{R}^3 \rightarrow \mathbb{R}^3.$$

The translation of the multipole expansion, $\phi_m$, of the box $i_1$ centered at $(x_0, y_0, z_0)$ to the multipole expansion, $\phi_{m2m}$, of the box $i_2$ centered at $(x'_s, y'_s, z'_s)$ using the operator $T_{mm}$ is known as the multipole-to-multipole (M2M) translation in the FMM terminology, which was previously described. We can denote the multipole expansion of the $i^{th}$ box as a map $M_i$. The subscript $i$ spans over all boxes that need multipole expansions, and these boxes are
Figure 3.6: 2D analog of the 3D computational domain depicting the upward pass and downward pass.

identified by the FMM algorithm itself during the spatial subdivision of the domain. Let $M_{i_1,m}$ and $M_{i_2,m}$ be the corresponding maps of the multipole expansions of the two boxes, $i_1$ and $i_2$, respectively.

Mathematically, this translation process can be represented as the composition of the two maps.

$$M_{i_{2,m}} = M_{i_{1,m}} \circ \mathcal{T}_{mm}(i_1, i_2).$$

This translation requires a new set of DA variables, and the new and old DA variables are related through the distance between the centers of the boxes $i_1$ and $i_2$. 
The new DA variables can be expressed as in Eq. 3.7.

\[
DA'(1) = \frac{x - (x'_s - x_0)}{r'^2} = \frac{x'}{r'^2}
\]
\[
DA'(2) = \frac{y - (y'_s - y_0)}{r'^2} = \frac{y'}{r'^2}
\]
\[
DA'(3) = \frac{z - (z'_s - z_0)}{r'^2} = \frac{z'}{r'^2}
\]

(3.7)

where \( r'^2 = x'^2 + y'^2 + z'^2 \).

In fact, the purpose of M2M translation is to translate the multipole expansion of a child box to the center of its parent box. In 3D setup, we have a maximum of 8 children for each parent box, and the multipole expansion around the center of the parent box can be obtained by simply adding the multipole expansions of all the children. Therefore, we can assume that the two boxes, \( i_1 \) and \( i_2 \), are at levels \( l + 1 \) and \( l \), respectively to hold the parent-child relationship. \( T_{mm} \) is used in up-sweep or going up from finest levels to the coarser levels in the spatial domain until level 2 is reached.

Once the multipole expansion \( \phi_{m2m} \) at the center \( (x'_s, y'_s, z'_s) \) of the parent box is known, it can be translated to a far away box \( j_1 \) of the same level as the parent centered at \( (x'_c, y'_c, z'_c) \) using the multipole-to-local (M2L) translation technique (Fig. 3.7). The observer’s location is much closer to the new center \( (x'_c, y'_c, z'_c) \). Any box with a non-empty interaction list [37] is qualified to perform the multipole-to-local translation. The boxes in the interaction list are identified during the partitioning of the domain, and in 3D (2D) it has at most 189 (27) boxes. It is apparent that the multipole-to-local expansion is converting \( \phi_m \) to the location \( (x', y', z') \) in the new frame. Let this translation operator be \( T_{ml} \). Hence,

\[
T_{ml} : \mathbb{R}^3 \mapsto \mathbb{R}^3.
\]
Figure 3.7: Multipole-to-Local translation at level 2 and level 3 in a 2D layout. Interaction list boxes are labeled as ‘i’. M2L translation starts from level 2.

Here,

\[ x' = x - (x'_c - x'_s), \quad y' = y - (y'_c - y'_s), \quad z' = z - (z'_c - z'_s) \]

is the observer’s location w.r.t. the new frame. In addition, we can establish the following relationship.

\[ \mathcal{L}_{j_1} = \mathcal{M}_{i_2} \circ \mathcal{T}_{ml}(i_2, j_1), \quad (3.8) \]

where \( \mathcal{L}_{j_1} \) is the map corresponding to the local expansion of the potential of the box \( j_1 \).

The DA variables involve in this translation can be expressed as in Eq. 3.9.

\[ da'(1) = x - (x'_c - x'_s) \]
\[ da'(2) = y - (y'_c - y'_s) \]
\[ da'(3) = z - (z'_c - z'_s) \]

\[ (3.9) \]
In this translation, the DA variables and the coordinates with the prime notation ($t'$) are in the new frame.

In the entire FMM procedure, the M2L translation has been identified as one of the costly calculations, and we delved deeper into the causes for its long runtime. We implemented the M2L procedure with two different options: 1DM2L and 3DM2L. The former is a one dimensional translation that must be employed for FMM orders greater than 5 to enhance the efficiency (Fig. 3.30). The results obtained for 800k particles are shown in Table 3.7. This transformation will be discussed in detail in Appendix A.

The local expansion at the box center ($x'_c, y'_c, z'_c$) of the parent box $j_1$ needs to be transferred to the child box(es) $j_2$, and it is called the local-to-local (L2L) translation (Fig. 3.8), and it can be denoted by Eq. 3.10.
\[ L_{j_2} = L_{j_1} \circ T_{ll}(j_1, j_2), \]  
(3.10)

where \( T_{ll} \) is the local-to-local translation operator. Also,

\[ T_{ll} : \mathbb{R}^3 \mapsto \mathbb{R}^3. \]  
(3.11)

If the center of the child box \( j_2 \) is \((x'_cc, y'_cc, z'_cc)\), the observer or the target at \((x', y', z')\) is now much closer to the child box center.

In contrast to the up-sweep, the operators \( T_{ml} \) and \( T_{ll} \) are used in the down-sweep or traversing down from level 2 to the finest level in the spatial domain. If the box \( j_2 \) encompasses any target points, the local expansion \( L_{j_2} \), which is a DA vector involving spatial coordinates of any arbitrary target point, must be evaluated at each target to get the potential at that location. Let \( E_{j_2}(P) \) be the local expansion evaluated at the target point \( P \) in box \( j_2 \). Then

\[ E_{j_2}(P) : \mathbb{R}^3 \mapsto \mathbb{R}. \]

\( j_2 \) spans all boxes in the child box set with local expansions.

In fact, each leaf node of each tree in the C-forest (Fig. 3.20) gets potentials from the multipole expansion, the multipole-to-multipole translation, the multipole-to-local translation, and the local-to-local translation. However, each potential is declared at the center of the leaf node as described above. Therefore, once we know the potential at the center, the potential at each target in that leaf node can be evaluated. As the final step of the FMM algorithm, this potential value at each target must be modified by adding the point-to-point potential evaluation due to the sources in the neighborhood of the target. In 3D, each target box can be surrounded by 27 neighboring boxes at most.
3.2.3 Data Structure of the Novel FMM Algorithm

We have developed a 3D FMM code which can be utilized for N-body potential/field calculations. This can be categorized as an fully adaptive fast multipole method. This code can be modified for the 2D version. The code development is fully discussed in chapter 5.

The main task of implementing the FMM is identifying the suitable data structure. Grouping of particles in the data set, which is common to all FMM algorithms, can be performed by means of the data structures automatically and efficiently if it is chosen properly. Our novel algorithm requires an octree type data structure (in 2D quadtree type) with a branching factor equal to $2^d$, where $d$ is the dimensionality. An octree is a tree data structure in which each internal node comprises of eight children. Each node of the octree represents a box. Establishing an octree to save all necessary information makes the code run fast. In the 3D configuration, each particle is identified by its coordinates $x, y, z$, and it belongs to different boxes at different levels. By means of Morton indexing introduced in 1966 by G. M. Morton [38], each box can be assigned a universal index $(n, l)$, where $n$ and $l$ represent the index and the level of the box, respectively (Fig. 3.13). Space-filling with Morton order (also known as z-order) is shown in Fig. 3.9.

![Image showing Morton indexing](image)
3.2.3.1 Universal Index \((n,l)\)

Since the branching factor is \(2^d\), it is apparent that each parent box has \(2^d\) children. The children set can be numbered as \(0, 1, 2, ..., 2^d - 1\). Therefore, in 2D there are 4 children with numbers 0, 1, 2, 3 as shown in Fig. 3.10. The box index can be represented by a string of numbers. The string representation of the box \((n,l)\) is [14]

\[
\text{string}(n, l) = (N_1, N_2, ..., N_l) \quad N_l = 0, ..., 2^d - 1, \quad j = 1, ..., l, \quad (3.12)
\]

where \(N_j\) denotes the box index at level \(j\). For instance, the index strings of the boxes shown in Fig. 3.11 are

\((1, 1)\) for the yellow box
\((3, 2, 3)\) for the gray box
\((0, 1, 1)\) for the blue box.

Figure 3.10: Hierarchical box numbering in 2D.
However, it is customary to drop the first 0, which represents the root box. Using this string, we can form a single number \( n \).

\[
n = (2^d)^{l-1}N_1 + (2^d)^{l-2}N_2 + \ldots + (2^d)N_{l-1} + N_l.
\]  

(3.13)

By applying the Eq. 3.13 to the gray box in Fig. 3.11 at level 3, we can get the box number 59. The index strings of that box is (3, 2, 3).

\[
n = (2^2)^3 - 1 \times 3 + (2^2)^3 - 2 \times 2 + (2^2)^3 - 3 \times 3 = 59.
\]

Since the same box index represents different boxes at different levels, the box index by itself is not sufficient to uniquely identify a particular box. The box index 5 appears at both level 2 and level 3, but they represent two different boxes as shown in Fig. 3.12. Hence, the box index and the level must be used in the universal index. The yellow box at level 2 is
(5, 2) while the blue box at level 3 is (5, 3).

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>13</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>8</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>21</th>
<th>23</th>
<th>29</th>
<th>31</th>
<th>53</th>
<th>55</th>
<th>61</th>
<th>63</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>22</td>
<td>28</td>
<td>30</td>
<td>52</td>
<td>54</td>
<td>60</td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>19</td>
<td>25</td>
<td>27</td>
<td>49</td>
<td>51</td>
<td>57</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>18</td>
<td>24</td>
<td>26</td>
<td>48</td>
<td>50</td>
<td>56</td>
<td>58</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>13</td>
<td>15</td>
<td>37</td>
<td>39</td>
<td>45</td>
<td>47</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>12</td>
<td>14</td>
<td>36</td>
<td>38</td>
<td>44</td>
<td>46</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>9</td>
<td>11</td>
<td>33</td>
<td>35</td>
<td>41</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>8</td>
<td>10</td>
<td>32</td>
<td>34</td>
<td>40</td>
<td>42</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 3.12:** Box index 5 represents different boxes at different levels. The parent box number of the box 5 is always 1 and it is independent of the level. The box 14 at level 2 has 4 children (box numbers 56, 57, 58, 59) at level 3.

### Parent Box

In terms of Eq. 3.13, the parent of box \( n \) can be given by,

\[
\text{Parent}(n) = (2^d)^{l-2}N_1 + (2^d)^{l-3}N_2 + \ldots + N_{l-1}.
\]  

(3.14)

The number which represents the highest level, \( N_l \), is excluded in the parent index string.

For any given box number, the parent box number is the same, and it is independent of the level. The box 5 at level 2 (yellow box) and the box 5 at level 3 (blue box) have the same parent box number, 1.

Using the string representation on both sides [14]

\[
\text{Parent}(N_1, N_2, \ldots, N_{l-1}, N_l) = (N_1, N_2, \ldots, N_{l-1}).
\]  

(3.15)
According to the Eq. 3.14, the box 59 at level 3 has its parent at level 2, which bears the number 14.

\[ \text{Parent}(59) = (2^2)^{3-2} \times 3 + (2^2)^{3-3} \times 2 = 14. \]

The general expression to find the parent index is

\[ \text{Parent}(n, l) = (\text{Parent}(n), l - 1). \quad (3.16) \]

Finding the parent box index can be found by performing right bit shifting on the binary representation of the box index. The right bit shifting operation is equivalent to dividing the box number by \(2^d\) and taking the integer part of that.

For instance, the parent of box 59 at level 3 is

\[ \left\lfloor \frac{59}{2^2} \right\rfloor = 14 \text{ at level 2}. \]

**Children**

Each box at level \(l\) has a set of \(2^d\) children, and they belong to the next finer level \(l + 1\). Therefore, the index string of the parent box must be modified by adding an extra number.

The **ChildrenSet** includes all \(2^d\) child nodes of a particular box.

\[ \text{ChildrenSet}(N_1, N_2, ..., N_l) = ((2^d)^l N_1 + (2^d)^{l-1} N_2 + ... + (2^d) N_l + N_{l+1}) \quad (3.17) \]

with \(N_{l+1} = 0, 1, ..., 2^d - 1\).

Using the string representation on both sides

\[ \text{ChildrenSet}(n) = ((2^d)^l N_1 + (2^d)^{l-1} N_2 + ... + (2^d) N_l + N_{l+1}), \quad N_{l+1} = 0, 1, 2^d - 1. \quad (3.18) \]
The general expression to find the children set indices is

$$\text{ChildrenSet}(n, l) = (\text{ChildrenSet}(n), l + 1).$$  \hspace{1cm} (3.19)

The children set of the box 14 at level 2, shown in Fig. 3.12, can be determined by Eq. 3.18.

$$n = 14 = (3, 2)$$

$$N_{l+1} = 0, 1, 2, 3.$$  

$$\text{ChildrenSet}(3, 2) = \begin{cases} 
((2^2)^2 \times 3 + (2^2)^2-1 \times 2 + 0) = 56 \\
((2^2)^2 \times 3 + (2^2)^2-1 \times 2 + 1) = 57 \\
((2^2)^2 \times 3 + (2^2)^2-1 \times 2 + 2) = 58 \\
((2^2)^2 \times 3 + (2^2)^2-1 \times 2 + 3) = 59.
\end{cases}$$  \hspace{1cm} (3.20)

Left-bit shifting can be used to find the children set of a particular box. This operation is equivalent to multiplying the box number by $2^d$. Then by adding all possible values of $N_{l+1}$ the box numbers can be obtained.

Therefore, the box number of the box 14 at level two has children, $14 \times 2^2 + 0 = 56$, $14 \times 2^2 + 1 = 57$, $14 \times 2^2 + 2 = 58$ and $14 \times 2^2 + 3 = 59$.

The particle coordinates $x, y, z$ can be represented in binary numbers, and then by using the technique of bit interleaving and deinterleaving one can precisely determine the index of the box or the box number at a given level [14]. Accordingly, the universal index of the root box is $(0, 0)$. 


3.2.3.2 Techniques of Bit Interleaving and Deinterleaving

Bit-interleaving

For a given particle point, the box number at a given level can be found by employing the bit-interleaving method. The first step of this is the coordinate scaling explained below. Taking into account $x, y, z$ coordinates of all sources and targets, we can determine the overall minimum $x_{\text{min}}$ and the overall maximum $x_{\text{max}}$. $x$ denotes $x, y, \text{ or } z$. The spread of spatial coordinates in any dimension $L_d$ is

$$L_d = x_{\text{max}} - x_{\text{min}},$$  \hspace{1cm} (3.21)

where $d$ represents the three dimensions. Hence, $d = 1, 2, 3$. If $D_0$ is defined as

$$D_0 = \max \{L_d\},$$  \hspace{1cm} (3.22)
Then the true size of the box is $D_0 \times D_0 \times D_0$ [14]. In order to map this box to a box of unit length the following scaling must be performed.

$$\tilde{x} = \frac{x - x_{\text{max}}}{D_0}.$$  (3.23)

This scaling guarantees that the all coordinates, $x, y, z$, are less than or equal to one. Hence,

$$\tilde{x} \in (0, 1).$$

After scaling the coordinates of a particular point, we can determine the number of the box contains that point at any given level.

**Example - Finding the index of the box given the coordinates of a point**

If the scaled coordinates of a point in a 2D configuration are $x_1 = (0.375)_{10}$, $x_2 = (0.625)_{10}$, then their binary representations can be given as $x_1 = (0.011)_2$, $x_2 = (0.101)_2$. It is apparent that, if the dimensionality is $d$, then $d$ coordinates are required to characterize each point. By ordered mixing of the binary digits in the $d$ coordinates, a single number can be created to represent any point, and this is called bit interleaving. In order to find the box number to which this point belongs, bit shifting must be performed. If the required level is $l$, then $l$ positions of $\bar{x}$ must be left-shifted and take the integer part of that (Fig. 3.14). For $l = 2$,

$$\bar{x} = (0.N_1, N_2, N_3) \rightarrow \bar{x} = (N_1 N_2. N_3)$$

$$N_1 N_2 = (0110)_2 = 6.$$
Therefore, $x_1 = (0.375)$, $x_2 = (0.625)$ point is in the box 6 at level 2 (Fig. 3.15). In addition, this bit shifting is equivalent to multiplying $\bar{x}$ by $2^d l$. Hence, the box number and the level in the universal index notion can be expressed as

$$(n, l) = (2^d l, \bar{x}).$$

Accordingly,

$$(2^d l, \bar{x}) = (2^l, (0.011011))_2$$

$$= 0110.11.$$ 

Therefore, the integer part is 6. Also, Fig. 3.15 justifies that the point (0.375, 0.625) belongs to the box 6.

**Bit-deinterleaving**

Decomposing the box index $n$ into groups of $d$ bits is known as bit-deinterleaving. As usual $d$ denotes the dimension. Decomposition must start from the last digit.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$ = 0.</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$x_2$ = 0.</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$\bar{x}$ = (0.</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$\bar{x}$ = (0.</td>
<td></td>
<td>$N_1$</td>
<td>$N_2$</td>
</tr>
</tbody>
</table>

Figure 3.14: Bit interleaving to find the box number.
**Example - Finding the center of the box given the box index**

The bit-deinterleaving technique can be used to find the center of a given box. The box number $n$ can be represented in binary notation as $n = (b_{11}b_{21} \ldots b_{d1}b_{12}b_{22} \ldots b_{d2} \ldots b_{1l}b_{2l} \ldots b_{dl})_2$, \( (3.24) \)

where $l$ is the level.

After bit-deinterleaving into groups of $d$ ($d=2$ for 2D), it will result $d$ coordinates, $\mathbf{x}_d$. Therefore,

$$
\begin{align*}
\mathbf{x}_1 &= (b_{11}, b_{12}, \ldots, b_{1d})_2 \\
\mathbf{x}_2 &= (b_{21}, b_{22}, \ldots, b_{2d})_2 \\
& \vdots \\
\mathbf{x}_d &= (b_{d1}, b_{d2}, \ldots, b_{d2d})_2.
\end{align*}
$$

\( (3.25) \)
Since certain values of the bit string can be zero and the length of the each bit string must be $dl$, it is required to add zeros before the first $b$ value of the string. In order to get the center coordinates of the box at level $l$, we perform $l$ bit shift and then add the digit 1 to the end of the bit string. Hence, the general expression for the center coordinates is,

$$x_m(n, l) = (0.b_{m1}b_{m2} \ldots b_{ml}),$$

where $m = 1, \ldots, d$. For instance, the center of the box number 14 at level 2 can be found using the bit-deinterleaving technique.

$$n = 14 = 1110.$$  

After de-interleaving,

$$n = 14 = (\overbrace{\begin{array}{c}
11 \\
\text{coordinate 1}
\end{array}}^{\text{coordinate 1}}, \overbrace{\begin{array}{c}
10 \\
\text{coordinate 2}
\end{array}}^{\text{coordinate 2}})_2.$$  

For the two dimensional setup, $d=2$. Then,

$$x_1 = (11)_2,$$

$$x_2 = (10)_2.$$  

After bit shifting and adding 1,

$$x_1 = (0.111)_2,$$

$$x_2 = (0.101)_2.$$  

The box center in decimal notation is

$$x_1 = (0.875)_{10},$$

$$x_2 = (0.625)_{10}.$$
and it agrees with Fig. 3.16.

Example - Finding the neighbors of box given the box index and the dimension

We can find the neighbor boxes of box 50 at level 3 using the bit-interleaving and the bit-deinterleaving processes. The number of neighbors varies with dimension. The maximum number of neighbors including the self box in 1D, 2D and 3D are 3, 9 and 27, respectively. In general, any box, which is not closer to the boundary of the root box, has $3^d$ number of neighbors including the box itself. We can find the binary representation of the index of a given box $(n, l)$ according to Eq. 3.24. After bit deinterleaving as in Eq. 3.25,

$$n = (n_1, n_2, \ldots, n_m, \ldots, n_d) \quad m = 1, \ldots, d.$$
Figure 3.17: Neighbor boxes of the box 50 are shown in blue.

Therefore, for each coordinate we have \( n_m \). The set of indices for the neighbor boxes are defined as [14]

\[
\begin{align*}
    n^+_m &= n_m + 1 \\
    n^-_m &= n_m - 1.
\end{align*}
\] (3.26)

If any of these indices are not in the range of \([0, 2^l - 1]\), it must be discarded. Fig 3.17 shows that all box indices at level 3 are in the range of \([0, 7]\).

\[
Neighbor_m(n, l) = \begin{cases}
    \{n^-_m, n_m, n^+_m\}, & \text{for } n_m \neq 0, 2^l - 1, \\
    \{n_m, n^+_m\}, & \text{for } n_m = 0, \\
    \{n^-_m, n_m\}, & \text{for } n_m = 2^l - 1, \quad \text{where } m = 1, \ldots, d.
\end{cases}
\] (3.27)
Using the elements in Eq. 3.27, the set of indices that generates neighbor boxes can be given by,

$$\nu = (\nu_1, \nu_2, ..., \nu_d), \text{ where } \nu_m \in \text{Neighbor}_m, \quad m = 1, ..., d.$$  \hspace{1cm} (3.28)

If \(n_m\) is represented in binary, it results

$$\nu_1 = (\nu_{11}, \nu_{12}, ..., \nu_{1l})_2, \quad \nu_2 = (\nu_{21}, \nu_{22}, ..., \nu_{2l})_2, \quad \nu_d = (\nu_{d1}, \nu_{d2}, ..., \nu_{dl})_2,$$  \hspace{1cm} (3.29)

$$\nu_{mj} = 0, 1 \text{ where } m = 1, ..., d \text{ and } j = 1, ..., l.$$ 

By bit interleaving, we can form the bit strings for all neighbor indices. Hence,

$$\text{Neighbors}(n, l) = \{ (\nu_{11}\nu_{21}...\nu_{dl1}\nu_{12}...\nu_{dl2}...\nu_{1l}\nu_{2l}...\nu_{dl})_2 \}.$$  \hspace{1cm} (3.30)

Since the length of the bit string varies for different \(m\), the length \(l\) must be made equal by adding zeros to the beginning of each string.

By following the above procedure, we can find the neighbors of the box 50 at level 3.

$$n = 50 = (110010)_2.$$ 

Since the dimension \(d = 2\), bit-deinterleaving results,

$$n = (\begin{array}{c}
101 \\
\text{coordinate 1}
\end{array}, \begin{array}{c}
100 \\
\text{coordinate 2}
\end{array})_2 = \text{number 1, number 2}_10 = (x_1, x_2)_2 = (5, 4)_{10}$$

In Fig. 3.17, box 50 is indicated by (5, 4). By shifting the coordinate numbers,

$$5 \rightarrow \{ 5-1, 5, 5+1 \} = \{ 4, 5, 6 \}.$$
The neighbor box set can be generated by all possible combinations of the above two sets of numbers. By converting them to the binary notation followed by the bit-interleaving techniques, the neighbor box indices of the box 50 can be recovered.

Table 3.2: The neighbor box set of box 50.

<table>
<thead>
<tr>
<th>Box</th>
<th>Binary</th>
<th>Bit-interleaving</th>
<th>Neighbor</th>
</tr>
</thead>
<tbody>
<tr>
<td>(4, 3)</td>
<td>(100, 011)</td>
<td>(100101)</td>
<td>37</td>
</tr>
<tr>
<td>(4, 4)</td>
<td>(100, 100)</td>
<td>(110000)</td>
<td>48</td>
</tr>
<tr>
<td>(4, 5)</td>
<td>(100, 101)</td>
<td>(110001)</td>
<td>49</td>
</tr>
<tr>
<td>(5, 3)</td>
<td>(101, 011)</td>
<td>(100111)</td>
<td>39</td>
</tr>
<tr>
<td>(5, 5)</td>
<td>(101, 101)</td>
<td>(100011)</td>
<td>51</td>
</tr>
<tr>
<td>(6, 3)</td>
<td>(110, 011)</td>
<td>(101101)</td>
<td>45</td>
</tr>
<tr>
<td>(6, 4)</td>
<td>(100, 011)</td>
<td>(100101)</td>
<td>56</td>
</tr>
<tr>
<td>(6, 5)</td>
<td>(110, 101)</td>
<td>(111001)</td>
<td>57</td>
</tr>
</tbody>
</table>

In addition to the boxes in Table 3.2, the neighborhood includes the self-box, 50. Hence, there are 9 boxes in total.

The algorithm for data structuring describes four main steps. In step 1, we construct the root box containing all sources and targets and divide it to sub-spaces until it reaches level 2 ($l=2$). This process is similar to the non-adaptive FMM, and it will produce 64 boxes in 3D (16 boxes in 2D). In step 2, we need to count the total number of sources in the neighborhood $N_n$ of each box at level 2. In the third step, if we find that $N_n \leq q$, then we tag these boxes as $(n, 2)$ boxes and cease further partitioning. Otherwise, divide these boxes one more time and create level 3 or $(n, 3)$ boxes. At the last stage, we continue step 2 and 3 on newly created boxes in step 3 until all leaf nodes boxes satisfy the condition $N_n \leq q$. The process of subdivision may result in empty boxes, which do not contain any source particles or target particles. We implemented the algorithm in such a way to discard all empty boxes at each
level. Figure 3.18 shows partitioning of a root box. For simplicity, we choose 7 targets and 7 sources in two dimensional setup with $q=2$.

Figure 3.18: Hierarchical subdivision of a root box containing 7 targets (empty blue squares) and 7 sources (filled blue squares). The $q$ value is 2. The maximum level reached is 4.

3.2.3.3 The $D$-tree

The above four steps result in the set of target boxes, and parents of these target boxes build the $D$-tree, where each level $l$ of the $D$-tree contains a set of boxes $(n, l)$ whose $N_n \leq q$. 
Our FMM algorithm explains the formation of the $D$-tree with target boxes while maintaining the parent-child relationship traversing from the root box downwards. In the $D$-tree, the leaves or ends for each branch of the tree are known as $D$-leaf nodes, and the addition of the number of targets in leaf nodes is equal to the total number of targets as shown in Table 3.3. The $D$-tree is used in the downward pass of the adaptive FMM algorithm. The Fig. 3.19 depicts the corresponding $D$-tree for the subdivision of the box explained in Fig. 3.18.

3.2.3.4 The $C$-forest

This has many trees known as $C$-trees. During the process of constructing the $D$-tree, the interaction list of each $D$-tree node was identified. Therefore, the boxes in the interaction list of the each node in the $D$-tree containing source particles can be used to create the leaves
Table 3.3: Distribution of targets among the $D$-leaf nodes.

<table>
<thead>
<tr>
<th>Level</th>
<th>Leaf Node</th>
<th>Number of Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>38</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>105</td>
<td>1</td>
</tr>
</tbody>
</table>

or nodes of a $C$-tree. The $C$-tree is built by traversing through these nodes bottom-up and finding their parent-child relationships. The collection of such trees is the $C$-forest [14]. The $C$-forest is used in the upward pass of the adaptive FMM algorithm [14]. In general, the number of nodes of the $C$-forest at the finest level is remarkably smaller than the particle number, $N$. The Fig. 3.20 depicts the corresponding trees and the $C$-forest for the subdivision of the box explained in Fig. 3.18.

Table 3.4: Distribution of sources among the tree leaf nodes.

<table>
<thead>
<tr>
<th>Tree</th>
<th>Level</th>
<th>Leaf Node</th>
<th>Number of Sources</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>24</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>52</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>22</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>15</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.4 shows that the leaf nodes of trees in the $C$-forest contain sources, and their total is equal to the total number of sources. The number of trees in the $C$-forest varies with $q$ (Fig. 3.21). With small $q$ values, the particle domain needs fine resolution, which leads to create more octree nodes and $C$-forest trees. In 3D, each node has 27 neighbor boxes or
adjacent cubes including the self node. When $q=1$, only one of the 27 boxes contains only one source. As $q$ increases, the tree level goes down as a result of aggregating child nodes to form parent nodes while the number of trees remains unchanged.

### 3.2.4 Maximum Level Reached

For a fixed $q$ and $N$, a Gaussian distribution always reaches a higher level compared to a uniform distribution as shown in Fig. 3.22 due to the nature of the Gaussian distribution.
a. Gaussian Distribution  
b. Uniform Distribution

Figure 3.21: Number of trees in the C-forest due to two different particle distributions ($N=10^6$) as a function of $q$. When $q=1$, the number of trees is 56,622 and 61,700 in Gaussian and uniform distribution, respectively. Also, when $q=10^6$ the number of trees is 53 and 64 in Gaussian and uniform distribution, respectively.

If $q = N$, however, they both always reach level 2 or the minimum level. Similarly, if $q=1$, the data structures of both distributions always reach their maximum level. In this particular example, when $q = 1$, both distributions reach the same maximum level as a result of recursive subdivision.

The definition of adaptivity is algorithm dependent. The recursive space partitioning may result in empty boxes. The common practice is to skip these empty boxes found in each level and many algorithms consider this as adaptivity. In our algorithm, however, the recursive subdivision is performed only at specific locations of the domain with high particle density until it meets the prescribed condition. In other words, the subdivision process automatically adjusts to the density of the particle distribution. Since the level of the box of a given target is determined by both target and source sets, this algorithm earns the name **fully adaptive multi-level fast multipole method**.
Figure 3.22: Comparison of highest level reached in the two distributions \((N=10^6)\) for different \(q\). For any given \(q\), the Gaussian distribution reached a higher level compared to that of the normal distribution.

### 3.2.5 Parameter Optimization

Two main parameters in FMM have been identified, which can be used to manipulate the accuracy of performance: multipole expansion order \(p\) and the \(q\) value. The latter can be viewed as the minimum separation of boxes. Hence, the optimum \(q\) and \(p\) must be employed.

#### 3.2.5.1 Optimization of \(q\)

In a 3D setup, the position of each particle is defined by its coordinates \((x, y, z)\). We recursively and self-similarly subdivide the computational domain or the root box until it satisfies a precondition. This precondition is parameterized in different ways in different algorithms. In our algorithm, we use the \(q\) value as described in the previous section.
For a given set of target and source particles, the very first step is to specify the value of \( q \). The hierarchical decomposition of the domain is solely governed by the \( q \) value and it determines the highest level of the tree data structure, the runtime and the runtime memory usage. Also, \( q \) varies with the number of particles and the type of distribution.

A small \( q \) value increases the number of multipole-to-local translations which is the most costly operation in the downward pass of this algorithm. On the other hand, a large \( q \) value increases the direct (also known as pairwise or point-to-point) calculations, which is identified as another expensive calculation. In the extreme, where \( q = N \), the computational cost again goes back to \( O(N^2) \) as in the pairwise calculation. We can balance the runtime value taken by those costly calculations if we use the appropriate \( q \). Therefore, before building the data structure finding the optimum \( q \) value is indispensable, and it guarantees the minimum runtime (Fig. 3.23). For a given number of particles, the optimum \( q \) varies with the type of distribution. Gaussian distributions typically have a well-defined sharp minimum point at the optimum \( q \) (Fig.3.23.a). On the contrary, uniform distributions a range of \( q \) values can be accepted as the optimum \( q \) (Fig.3.23.b).

Table 3.5: Optimum \( q \) of uniformly distributed data sets of different sizes.

<table>
<thead>
<tr>
<th>( N )</th>
<th>optimum q</th>
<th>( N )</th>
<th>optimum q</th>
<th>( N )</th>
<th>optimum q</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>240</td>
<td>2500</td>
<td>25 \times 10^4</td>
<td>11000</td>
<td>220 \times 10^4</td>
</tr>
<tr>
<td>1 \times 10^3</td>
<td>450</td>
<td>30 \times 10^3</td>
<td>1700</td>
<td>250 \times 10^3</td>
<td>2100</td>
</tr>
<tr>
<td>2 \times 10^3</td>
<td>900</td>
<td>40 \times 10^3</td>
<td>2200</td>
<td>275 \times 10^3</td>
<td>2000</td>
</tr>
<tr>
<td>5 \times 10^3</td>
<td>2300</td>
<td>50 \times 10^3</td>
<td>2800</td>
<td>3 \times 10^5</td>
<td>2100</td>
</tr>
<tr>
<td>10 \times 10^3</td>
<td>4400</td>
<td>60 \times 10^3</td>
<td>3300</td>
<td>5 \times 10^5</td>
<td>3500</td>
</tr>
<tr>
<td>20 \times 10^3</td>
<td>8700</td>
<td>70 \times 10^3</td>
<td>3900</td>
<td>8 \times 10^5</td>
<td>5500</td>
</tr>
<tr>
<td>21 \times 10^3</td>
<td>9000</td>
<td>100 \times 10^3</td>
<td>5400</td>
<td>1 \times 10^6</td>
<td>7000</td>
</tr>
<tr>
<td>24 \times 10^3</td>
<td>10500</td>
<td>200 \times 10^3</td>
<td>11000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
a. Gaussian Distribution  
b. Uniform Distribution

Figure 3.23: Variation of the runtime with different $q$ values of two different distributions ($N=100k$) at FMM order=2. The optimum $q$ value gives the minimum runtime for data structuring.

As discussed before, in our algorithm, we use the parameter $q$, and it is the maximum number of sources allowed in the neighborhood of a particular target. The largest level reached by the space partitioning solely depends on this parameter.

To harness the full advantage of the FMM, we have to optimize these parameters. The value of $q$ becomes very sensitive for large $N$. On the other hand, for a given $N$, the $q$ value changes with the type of particle distribution. Therefore, $q$ must be carefully chosen before implementing the FMM. The rule of thumb is to use $0.1N$ as the $q$. The variation of optimum $q$ for Gaussian and uniform distribution at the FMM order 8 is shown in Fig. 3.24. The optimum values for different sizes of data sets derived from a uniform distribution are shown in Table 3.5.
3.2.5.2 FMM order, $p$

The other parameter is the number of multipole expansion terms needed while maintaining the accuracy and efficiency. This is determined by the FMM order.

The lowest possible FMM order is 2, and it takes the minimum runtime. To ensure higher accuracy, however, the FMM order $p$ must be increased. We measured the runtime for a Gaussian distribution of 300k particles with different orders and it shows that the runtime increases approximately as the third power of $p$ (Fig. 3.25).

Furthermore, using a Gaussian distribution of 1000 particles with unit charge, we calculated the potential at each particle and compared them with the potential calculated in the point-to-point method. The absolute relative error of each particle’s potential was computed by comparing the two potential measurements. The error histograms for four different orders are shown in Fig. 3.26 and it suggests that the FMM order 12 suffices for our simulation.
since the absolute error is less than $10^{-7}$. As expected, it also shows a non-linear trend with runtime increasing rapidly as the order increases and it hinders efficiency.

![Graph](image)

Figure 3.25: Non-linear relationship between the runtime and the FMM order, $p$, for Gaussian distribution of $N=300k$.

To investigate the performance of the algorithm we measured the runtime and runtime memory usage for different data sets [37].

3.2.5.3 Scaling with the error, $\varepsilon$

We calculated the average potential of 10,000 particles using the direct method and the FMM with different FMM orders at the optimum $q$. Fig. 3.27 shows that the relative
Figure 3.26: Relative error of the potential calculated with different FMM orders for 1000 charged particles in a Gaussian distribution.

<table>
<thead>
<tr>
<th>FMM order</th>
<th>Gaussian</th>
<th>Uniform</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>$8.00 \times 10^{-4}$</td>
<td>$3.88 \times 10^{-4}$</td>
</tr>
<tr>
<td>3</td>
<td>$2.99 \times 10^{-4}$</td>
<td>$1.72 \times 10^{-4}$</td>
</tr>
<tr>
<td>4</td>
<td>$6.77 \times 10^{-5}$</td>
<td>$1.95 \times 10^{-5}$</td>
</tr>
<tr>
<td>5</td>
<td>$1.52 \times 10^{-5}$</td>
<td>$3.80 \times 10^{-6}$</td>
</tr>
<tr>
<td>6</td>
<td>$3.11 \times 10^{-6}$</td>
<td>$9.75 \times 10^{-7}$</td>
</tr>
<tr>
<td>7</td>
<td>$7.47 \times 10^{-7}$</td>
<td>$3.90 \times 10^{-7}$</td>
</tr>
<tr>
<td>8</td>
<td>$2197 \times 10^{-7}$</td>
<td>$1.22 \times 10^{-7}$</td>
</tr>
<tr>
<td>9</td>
<td>$8.95 \times 10^{-8}$</td>
<td>$6.15 \times 10^{-8}$</td>
</tr>
</tbody>
</table>

error between two average potential values linearly diminishes as the FMM order $p$ increases (Table 3.6). The relative error is defined by the formula given in Eq. 3.31.

$$E_{rel} = \varepsilon = \sqrt{\frac{\sum_{i}^{N} (\phi_{if} - \phi_{id})^2}{\sum_{i}^{N} \phi_{id}^2}},$$ (3.31)
where $\phi_{id}$ and $\phi_{if}$ denote the average potential calculated for each particle $i$ in the direct method and in the FMM, respectively. In response to the challenge of improving the accuracy of the results effectively, we must increase the FMM order. However, the price to pay for higher accuracy by increasing the FMM order is the longer run time as described in section 3.2.6.

### 3.2.6 Runtime

The simulations performed using our novel algorithm show that the runtime has a linear relationship with the total number of particles from any type of distribution (Fig. 3.28). Therefore, for very large $N$, which is the case in particle beams, the FMM becomes an indispensable method for Coulomb potential evaluation.
Figure 3.28: Data structuring runtime and full FMM (data structuring and Coulomb potential calculation) runtime linearly scale with the number of particles, $N$, in both uniform and Gaussian distributions. In each case, the $q$ value is taken as $0.1N$.

Figure 3.29: Comparison of time taken by the direct method or pairwise calculation (red solid line) with those of full FMM at different FMM orders. When $N$ is small, in order to calculate the potentials with high accuracy, the pairwise calculation is the best method.
Also, the runtime to calculate the potential of charged particle distributions of different sizes were measured with different FMM orders and compared with the time taken for the point-to-point evaluation method. This is depicted in Fig. 3.29. It is apparent that, for small $N$, the FMM with higher orders take longer time than the point-to-point (or pairwise) method due to the FMM has a quite a big overhead. Therefore, for small particle ensembles where $N < 10^4$ the point-to-point method is the appropriate method for potential evaluation. For large $N$, however, the FMM becomes the most efficient and viable method since the point-to-point method is practically unfeasible. Due to the reduced rounding errors, the FMM can be more accurate than the direct method for large number of particles.

Table 3.7: Runtime measured varying the FMM order with 1DM2L and 3DM2L for 800k particles. For higher orders ($> 5$) 3DM2L is faster.

<table>
<thead>
<tr>
<th>FMM order</th>
<th>3DM2L (min)</th>
<th>1DM2L(min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>8.72</td>
<td>7.77</td>
</tr>
<tr>
<td>3</td>
<td>10.59</td>
<td>9.62</td>
</tr>
<tr>
<td>4</td>
<td>14.15</td>
<td>13.31</td>
</tr>
<tr>
<td>5</td>
<td>20.58</td>
<td>20.23</td>
</tr>
<tr>
<td>6</td>
<td>28.62</td>
<td>31.34</td>
</tr>
<tr>
<td>7</td>
<td>39.29</td>
<td>45.77</td>
</tr>
<tr>
<td>8</td>
<td>32.29</td>
<td>66.44</td>
</tr>
<tr>
<td>9</td>
<td>67.18</td>
<td>96.18</td>
</tr>
</tbody>
</table>

### 3.2.7 Runtime Memory Footprint

The runtime memory usage linearly scales with the number of particles for any $q$ value \[[37]\] as shown in Fig 3.31.
Figure 3.30: Variation of the runtime with the FMM order for 1DM2L and 3DM2L in Gaussian distribution of $N=800k$.

Figure 3.31: Runtime memory (measured using Valgrind 3.7.0) scales with the number of particles, $N$. 
Table 3.8: Variation of peak runtime-memory with particle number, $N$.

<table>
<thead>
<tr>
<th>$N$</th>
<th>Memory (MB)</th>
<th>$N$</th>
<th>Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 \times 10^4$</td>
<td>38.27</td>
<td>$1 \times 10^6$</td>
<td>207.10</td>
</tr>
<tr>
<td>$4 \times 10^4$</td>
<td>63.00</td>
<td>$2 \times 10^6$</td>
<td>354.30</td>
</tr>
<tr>
<td>$6 \times 10^4$</td>
<td>64.96</td>
<td>$3 \times 10^6$</td>
<td>539.10</td>
</tr>
<tr>
<td>$8 \times 10^4$</td>
<td>69.40</td>
<td>$4 \times 10^6$</td>
<td>642.40</td>
</tr>
<tr>
<td>$1 \times 10^5$</td>
<td>71.98</td>
<td>$5 \times 10^6$</td>
<td>888.40</td>
</tr>
<tr>
<td>$1 \times 10^5$</td>
<td>106.20</td>
<td>$6 \times 10^6$</td>
<td>999.00</td>
</tr>
<tr>
<td>$1 \times 10^5$</td>
<td>129.80</td>
<td>$8 \times 10^6$</td>
<td>1230.00</td>
</tr>
</tbody>
</table>

The runtime memory usage measured using Valgrind 3.7.0 for the particle limit $q = 0.01N$ is shown in Table 3.8.

Therefore, we were able to achieve an impressive efficiency with the new FMM algorithm. Even though the Gaussian distribution is used in the most of the simulations, the other types of distributions can be treated in a similar way.

### 3.2.8 Potential Measured with the New code

To investigate the performance of the FMM code, we measured potential at some target points distributed on a 2D grid using three types of source distributions. We choose the source distributions such that we can predict the shape of the potential plot.

**source points: 3D Single Gaussian, target points: 2D uniform grid**

In Fig. 3.32, the source points are distributed as a single round 3D Gaussian and the targets are uniformly distributed in a 2D mesh. The potential at the center of the grid is maximum with a well-defined single peak as we anticipated as shown in Fig. 3.33.
Figure 3.32: Target-source configuration. Sources are derived from a single Gaussian distribution.
source points: Uniform cube, target points: 2D uniform grid

In Fig. 3.34, the source points are uniformly distributed in a cube in the range $[-1, 1]$ in all three dimensions $x, y, z$. The targets are uniformly distributed in a 2D mesh. The potential is maximum at the center of the grid with a flat-top peak as shown in Fig. 3.35.
Figure 3.34: Target-source configuration. Sources are derived from a uniform distribution.
source points: Nine-Gaussian, target points: 2D uniform grid

In Fig. 3.36, the source points are distributed derived from nine Gaussian distributions. The targets are uniformly distributed in a 2D mesh. The potential is nine-peaked corresponding to source particle locations as shown in Fig. 3.37.
Figure 3.36: Target-source configuration. Sources are derived from a nine-Gaussian distribution.
source points: Gaussian distribution, target points: uniform distribution

Figure 3.38 shows the 3D contour plot of the potential measured using 10,000 source points (Gaussian distribution) and 35,937 target points. The target points are uniformly distributed in the range \([-3, 5]\) in all three dimensions. Figure 3.39 shows the 2D contour plots at different \(z\) values. The accuracy and efficiency of the new FMM code is very impressive. When we deal with a large number of particles, we can calculate the interaction according to our need. In order to gain high accuracy we can increase the FMM order. On the other hand, by decreasing the FMM order we can reduce the runtime.

Results produced in all the tests using the new FMM perfectly match with the results of the point-to-point method. By adjusting the FMM order we can change the level of accuracy of the FMM results. In most cases, the FMM order 5 is a moderate value to maintain the accuracy and the efficiency. It is practically impossible to calculate the potential/fields in any other methods in a reasonable amount of time at this accuracy. Comparison of the FMM runtime with the direct runtime is shown in Table 3.9.
Figure 3.38: 3D Contour plot. The FMM order is 2.

Table 3.9: Comparison of FMM runtime with direct runtime for different particle numbers, $N$, and different FMM orders.

<table>
<thead>
<tr>
<th>$N$</th>
<th>Method</th>
<th>FMM order</th>
<th>Runtime (min)</th>
<th>Error</th>
<th>Speed increased factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>$5 \times 10^4$</td>
<td>Direct</td>
<td>-</td>
<td>47</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$5 \times 10^4$</td>
<td>FMM</td>
<td>2</td>
<td>0.44</td>
<td>$1 \times 10^{-2}$</td>
<td>106</td>
</tr>
<tr>
<td>$5 \times 10^4$</td>
<td>FMM</td>
<td>5</td>
<td>1.3</td>
<td>$1 \times 10^{-4}$</td>
<td>36</td>
</tr>
<tr>
<td>$1 \times 10^6$</td>
<td>Direct</td>
<td>-</td>
<td>23,040 (predicted)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$1 \times 10^6$</td>
<td>FMM</td>
<td>2</td>
<td>9</td>
<td>$1 \times 10^{-2}$</td>
<td>12000</td>
</tr>
<tr>
<td>$1 \times 10^6$</td>
<td>FMM</td>
<td>9</td>
<td>20</td>
<td>$1 \times 10^{-8}$</td>
<td>1200</td>
</tr>
</tbody>
</table>
a. $z=1$.  

b. $z=4$. 

Figure 3.39: 2D Contour plots of the potential at different $z$ values.
CHAPTER 4
PARTICLES’ HIGH-ORDER ADAPTIVE DYNAMICS (PHAD)

4.1 Transition from FMM to Particles High-order Adaptive Dynamics (PHAD)

We can treat an idealized ensemble of particles, which contains stationary particles, with the FMM code we developed. However, to study a realistic particle beam, which can be considered representative of a practical situation, we need to modify our simulations to incorporate the temporal dependence. The time evolution of the particle beam can be studied by numerically solving the Newton’s equations of motion. Therefore, we developed another new code — Particles’ High-order Adaptive Dynamics (PHAD).

The previously explained FMM will be hereafter referred to as the *stand-alone FMM*. The PHAD code includes the stand-alone FMM. In addition, in PHAD, we have introduced a time stepping integrator called Picard iteration-based integrator [39]. Accordingly, we can trace the particle trajectories as time advances. Also, we employ an operator splitting method called Strang splitting [40] in PHAD to enhance the efficiency. Another advantage of Strang splitting is its ability to maintain symplecticity [41].

4.2 N-body Problem

The N-body problem is originated in celestial mechanics, which has a very long and distinguished history. Investigation of a system of identical objects, interacting via a force
proportional to the inverse square of the distance between objects comes under the typical \( N \)-body problem. In general, it is the study of a dynamical system of many particles under the influence of mutual interactions. The motion of \( N \) objects under the influence of mutual attraction can be described by the equations of motion as

\[
m_i \ddot{\vec{r}}_i = \vec{F}_i = \frac{1}{2} G m_i \left[ \sum_{j=1, j \neq i}^{N} \frac{m_j (\vec{r}_i - \vec{r}_j)}{d_{ij}^3} \right],
\]

where \( \vec{r}_i \) and \( \vec{r}_j \) are position vectors to particle \( i \) and \( j \), respectively while \( d_{ij} \) is the distance between the particles. Also, \( m_i \) and \( G \) denote the mass of the particle \( i \) and the gravitational constant. This can be analytically solved exactly for 2 particles. For a system of \( N \) (\( N > 2 \)) particles, there are \( 6N \) differential equations and numerical methods must be used to solve them.

In the case of a charged particle beam, due to the long range of coulomb interaction, every charged particle interacts simultaneously with many other particles in the system. This occurs whether these particles are in the near or far region with respect to the reference particle. Based on this fact, one can identify the effect of far region and the near region. The effect of far region is a smoothly varying function of the position of the particle and time while the near region effect is a rapidly decaying function of the distance.

### 4.2.1 Electromagnetic N-body Problem

An ensemble of moving charged particles forms a beam in accelerators. Beams propagate in the accelerator in the longitudinal direction as a continuous or a bunched beam. Typically, a single bunch is composed of a large number of particles in the order of \( \sim 10^{13} \). Hence, the particles in a beam can frequently be considered a \( N \)-body problem. For instance, in
order to study the space charge effect, intra-beam scattering and electron cooling of heavy ions can be viewed as $N$-body problems. Since the charged particles experience Coulomb interactions, we reformulate the equations of motion given in Eq. 4.1 as,

$$
\vec{F}_i = \frac{q_i}{4\pi\varepsilon_0} \sum_{j=1 \atop j \neq i}^{N} \frac{q_j (\vec{r}_i - \vec{r}_j)}{||\vec{r}_i - \vec{r}_j||^2}, \tag{4.2}
$$

where $\vec{F}_i$ is the force exerted on the $i^{th}$ particle by the $j^{th}$ particle, and $q_i$ and $q_j$ are the charges of the two particles, respectively.

This force can further be decomposed into two vectors as follows.

$$
\vec{F}_i = \vec{F}_{i\text{self}} + \vec{F}_{i\text{external}}. \tag{4.3}
$$

External forces usually independent of the relative positions of particles and the number of particles while the self forces depend on the position or momentum of particles. Accordingly, Eq. 4.2 takes the form,

$$
\vec{F}_{i\text{self}} = \frac{q_i}{4\pi\varepsilon_0} \sum_{j=1 \atop j \neq i}^{N} \frac{q_j (\vec{r}_i - \vec{r}_j)}{||\vec{r}_i - \vec{r}_j||^\frac{3}{2}}. \tag{4.4}
$$

If the charge $q_i$ is moving with velocity $\vec{v}_i$ under the influence of the applied electric field $\vec{E}$ and magnetic field $\vec{B}$, the force experienced by the particle $i$ becomes

$$
\vec{F}_{i\text{external}} = q_i \vec{E} + q_i \vec{v}_i \times \vec{B}. \tag{4.5}
$$

### 4.2.2 Fundamental Equations in Beam Dynamics

We need to identify the set of differential equations in our problem due to the change in position and the momentum of each particle. In the Cartesian coordinate system, the
position of the \(i^{th}\) particle can be given by \((x_i, y_i, z_i)\). Similarly, the momentum can be given as \((p_{x_i}, p_{y_i}, p_{z_i})\). The velocity of the particle \(i\) can be expressed as

\[
\vec{v}_i = \frac{d\vec{r}_i}{dt}.
\]

If the Lorentz factor is \(\gamma\), and the rest mass of the particle is \(m_i\)

\[
\gamma = \frac{1}{\sqrt{1 - \left(\frac{\|\vec{v}_i\|}{c}\right)^2}} \tag{4.6}
\]

and the relativistic momentum is

\[
\vec{p}_i = \gamma m_i \vec{v}_i = \frac{m_i \vec{v}_i}{\sqrt{1 - \left(\frac{\|\vec{v}_i\|}{c}\right)^2}} = \frac{m_i \, c \, \vec{v}_i}{\sqrt{c^2 - \|\vec{v}_i\|^2}}, \tag{4.7}
\]

where \(c\) is the speed of light. This follows that the velocity is

\[
\vec{v}_i = \frac{\sqrt{c^2 - \|\vec{v}_i\|^2}}{m_i \, c} \vec{p}_i. \tag{4.8}
\]

We use momentum instead of velocities in the differential equations (ODEs) due to numerical stability reasons. Therefore, after much simplification, the velocity in terms of momentum is

\[
\vec{v}_i = \frac{d\vec{r}_i}{dt} = \frac{\vec{p}_i}{m_i \sqrt{1 + \left(\frac{\|\vec{p}_i\|}{m_i c}\right)^2}}. \tag{4.9}
\]

The force acting on the particle \(i\) is

\[
\vec{F}_i = \frac{q_i}{4\pi\varepsilon_0} \sum_{j=1}^{N} q_j \frac{(\vec{r}_i - \vec{r}_j)}{\|\vec{r}_i - \vec{r}_j\|^2} + q_i \vec{E} + q_i \vec{v}_i \times \vec{B}
\]
or as a derivative
\[
\frac{d\vec{r}_i}{dt} = q_i \left( \frac{1}{4\pi\epsilon_0} \sum_{j=1, j \neq i}^{N} \frac{q_j (\vec{r}_i - \vec{r}_j)}{|\vec{r}_i - \vec{r}_j|^2} + \vec{E} + \vec{v}_i \times \vec{B} \right).
\]

(4.10)

### 4.2.3 Lorentz Transformation

There are two frames of reference, the laboratory frame and the beam rest frame. In the laboratory frame (or lab frame), the particles move at velocity \(v\hat{z}\) and in the beam rest frame they are stationary [42]. In general, \(\vec{E}\) and \(\vec{B}\) have two components, the external field and the space charge field. It has become customary to express the physical quantities in the lab frame. Hence, the external electric field \(\vec{E}\) and the magnetic field \(\vec{B}\) can be given in the lab frame. However, in the beam rest frame, the \(\vec{B}\) field is zero since the particles are stationary in that frame. Therefore, in the beam frame it becomes an electrostatic problem, and the space charge field can be conveniently calculated in the beam frame.

If the particles move at a relativistic velocity \(v\hat{z} = \beta c\hat{z}\), the space charge field can be calculated in the beam rest frame and transformed to the lab frame by means of Lorentz transformation. Finally, in the lab frame, we have to solve an electromagnetic problem, which includes the forces due to external electric and magnetic fields as well as the space charge effect. Therefore, we solve the equations of motion in the lab frame. For all practical purposes, we follow this procedure since it allows us to know where the beam is at every moment.

For simplicity, we can consider the two coordinate systems with the \(z\) axes aligned. The primed notation is used to designate the beam rest frame \(S'\) and the unprimed notation for the laboratory frame \(S\). The origins \(O\) and \(O'\) denote their respective origins as shown in Fig. 4.1.
Figure 4.1: The laboratory frame (unprimed) and the beam frame (primed) with z axes aligned. The particle beam moves at the velocity $v \hat{z} = \beta c \hat{z}$.

At $t = t' = 0$,

$$
x = x',
y = y'.
$$

(4.11)

The space charge field in the rest frame is

$$
E'(x', y', z') = \begin{cases} 
x' - x_0' \\
\frac{(x' - x_0')^2 + (y' - y_0')^2 + (z' - z_0')^2}^{3/2} y' - y_0' \\
\frac{(x' - x_0')^2 + (y' - y_0')^2 + (z' - z_0')^2}^{3/2} z' - z_0' \end{cases}
$$

(4.12)

In the lab frame, the electric field is

$$
E(x, y, z) = \begin{cases} 
\gamma E'_x(x', y', z') = \gamma E'_x(x, y, z), & \text{since } x = x_0', y = y_0', \gamma z = z_0' \\
\gamma E'_y(x, y, z') \\
E'_z(x, y, z')
\end{cases}
$$

(4.13)
and the magnetic field is

\[ B(x, y, z) = \begin{cases} 
- \frac{\gamma}{c} E_y' \\
\frac{\gamma}{c} E_x' \\
0.
\end{cases} \tag{4.14} \]

If the potential at \((x', y', z')\) (or \(x, y, z')\) in the beam rest frame is \(\phi(x', y', z')\) (or \(\phi(x, y, z')\)) then

\[ E_z'(x', y', z') = - \nabla \phi = - \frac{\partial}{\partial z'} \phi(x, y, z') \]

\[ = - \frac{\partial}{\partial z'} \frac{1}{\sqrt{(x' - x_0')^2 + (y' - y_0')^2 + (z' - z_0')^2}} \]

\[ = \frac{\gamma(z - z_0)}{[(x - x_0)^2 + (y - y_0)^2 + \gamma^2(z - z_0)^2]^{3/2}}, \quad \text{since } z' = \gamma z. \tag{4.15} \]

\[ E_z(x, y, z) = E_z'(x, y, z), \quad \text{since } x_0 = x'_0, \ y_0 = y'_0 \tag{4.16} \]

\[ E_x'(x', y', z') = - \nabla \phi = - \frac{\partial}{\partial x'} \phi(x, y, z') \]

\[ = - \frac{\partial}{\partial x'} \frac{1}{\sqrt{(x' - x_0')^2 + (y' - y_0')^2 + (z' - z_0')^2}} \]

\[ = \frac{x - x_0}{[(x - x_0)^2 + (y - y_0)^2 + \gamma^2(z - z_0)^2]^{3/2}}, \tag{4.17} \]

Similarly,

\[ E_y'(x', y', z') = \frac{y - y_0}{[(x - x_0)^2 + (y - y_0)^2 + \gamma^2(z - z_0)^2]^{3/2}}. \tag{4.18} \]
The final formula in $S(O)$, can be obtained by Lorentz transformation of $(x'_0, y'_0, z'_0)$ in $S'(O')$ to the corresponding values of $(x_0, y_0, z_0)$ in $S(O)$.

The relationship between the two frames is

$$
\begin{align*}
&x'_0 = x_0 \\
y'_0 = y_0 \\
z'_0 = \gamma z_0. 
\end{align*}
$$

(4.19)

The transformed fields in the lab frame are,

$$
\begin{align*}
E_x(x, y, z) &= \frac{\gamma (x - x_0)}{\left[(x - x_0)^2 + (y - y_0)^2 + \gamma^2 (z - z_0)^2\right]^{3/2}} \\
E_y(x, y, z) &= \frac{\gamma (y - y_0)}{\left[(x - x_0)^2 + (y - y_0)^2 + \gamma^2 (z - z_0)^2\right]^{3/2}} \\
E_z(x, y, z) &= \frac{\gamma (z - z_0)}{\left[(x - x_0)^2 + (y - y_0)^2 + \gamma^2 (z - z_0)^2\right]^{3/2}}.
\end{align*}
$$

(4.20)

The electromagnetic forces can be expressed as

$$
\vec{F} = q(\vec{E} + \vec{v} \times \vec{B})
$$

$$
= q(\vec{E} + \vec{v} \times \frac{(\vec{v} \times \vec{E})}{c^2})
$$

$$
= q[\vec{E} \left(1 - \frac{v^2}{c^2}\right) + \vec{v} \cdot \frac{v^2}{c^2} \hat{z}].
$$

(4.21)

The components of the force can be explicitly given by

$$
\begin{align*}
F_x &= \frac{q}{\gamma^2} E_x = \frac{q}{\gamma} E'_x \\
F_y &= \frac{q}{\gamma^2} E_y = \frac{q}{\gamma} E'_y \\
F_z &= qE_z = qE'_z.
\end{align*}
$$

(4.22)
Hence,

\[
\begin{align*}
\frac{d\phi_x}{dt} &= \frac{q}{\gamma^2} E_x = \frac{q}{\gamma} \frac{x - x_0}{\left( (x - x_0)^2 + (y - y_0)^2 + \gamma^2 (z - z_0)^2 \right)^{3/2}} \\
\frac{d\phi_y}{dt} &= \frac{q}{\gamma^2} E_y = \frac{q}{\gamma} \frac{y - y_0}{\left( (x - x_0)^2 + (y - y_0)^2 + \gamma^2 (z - z_0)^2 \right)^{3/2}} \\
\frac{d\phi_z}{dt} &= qE_z = q\gamma \frac{z - z_0}{\left( (x - x_0)^2 + (y - y_0)^2 + \gamma^2 (z - z_0)^2 \right)^{3/2}}.
\end{align*}
\] (4.23)

Therefore, Eq. 4.9 and Eq. 4.10 need to be solved, and for \( N \) particles there are \( 6N \) such differential equations in total. Solving this set of equations numerically is tedious unless we do not use proper techniques. We use a Picard iteration based integrator with variable order and dense output to solve them. Also, we have used a fixed time step size throughout the entire time interval.

### 4.3 6N Differential Equations

The velocity of the \( i^{th} \) particle at the position \( \vec{r}_i \) is given by

\[
\frac{d\vec{r}_i}{dt} = \frac{\vec{p}_i}{m \sqrt{1 + \left( \frac{\|\vec{p}_i\|}{mc} \right)^2}},
\] (4.24)

where \( \vec{p}_i \) is the momentum.

The total force including the Coulomb force and the forces exerted by the external electric and magnetic fields can be given by
\[ \frac{d\vec{p}_i}{dt} = q_i \left\{ \begin{array}{l} \frac{1}{4\pi\epsilon_0} \sum_{j=1}^{N} q_j \frac{(x_i - x_j)i + (y_i - y_j)j + (z_i - z_j)k}{\left[ (x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2 \right]^{3/2}} \\ j \neq i \end{array} \right\} \\
+ \vec{E}_{\text{external}} \]

\[ + \frac{p_{x_i}i + p_{y_i}j + p_{z_i}k}{m_i} \left[ \left( \frac{\sqrt{p_{x_i}^2 + p_{y_i}^2 + p_{z_i}^2}}{m_i c} \right)^2 + 1 \right]^{1/2} \times \vec{B} \]  

The input parameters of the \( i^{th} \) particle can be given by

\[ Y_i = \begin{pmatrix} x_i \\ y_i \\ z_i \\ p_{x_i} \\ p_{y_i} \\ p_{z_i} \end{pmatrix} \]
for particles 1 through \( N \) in order to write the equations in a more compact way. We can define the function \( F_i \) by \( F_i(Y_i(t), t) = Y_i'(t) \). Using Eq. 4.24 and Eq. 4.25 the set of equations for particle \( i \) is

\[
\begin{align*}
Y_i' &= \left[ \begin{array}{c}
\frac{p_{x_i}}{m_i \sqrt{1 + \left( \frac{\sqrt{p_{x_i}^2 + p_{y_i}^2 + p_{z_i}^2}}{m_i c} \right)^2}} \\
\frac{p_{y_i}}{m_i \sqrt{1 + \left( \frac{\sqrt{p_{x_i}^2 + p_{y_i}^2 + p_{z_i}^2}}{m_i c} \right)^2}} \\
\frac{p_{z_i}}{m_i \sqrt{1 + \left( \frac{\sqrt{p_{x_i}^2 + p_{y_i}^2 + p_{z_i}^2}}{m_i c} \right)^2}}
\end{array} \right] = F_i(Y_i, t),
\end{align*}
\]

where

\[
F_i,4(Y_i, t) + q_i E_{x_i} \\
F_i,5(Y_i, t) + q_i E_{y_i} \\
F_i,6(Y_i, t) + q_i E_{z_i}
\]
\begin{align*}
F_{i,4}(Y_i, t) &= q_i \left[ \frac{1}{4\pi\epsilon_0} \sum_{j=1 \atop j \neq i}^N \frac{q_j(x_i - x_j)}{[(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2]^{3/2}} + B_{z_i}v_{y_i} - B_{y_i}v_{z_i} \right] \\
&= q_i \left[ \frac{1}{4\pi\epsilon_0} \sum_{j=1 \atop j \neq i}^N \frac{q_j(x_i - x_j)}{[(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2]^{3/2}} \\
&\quad + B_{z_i} \frac{p_{y_i}}{m_i} \sqrt{1 + \left( \frac{\sqrt{p_{x_i}^2 + p_{y_i}^2 + p_{z_i}^2}}{m_i c} \right)^2} - B_{y_i} \frac{p_{x_i}}{m_i} \sqrt{1 + \left( \frac{\sqrt{p_{x_i}^2 + p_{y_i}^2 + p_{z_i}^2}}{m_i c} \right)^2} \right], \\
F_{i,5}(Y_i, t) &= q_i \left[ \frac{1}{4\pi\epsilon_0} \sum_{j=1 \atop j \neq i}^N \frac{q_j(y_i - y_j)}{[(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2]^{3/2}} - (B_{z_i}v_{x_i} - B_{x_i}v_{z_i}) \right] \\
&= q_i \left[ \frac{1}{4\pi\epsilon_0} \sum_{j=1 \atop j \neq i}^N \frac{q_j(y_i - y_j)}{[(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2]^{3/2}} \\
&\quad - B_{z_i} \frac{p_{x_i}}{m_i} \sqrt{1 + \left( \frac{\sqrt{p_{x_i}^2 + p_{y_i}^2 + p_{z_i}^2}}{m_i c} \right)^2} + B_{x_i} \frac{p_{z_i}}{m_i} \sqrt{1 + \left( \frac{\sqrt{p_{x_i}^2 + p_{y_i}^2 + p_{z_i}^2}}{m_i c} \right)^2} \right], \\
&\quad \text{and}
\end{align*}
\[ F_{i,6}(Y_i, t) = q_i \left[ \frac{1}{4\pi \varepsilon_0} \sum_{j=1 \atop j \neq i}^N q_j \left( z_i - z_j \right) \left( \frac{1}{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2} \right)^{3/2} + B_{y_i} v_{x_i} - B_{x_i} v_{y_i} \right] \]

\[ = q_i \left[ \frac{1}{4\pi \varepsilon_0} \sum_{j=1 \atop j \neq i}^N q_j \left( z_i - z_j \right) \left( \frac{1}{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2} \right)^{3/2} \right. \]

\[ + B_{y_i} \left( \frac{p_{x_i}}{m_i \sqrt{1 + \left( \frac{\sqrt{p^2_{x_i} + p^2_{y_i} + p^2_{z_i}}}{m_i c} \right)^2}} \right)^2 - B_{x_i} \left( \frac{p_{y_i}}{m_i \sqrt{1 + \left( \frac{\sqrt{p^2_{x_i} + p^2_{y_i} + p^2_{z_i}}}{m_i c} \right)^2}} \right)^2 \]

The force including the external electric and magnetic field contribution becomes

\[
\frac{dp_i}{dt} = q_i \left\{ \frac{1}{4\pi \varepsilon_0} \sum_{j=1 \atop j \neq i}^N q_j \left( \frac{(x_i - x_j)\vec{i} + (y_i - y_j)\vec{j} + (z_i - z_j)\vec{k}}{[(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2]^{3/2}} \right) \right. \]

\[ \left. + \vec{E}_{\text{external}} \right\} \]

\[
+ \frac{p_{x_i} \vec{i} + p_{y_i} \vec{j} + p_{z_i} \vec{k}}{m_i \sqrt{1 + \left( \frac{\sqrt{p^2_{x_i} + p^2_{y_i} + p^2_{z_i}}}{m_i c} \right)^2}} \times \vec{B} \]

\[ (4.28) \]
The components of the magnetic field can be expanded out component-wise to get the following relationship.

\[ v_i \times \vec{B}(x_i, y_i, z_i) = (B_{z_i}v_{y_i} - B_{y_i}v_{z_i})\hat{i} - (B_{z_i}v_{x_i} - B_{x_i}v_{z_i})\hat{j} + (B_{y_i}v_{x_i} - B_{x_i}v_{y_i})\hat{k}. \]

### 4.3.1 Scaling of Equations

Since we have to deal with extremely small values or large values, it is desirable to manipulate those numbers by means of known physical constants. For instance, the mass and the charge of an electron are very small quantities. Similarly, the time step size required for our simulation is in the order of \( \approx 10^{-12} \text{s} \) since the particles move at a relativistic speed. To sidestep this issue, we scale the time step size by multiplying the proper time step size given in seconds by the speed of light. Therefore, the scaled time step size can be given in meters. Similarly, the momentum, the electric field, the charge, and the mass are scaled as shown in the Table 4.1.

Table 4.1: Scaled quantities are denoted with the hat notation. \( m \) and \( q \) represent the mass and the charge of a proton, respectively.

<table>
<thead>
<tr>
<th>unscaled term</th>
<th>scaled term</th>
</tr>
</thead>
<tbody>
<tr>
<td>time ( t )</td>
<td>( t = tc )</td>
</tr>
<tr>
<td>charge ( q_i )</td>
<td>( \frac{q_i}{q} )</td>
</tr>
<tr>
<td>mass ( m_i )</td>
<td>( \frac{m_i}{m} )</td>
</tr>
<tr>
<td>momentum ( p_{x_i} )</td>
<td>( \frac{p_{x_i}}{mc} )</td>
</tr>
<tr>
<td>momentum ( p_{y_i} )</td>
<td>( \frac{p_{y_i}}{mc} )</td>
</tr>
<tr>
<td>momentum ( p_{z_i} )</td>
<td>( \frac{p_{z_i}}{mc} )</td>
</tr>
</tbody>
</table>
4.3.1.1 Scaling of Equations with Time

As shown in Table 4.1, we use \( \hat{t} = ct \). Therefore, the derivative with respect to proper time must be declared in terms of scaled time. Therefore,

\[
\frac{dY_i}{d\hat{t}} = \frac{dY_i}{dt} \frac{dt}{d\hat{t}} = \frac{1}{c} \frac{dY_i}{dt}.
\] (4.29)

Using the relationship in Eq. 4.29, the Eq. 4.27 with respect to \( \hat{t} \) is

\[
\frac{dY_i}{d\hat{t}} = \frac{1}{c} \left[ \frac{p_{z_i}}{m_i} \sqrt{1 + \left( \frac{\sqrt{p_{x_i}^2 + p_{y_i}^2 + p_{z_i}^2}}{m_i c} \right)^2} \right] 
\]

\[
\frac{1}{c} \left[ \frac{p_{y_i}}{m_i} \sqrt{1 + \left( \frac{\sqrt{p_{x_i}^2 + p_{y_i}^2 + p_{z_i}^2}}{m_i c} \right)^2} \right]
\]

\[
\frac{1}{c} \left[ \frac{p_{z_i}}{m_i} \sqrt{1 + \left( \frac{\sqrt{p_{x_i}^2 + p_{y_i}^2 + p_{z_i}^2}}{m_i c} \right)^2} \right]
\]

\[
\frac{1}{c} \left[ F_{i,4}(Y_i, t) + q_i E_{x_i} \right]
\]

\[
\frac{1}{c} \left[ F_{i,5}(Y_i, t) + q_i E_{y_i} \right]
\]

\[
\frac{1}{c} \left[ F_{i,6}(Y_i, t) + q_i E_{z_i} \right]
\]
4.3.1.2 Scaling of Equations with Momentum

We use the momentum and mass scaling as in Eq. 4.31.

\[ \hat{p}_i = \frac{1}{mc} \tilde{p}_i \]
\[ f_i = \frac{m_i}{m}. \]  

(4.31)

Accordingly, the Eq. 4.30 can be written as

\[ \frac{dx_i}{dt} = \frac{p_{x_i}}{c \ m_i \ \sqrt{1 + \left( \frac{\sqrt{p_{x_i}^2 + p_{y_i}^2 + p_{z_i}^2}}{m_i c} \right)^2}} = \frac{p_{x_i}}{m \ c \ f_i \ \sqrt{1 + \left( \frac{\sqrt{\frac{p_{x_i}^2 + p_{y_i}^2 + p_{z_i}^2}{m_i^2 c^2}}} {f_i^2 + m^2 c^2} \right)^2}} \]

= \frac{p_{x_i}}{m \ c \ \sqrt{f_i^2 + \left( \frac{\sqrt{p_{x_i}^2 + p_{y_i}^2 + p_{z_i}^2}}{m^2 c^2} \right)^2}} = \frac{\hat{p}_{x_i}}{\sqrt{f_i^2 + \left( \frac{\sqrt{p_{x_i}^2 + p_{y_i}^2 + p_{z_i}^2}}{m^2 c^2} \right)^2}} \]

(4.32)
Similar expressions can be derived for $y_i$ and $z_i$. The scaled momentum $\hat{p}_{x_i}$ with respect to $\hat{t}$ becomes

$$\frac{d\hat{p}_{x_i}}{d\hat{t}} = \frac{dp_{x_i}}{dt} \frac{d\hat{p}_{x_i}}{dp_{x_i}} = \frac{1}{mc} \frac{dp_{x_i}}{dt}$$

$$= \frac{1}{mc} \frac{1}{c} [F_{i,4}(Y_i, t) + q_i E_{x_i}]$$

$$= \frac{1}{mc^2} F_{i,4}(Y_i, t) + \frac{q n_i}{mc^2} E_{x_i}, \quad (4.33)$$

$$\frac{F_{i,4}(Y_i, t)}{mc^2} = \frac{q_i}{mc^2} \left[ \frac{1}{4\pi \epsilon_0} \sum_{j=1}^{N} \frac{q_j (x_i - x_j)}{[(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2]^{3/2}} \right.$$

$$\left. + B_{z_i} \frac{p_{y_i}}{m_i \sqrt{1 + \left( \frac{\sqrt{p_{x_i}^2 + p_{y_i}^2 + p_{z_i}^2}}{m_i c} \right)^2}} \right]$$

$$- B_{y_i} \frac{p_{z_i}}{m_i \sqrt{1 + \left( \frac{\sqrt{p_{x_i}^2 + p_{y_i}^2 + p_{z_i}^2}}{m_i c} \right)^2}}. \quad (4.34)$$
The three terms in the above Eq. 4.34 can be explicitly written as follows. With \( n_i = \frac{q_i}{q} \) and \( n_j = \frac{q_j}{q} \), the first term becomes

\[
\frac{q_i}{mc^2} \frac{1}{4\pi \epsilon_0} \sum_{j=1}^{N} \left( \frac{q_j (x_i - x_j)}{[(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2]^{3/2}} \right)
\]

\[
= \frac{n_i q_i^2}{4 \pi \epsilon_0 m c^2} \sum_{j=1}^{N} \left( \frac{n_j (x_i - x_j)}{[(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2]^{3/2}} \right).
\]

The second term is

\[
\frac{q_i}{mc^2} B_{z_i} \frac{p_{y_i}}{m_i \sqrt{1 + \left( \frac{\sqrt{p_{x_i}^2 + p_{y_i}^2 + p_{z_i}^2}}{m_i c} \right)^2}} = \frac{q n_i}{m c} B_{z_i} \frac{p_{y_i}}{c m_i \sqrt{1 + \left( \frac{\sqrt{p_{x_i}^2 + p_{y_i}^2 + p_{z_i}^2}}{m_i c} \right)^2}}
\]

\[
= \frac{q n_i}{m c} B_{z_i} \frac{\hat{p}_{y_i}}{\sqrt{f_i^2 + \hat{p}_{x_i}^2 + \hat{p}_{y_i}^2 + \hat{p}_{z_i}^2}}.
\]

The third term is

\[
\frac{q_i}{mc^2} B_{y_i} \frac{p_{z_i}}{m_i \sqrt{1 + \left( \frac{\sqrt{p_{x_i}^2 + p_{y_i}^2 + p_{z_i}^2}}{m_i c} \right)^2}} = \frac{q n_i}{m c} B_{y_i} \frac{\hat{p}_{z_i}}{\sqrt{f_i^2 + \hat{p}_{x_i}^2 + \hat{p}_{y_i}^2 + \hat{p}_{z_i}^2}}.
\]

With \( \hat{Y}_i = (x_i, y_i, z_i, \hat{p}_{x_i}, \hat{p}_{y_i}, \hat{p}_{z_i}) \),

\[
\hat{F}_{i,4}(\hat{Y}_i, t) = \frac{F_{i,4}(Y_i, t)}{mc^2}
\]

\[
\hat{F}_{i,5}(\hat{Y}_i, t) = \frac{F_{i,5}(Y_i, t)}{mc^2}
\]

\[
\hat{F}_{i,6}(\hat{Y}_i, t) = \frac{F_{i,6}(Y_i, t)}{mc^2}.
\] (4.35)
The total \( N \) equations can now be written in a compact form with the initial condition as follows.

\[
\dot{\hat{Y}}' = \begin{bmatrix}
\dot{\hat{Y}}_1 \\
\dot{\hat{Y}}_2 \\
\vdots \\
\dot{\hat{Y}}_{N-1} \\
\dot{\hat{Y}}_N
\end{bmatrix} = \begin{bmatrix}
\hat{F}_1(\hat{Y}_1, \hat{t}) \\
\hat{F}_2(\hat{Y}_2, \hat{t}) \\
\vdots \\
\hat{F}_{N-1}(\hat{Y}_{N-1}, \hat{t}) \\
\hat{F}_N(\hat{Y}_N, \hat{t})
\end{bmatrix} = \hat{F}(\hat{Y}, \hat{t}) \quad \hat{Y}(0) = \hat{Y}_0.
\]

(4.36)
The Eq. 4.36 in detailed form is

\[
\frac{d\hat{Y}_i}{dt} = \left[ \begin{array}{c}
\frac{\hat{p}_{x_i}}{\sqrt{f_i^2 + \hat{p}_{x_i}^2 + \hat{p}_{y_i}^2 + \hat{p}_{z_i}^2}} \\
\frac{\hat{p}_{y_i}}{\sqrt{f_i^2 + \hat{p}_{x_i}^2 + \hat{p}_{y_i}^2 + \hat{p}_{z_i}^2}} \\
\frac{\hat{p}_{z_i}}{\sqrt{f_i^2 + \hat{p}_{x_i}^2 + \hat{p}_{y_i}^2 + \hat{p}_{z_i}^2}} \\
\end{array} \right] + \frac{q n_i}{mc^2} \left[ \begin{array}{c}
\frac{q}{4\pi\epsilon_0} \sum_{j=1}^{N} \frac{n_j (x_i - x_j)}{[\max(0, (x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2, 0)]^{3/2}} + E_{x_i} + c \left( B_{z_i} \hat{v}_{y_i} - B_{y_i} \hat{v}_{z_i} \right) \\
\frac{q}{4\pi\epsilon_0} \sum_{j=1, j \neq i}^{N} \frac{n_j (y_i - y_j)}{[\max(0, (x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2, 0)]^{3/2}} + E_{y_i} + c \left( B_{x_i} \hat{v}_{z_i} - B_{z_i} \hat{v}_{x_i} \right) \\
\frac{q}{4\pi\epsilon_0} \sum_{j=1, j \neq i}^{N} \frac{n_j (z_i - z_j)}{[\max(0, (x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2, 0)]^{3/2}} + E_{z_i} + c \left( B_{y_i} \hat{v}_{x_i} - B_{x_i} \hat{v}_{y_i} \right) \\
\end{array} \right]
\]

\[
= \hat{F}_i(\hat{Y}_i, t). \quad (4.37)
\]

### 4.3.2 Picard Integrator

In the six dimensional phase space, the description of each particle can be given by its three spacial coordinates \((x, y, z)\) and the corresponding momenta \((p_x, p_y, p_z)\). The configuration of the particles and the momenta change with time. Evaluating the ever changing
position and momenta can be considered solving a dynamical problem. We employ the Picard iteration based integrator to solve the $6N$ differential equations [39]. Picard iteration [43] method finds the solution to an initial value problem (IVP) in the form of a Taylor series by generating a sequence of functions. In general, the integral equation with the initial condition can be given as,

$$\frac{dy}{dt} = y' = f(t, y) \quad \text{with} \quad y(t_0) = y_0.$$ (4.38)

According to the theorem on Existence and Uniqueness [43], if $f(t, y)$ and $\frac{\partial f}{\partial y}$ are continuous in a certain region around a particular point $(t_0, y_0)$, then there is a unique solution to Eq. 4.38. It takes the initial value $y_0$ guessing and then repeatedly applies $f(t, y)$ to get a better solution. Hence, the solution for the equation can be expressed as,

$$Y_0(t) = y_0$$ (4.39)

$$Y_{n+1}(t) = y_0 + \int_{t_0}^{t} f(s, Y_n(s)) ds \quad \text{with} \quad n \geq 0.$$

If $f(t, y)$ is a single-valued continuous function of $t$, and $y$ and if the function satisfies the Lipschitz condition, then Eq. 4.39 is said to converge to the exact solution given by,

$$y(t) = \lim_{n \to \infty} Y_n(t).$$ (4.40)

**Example**

$$y' = ty^2; y(0) = 1.$$  

In the integral form

$$y(t) = y(0) + \int_{t_0}^{t} sy(s)^2 ds.$$
The general solution based on the iterative format

\[ Y_{n+1}(t) = y_0 + \int_{t_0}^{t} f(s, Y_n(s)) ds = 1 + \int_{t_0}^{t} s Y_n^2 ds. \]  \hspace{1cm} (4.41)

When \( n = 0 \), we can get \( Y_1 \), if the initial guess is

\[ Y_0(t) = y(0). \]

Hence, the solutions for Eq. 4.38 generated using four iterations are

\[ Y_0(t) = 1. \]
\[ Y_1(t) = 1 + \int_{t_0}^{t} s(-1)^2 ds = 1 + \frac{1}{2} t^2. \]
\[ Y_2(t) = 1 + \int_{t_0}^{t} s(-1 + \frac{1}{4} t^2)^2 ds = 1 + \frac{1}{2} t^2 - \frac{1}{4} t^3 + \frac{1}{24} t^6. \]
\[ Y_3(t) = 1 + \int_{t_0}^{t} s(1 + \frac{1}{4} t^2 - \frac{1}{8} t^6 + \frac{1}{24} t^4 - \frac{1}{96} t^8 - \frac{1}{576} t^{12}) ds \]
\[ = 1 + \frac{1}{2} t^2 - \frac{1}{4} t^4 + \frac{1}{8} t^6 - \frac{1}{24} t^8 + \frac{1}{96} t^{10} - \frac{1}{576} t^{12} + \frac{1}{8064} t^{14}. \]

In the Picard iteration based integrator, iterations are performed to advance a solution from the initial point to the final point via a sequence of time steps. We can exploit three advantages in this integrator, viz. it is time adaptive, and has dense output and arbitrary order. Since the accuracy of the solution is typically inversely proportional to the time step size, selecting the optimal time step size is crucial. However, it is not trivial. Too small time steps will jeopardize the overall computational efficiency. Hence, there is a trade-off between efficiency and accuracy. If small time steps are required, the Picard order must be reduced in order to maintain the accuracy. Similarly, when the larger time steps are appropriate, the order can be enhanced to improve the accuracy.
4.3.3 Strang Operator Splitting

Since there are two fields, external and space charge, the Hamiltonian can be split to two parts.

\[ H = H_1 + H_2. \]

Accordingly, symplectic [41] operator splitting method can be employed to solve the differential equations. We use the operator splitting method, Strang splitting, which is second order accurate [44]. The purpose of operator splitting is to split the complicated problem into simpler components and compose the solutions of these components to get the solution of the complicated system. Since the FMM is the most time-wise expensive procedure, we need to minimize the number of FMM calls. By means of Strang splitting, we split the problem into two simpler parts; near and and far range with the applied field. To get the full solution, solutions of the two parts can be composed as

\[ \phi(\tau) = \phi_1(\tau/2) \circ \phi_2(\tau) \circ \phi_1(\tau/2) + O(\tau^3), \tag{4.42} \]

where \( \tau \) is the time step size and \( \phi_1, \phi_2 \) are the solutions of \( H_1 \) and \( H_2 \), respectively. Eq. 4.42 indicates that the small time steps give better accuracy. As mentioned earlier, the FMM method used for the far or long range calculations is a time-wise costly operation. Each particle in the beam experiences two types of forces, fast varying forces and slow varying forces. When the test particle has close encounters with its neighbors, the test particle undergoes rapid fluctuations or fast varying forces. The rest of the particles in the distribution exert a force on the test particle as a collective effect which can be viewed as the mean field. The force due to the mean field varies slowly with time, and it is justifiable to consider it as unchanged. To the extent that this is correct, we have to decide the time step size such
that during this time step the slow varying forces stay approximately unchanged. Since it is difficult to model both fast varying forces and slow varying forces with the same time step size, we split the time step size used for the slow varying forces to get a smaller step size for fast varying forces. Therefore, we have two different time steps. The fast varying forces depend on the positions of particles and relative speed. The smaller time step can be considered as individualized adaptive time step. Reduction of number of FMM calls is an effective way to reduce the computational cost. Hence, we call FMM only for the larger time steps [45].

4.3.3.1 Preparation of Equations for Strang Splitting

The Eq. 4.37 can be split into two functions, near and far. The neighborhood of the $i^{th}$ particle is the near part and it is denoted by 1. The other part (far part) is denoted by 2. Hence,

\[
\hat{F}(\hat{Y}, \hat{t}) = \left[ \begin{array}{c}
\hat{Y}_1 \\
\hat{Y}_2 \\
\vdots \\
\hat{Y}_{N-1} \\
\hat{Y}_N
\end{array} \right] = \left[ \begin{array}{c}
\hat{F}^{[1]}_1(\hat{Y}_1, \hat{t}) \\
\hat{F}^{[1]}_2(\hat{Y}_2, \hat{t}) \\
\vdots \\
\hat{F}^{[1]}_{N-1}(\hat{Y}_{N-1}, \hat{t}) \\
\hat{F}^{[1]}_N(\hat{Y}_N, \hat{t})
\end{array} \right] + \left[ \begin{array}{c}
\hat{F}^{[2]}_1(\hat{Y}_1, \hat{t}) \\
\hat{F}^{[2]}_2(\hat{Y}_2, \hat{t}) \\
\vdots \\
\hat{F}^{[2]}_{N-1}(\hat{Y}_{N-1}, \hat{t}) \\
\hat{F}^{[2]}_N(\hat{Y}_N, \hat{t})
\end{array} \right]
\]

\[
= \hat{F}^{[1]}(\hat{Y}, \hat{t}) + \hat{F}^{[2]}(\hat{Y}, \hat{t}), \quad \hat{Y}(0) = \hat{Y}_0.
\]

(4.43)
If the $i^{th}$ particle has a set of particles in its neighborhood, and if it belongs to the set $S_i$, the near part of Eq. 4.43 is

$$\hat{F}_{i}^{[1]}(\hat{Y}_i, \hat{t}) = \left[ \begin{array}{c} \frac{\hat{p}_{x_i}}{\sqrt{\hat{f}_i^2 + \hat{p}_{x_i}^2 + \hat{p}_{y_i}^2 + \hat{p}_{z_i}^2}} \\ \frac{\hat{p}_{y_i}}{\sqrt{\hat{f}_i^2 + \hat{p}_{x_i}^2 + \hat{p}_{y_i}^2 + \hat{p}_{z_i}^2}} \\ \frac{\hat{p}_{z_i}}{\sqrt{\hat{f}_i^2 + \hat{p}_{x_i}^2 + \hat{p}_{y_i}^2 + \hat{p}_{z_i}^2}} \end{array} \right]$$

$$\frac{q}{4\pi\epsilon_0} \frac{n_i}{mc^2} \sum_{j \neq i}^{j \in S_i} \frac{n_j(x_i - x_j)}{[(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2]^{3/2}} + E_{x_i} + c \left( B_{z_i} \hat{v}_{y_i} - B_{y_i} \hat{v}_{z_i} \right)$$

$$\frac{q}{4\pi\epsilon_0} \frac{n_i}{mc^2} \sum_{j \neq i}^{j \in S_i} \frac{n_j(y_i - y_j)}{[(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2]^{3/2}} + E_{y_i} + c \left( B_{x_i} \hat{v}_{z_i} - B_{z_i} \hat{v}_{x_i} \right)$$

$$\frac{q}{4\pi\epsilon_0} \frac{n_i}{mc^2} \sum_{j \neq i}^{j \in S_i} \frac{n_j(z_i - z_j)}{[(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2]^{3/2}} + E_{z_i} + c \left( B_{y_i} \hat{v}_{x_i} - B_{x_i} \hat{v}_{y_i} \right)$$

(4.44)
and the second part is

$$\hat{F}^{[2]}(\hat{Y}, \hat{t}) = \begin{bmatrix}
0 \\
0 \\
0 \\
\end{bmatrix}
\begin{bmatrix}
\frac{q^2}{4\pi\epsilon_0 \, m \, c^2} \, n_i \sum_{j \notin S_i} \frac{n_j (x_i - x_j)}{[(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2]^{3/2}} \\
\frac{q^2}{4\pi\epsilon_0 \, m \, c^2} \, n_i \sum_{j \notin S_i} \frac{n_j (y_i - y_j)}{[(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2]^{3/2}} \\
\frac{q^2}{4\pi\epsilon_0 \, m \, c^2} \, n_i \sum_{j \notin S_i} \frac{n_j (z_i - z_j)}{[(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2]^{3/2}}
\end{bmatrix}.$$  \hspace{1cm} (4.45)

By applying the Strang operator splitting as explained in Eq. 4.42, we can solve three simple equations for a single time step of size $h$.

$$\begin{align*}
\hat{Y}'(0) &= \hat{Y}(0) = \hat{Y}_0, & \text{at } t = \frac{h}{2} \\
\hat{Y}'(0) &= \hat{Y}(0) = \phi_{h/2,1}^{[2]}(\hat{Y}_0), & \text{at } t = h \hspace{1cm} (4.46) \\
\hat{Y}'(0) &= \hat{Y}(0) = \phi_{h,1}^{[1]} \circ \phi_{h/2,1}^{[2]}(\hat{Y}_0), & \text{at } t = \frac{h}{2}.
\end{align*}$$
The solutions of these three equations are $\phi_{h/2,1}^2$, $\phi_{h,1}^1$ and $\phi_{h/2,2}^2$, respectively. The final solution can be obtained by composing them.

$$\hat{Y}_1 = S_h^1(\hat{Y}_0) = \phi_{h/2,2}^2 \circ \phi_{h,1}^1 \circ \phi_{h/2,1}^2(\hat{Y}_0).$$

(4.47)

The Eq. 4.46 can be generalized as,

$$\hat{Y}' = \hat{F}^{[2]}(\hat{Y}, \hat{t}) \quad \hat{Y}(0) = \hat{Y}_{k-1}, \quad \text{at } t = \frac{h}{2}$$

$$\hat{Y}' = \hat{F}^{[1]}(\hat{Y}, \hat{t}) \quad \hat{Y}(0) = \phi_{h/2,(2k-1)}^{[2]}(\hat{Y}_{k-1}), \quad \text{at } t = h$$

$$\hat{Y}' = \hat{F}^{[2]}(\hat{Y}, \hat{t}) \quad y(0) = \phi_{h,k}^{[1]} \circ \phi_{h/2,2k-1}^{[2]}(\hat{Y}_{k-1}), \quad \text{at } t = \frac{h}{2}.$$  

(4.48)

Accordingly, the solutions of these three equations are $\phi_{h/2,2k-1}^{[2]}$, $\phi_{h,k}^{[1]}$ and $\phi_{h/2,2k}^{[2]}$. The final solution is

$$\hat{Y}_k = S_h^k(\hat{Y}_{k-1}) = \phi_{h/2,2k}^2 \circ \phi_{h,k}^1 \circ \phi_{h/2,2k-1}^2(\hat{Y}_{k-1}),$$

(4.49)

where $k = 1, 2, ..., N$. This process can be repeatedly applied over $n$ time steps and the final solution can be expressed as the composition of all solutions.

$$Y_n = \phi_{h/2,2n}^2 \circ \phi_{h,n}^1 \circ \phi_{h/2,2n-1}^2 \circ \phi_{h/2,2n-2}^2 \circ \phi_{h,n-1}^1 \circ \phi_{h/2,2n-3}^2 \circ \cdots$$

$$\circ \phi_{h/2,4}^2 \circ \phi_{h,2}^1 \circ \phi_{h/2,3}^2 \circ \phi_{h/2,2}^2 \circ \phi_{h,1}^1 \circ \phi_{h/2,1}^2(\hat{Y}_0).$$

(4.50)

The near region solution is $\phi^{[1]}$, and the far region solution is $\phi^{[2]}$. The Picard iteration based integrator is used to generate the near region solutions of the Eq. 4.44 whereas the FMM
can be used to generate the far region solutions of the Eq. 4.56. It is worthwhile to notice that pairs $\phi_{h/2,2k} \circ \phi_{h/2,2k+1}$ for $2 \leq k \leq n - 1$ can be computed for one FMM call if the contributions of particles in the neighborhood at consecutive time steps stay unaltered. In other words, the particle indices of two sets $S_i(t_k)$ and $S_i(t_k + 1)$ must be the same.

### 4.3.4 Three Integrators in PHAD

Three integrators were implemented, the regular long range integrator, the modified long range integrator, and the Picard integrator. The regular long range integrator generates odd values $\phi_{h/2,2k+1}$ and the modified long range integrator generates even values $\phi_{h/2,2k}$. The regular long range integrator is applied on the set $S_i$ defined by the FMM on the positions before applying a particular Picard integrator. Similarly, the modified long range integrator is applied on the set $S_i$ defined by the FMM on the positions after a particular Picard integrator. The final solution $Y_n$ after $n$ time steps is

$$
\phi_{h/2,2n} \circ \phi_{h,n} \circ \phi_{h/2,2n-1} \circ \phi_{h/2,2n-2} \circ \phi_{h,n-1} \circ \phi_{h/2,2n-3} \circ \cdots \circ \phi_{h/2,4} \circ \phi_{h,2} \circ \phi_{h/2,3} \circ \phi_{h/2,2} \circ \phi_{h,1} \circ \phi_{h/2,1}(Y_0).
$$

(4.51)

In Eq. 4.51, the terms indicated in blue text are the solutions obtained from the Picard integrator. Also, the bold text indicates the solutions of the modified long range integrator while the regular text indicates the solutions of the long range integrator.

#### 4.3.4.1 Regular Long Range Integrator and the Picard Integrator

In the stand-alone FMM, the local expansions calculated in the M2L procedure will be transferred from the parent node to child nodes by means of the L2L procedure to get the
local expansions of child nodes. Then, the child node’s local expansion is evaluated at each target in the L2P procedure. As the final step, point-to-point direct (P2P) evaluation was carried out to get the contribution of neighbor sources on each target. In PHAD, there is no use of L2P, P2P, and the direct evaluation. Instead, two new procedures have been added to the PHAD: LNGRNGINT to implement the long range integrator and PICARDINT to implement the Picard iteration based integrator. The former replaces the L2P and the P2P while the latter replaces the direct evaluation.

The multipole-to-local (M2L) procedure in the FMM calculates the potential or the field created by the particles outside a particular particle’s neighborhood $S_i$. These values are indicated by the terms $\phi_{k/2,2k-1}^{[2]}$ with $k = 1, 2, ..., n$. The far region differential equations given in Eq. 4.56 has zero values for the first three components, and it indicates that there is no change in positions of particles. Accordingly, the last three components take the form

$$\frac{dy}{dt} = C \quad y(0) = y_0,$$

which has the solution $y_0 + C t$, where $C$ is a constant. The solution to Eq. 4.56 with initial condition

$$\hat{Y}_{i0} = [x_i^0 \; y_i^0 \; z_i^0 \; p_x^0 \; p_y^0 \; p_z^0]^T$$
is then

\[
\begin{align*}
\dot{Y}_i(h) = & \begin{bmatrix}
x_i^0 \\
y_i^0 \\
z_i^0 \\
\end{bmatrix} \\
& \begin{aligned}
p_{x_i}^0 &+ h \frac{q^2}{4\pi\epsilon_0 m c^2} n_i \left( \sum_{j \not\in S_i} \frac{n_j (x_i - x_j)}{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}^{3/2} \right) \\
p_{y_i}^0 &+ h \frac{q^2}{4\pi\epsilon_0 m c^2} n_i \left( \sum_{j \not\in S_i} \frac{n_j (y_i - y_j)}{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}^{3/2} \right) \\
p_{z_i}^0 &+ h \frac{q^2}{4\pi\epsilon_0 m c^2} n_i \left( \sum_{j \not\in S_i} \frac{n_j (z_i - z_j)}{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}^{3/2} \right) \\
\end{aligned}
\end{align*}
\tag{4.52}
\]

The last three components in Eq. 4.52 include the forces calculated in the FMM. The particle coordinates in the FMM are scaled as shown in Eq. 3.23. Therefore, the terms in the numerator can be expressed in the scaled coordinates.

\[
x_i - x_j = (D_0 \times \tilde{x}_i + x_{\text{min}}) - (D_0 \times \tilde{x}_j + x_{\text{min}}) = D_0(\tilde{x}_i - \tilde{x}_j)
\]
and

\[
[(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2]^{3/2} = [D_0^2(\bar{x}_i - \bar{x}_j)^2 + D_0^2(\bar{y}_i - \bar{y}_j)^2 + D_0^2(\bar{z}_i - \bar{z}_j)^2]^{3/2}
= D_0^3[(\bar{x}_i - \bar{x}_j)^2 + (\bar{y}_i - \bar{y}_j)^2 + (\bar{z}_i - \bar{z}_j)^2]^{3/2}. \quad (4.53)
\]

\(D_0\) is defined in 3.2.3.2. With the insertion of the FMM scaling, Eq. 4.52 becomes

\[
\hat{Y}_i(h) = \begin{bmatrix}
\hat{x}_i^0 \\
\hat{y}_i^0 \\
\hat{z}_i^0
\end{bmatrix} = \begin{bmatrix}
p_{x_i}^0 + h & \frac{q^2}{4\pi\epsilon_0} & \frac{\mathbf{m}^2}{c^2} & D_0^2 & n_i & \sum_{j \notin S_i} \frac{n_j(\bar{x}_i - \bar{x}_j)}{[(\bar{x}_i - \bar{x}_j)^2 + (\bar{y}_i - \bar{y}_j)^2 + (\bar{z}_i - \bar{z}_j)^2]^{3/2}} \\
\frac{q^2}{4\pi\epsilon_0} & \frac{\mathbf{m}^2}{c^2} & \frac{D_0^2}{n_i} & \sum_{j \notin S_i} \frac{n_j(\bar{y}_i - \bar{y}_j)}{[(\bar{x}_i - \bar{x}_j)^2 + (\bar{y}_i - \bar{y}_j)^2 + (\bar{z}_i - \bar{z}_j)^2]^{3/2}} \\
\frac{q^2}{4\pi\epsilon_0} & \frac{\mathbf{m}^2}{c^2} & \frac{D_0^2}{n_i} & \sum_{j \notin S_i} \frac{n_j(\bar{z}_i - \bar{z}_j)}{[(\bar{x}_i - \bar{x}_j)^2 + (\bar{y}_i - \bar{y}_j)^2 + (\bar{z}_i - \bar{z}_j)^2]^{3/2}}
\end{bmatrix}.
\quad (4.54)
\]
In terms of the electric field computed in the FMM, Eq. 4.54 becomes

\[
\hat{Y}_i(h) = \begin{bmatrix}
\tilde{x}_i^0 \\
\tilde{y}_i^0 \\
\tilde{z}_i^0 \\
p_{x_i}^0 + h \frac{q^2}{4\pi\varepsilon_0 m c^2 D_0^2} n_i \tilde{E}_{x_i} \\
p_{y_i}^0 + h \frac{q^2}{4\pi\varepsilon_0 m c^2 D_0^2} n_i \tilde{E}_{y_i} \\
p_{z_i}^0 + h \frac{q^2}{4\pi\varepsilon_0 m c^2 D_0^2} n_i \tilde{E}_{z_i}
\end{bmatrix}.
\] (4.55)

### 4.3.4.2 Modified Long Range Integrator

Another new addition to the PHAD is the procedure called MODLNGRNGINT, which is to implement the modified long range integrator. After each Picard iteration, the source particles in the neighborhood of a particular target particle change their positions. Due to this reason certain particles move outside the neighborhood while certain particles in the far region enter the neighborhood. To rectify this issue, we apply the modified long range integrator after each Picard procedure. This situation is illustrated in Fig. 4.2. The small red square represents a particular target in the gray box, and the set of blue boxes represents its neighborhood after the procedure long range modified integrator. Two particles, 3 and 15, are displaced after the Picard procedure. In other words, the source 3 left the neighborhood
and the source 15 entered the neighborhood. Before proceeding to the next time step and applying the long range integrator, the modified integrator is immediately applied to establish the previous neighborhood by adding source 3 and removing source 15.

The sets $S_i$ and $S_i^*$ are based on the positions before and after applying the Picard integrator, respectively. The computations are performed based on the current positions $S_i^*$. Hence, we need to compute the two sets $S_i \setminus S_i^*$ and $S_i^* \setminus S_i$. Here, $\setminus$ notation is used to indicate the relative complement. For instance, $S_i \setminus S_i^*$ gives the set of elements in $S_i$ but not in $S_i^*$. The \texttt{COMPARE} function (Appendix B) is used to find these sets. In set theory terminology, it can be given by

\[
S_i^c = (S_i^* \setminus S_i) \cup (S_i^{*c} \setminus S_i) = (S_i^* \setminus S_i) \cup (S_i^{*c} \setminus (S_i \setminus S_i^*)).
\]

\[
S_i = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}.
\]

\[
S_i^* = \{1, 2, 4, 5, 6, 7, 8, 9, 10, 11, 12, 15\}.
\]

\[
S_i^{*c} = \{3, 13, 14\}.
\]

The superscript $c$ denotes the complement. Hence

\[
S_i^c = (S_i^* \setminus S_i) \cup (S_i^{*c} \setminus S_i) = (S_i^* \setminus S_i) \cup (S_i^{*c} \setminus (S_i \setminus S_i^*)) = \{13, 14, 15\}.
\]

The modified long range integrator is used to solve Eq. 4.56, and it is applied on the neighborhood generated by the FMM based on the particles’ position before applying the Picard
Figure 4.2: After applying the long range integrator, the source particles (small blue squares) in the neighborhood (blue boxes) and outside the neighborhood of the target (red square) are shown in (a). The Picard integrator changes the location of the source particles. By applying the modified integrator we can move the displaced sources back to the previous locations in order to keep the neighborhood unchanged.

As mentioned earlier the even solutions of the modified long range integrator are denoted by $\phi^{[2]}_{k/2,2k}$.

$$
\hat{F}^{[2]}_s(\hat{Y}, \hat{t}) = \left\{ \begin{array}{c} 0 \\ 0 \\ 0 \\ \frac{q^2}{4\pi\varepsilon_0 \ m \ \frac{c^2}{m}} n_i \left[ \sum_{j \notin S_i} \frac{n_j(x_i - x_j)}{[(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2]^{3/2}} \right] \\
\frac{q^2}{4\pi\varepsilon_0 \ m \ \frac{c^2}{m}} n_i \left[ \sum_{j \notin S_i} \frac{n_j(y_i - y_j)}{[(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2]^{3/2}} \right] \\
\frac{q^2}{4\pi\varepsilon_0 \ m \ \frac{c^2}{m}} n_i \left[ \sum_{j \notin S_i} \frac{n_j(z_i - z_j)}{[(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2]^{3/2}} \right] \end{array} \right. \right) . \quad (4.56)
$$
This can be written as

\[
\frac{q^2}{4\pi\epsilon_0 \ mc^2} n_i \left[ \sum_{j \in S_i} \frac{n_j(x_i - x_j)}{[(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2]^{3/2}} \right]
\]

\[
= \frac{q^2}{4\pi\epsilon_0 \ mc^2} n_i \left[ \sum_{j \in S_i \setminus S_i} \frac{n_j(x_i - x_j)}{[(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2]^{3/2}} \right]
\]

\[
+ \sum_{j \in S_i \setminus (S_i \cup S_i^*)} \frac{n_j(x_i - x_j)}{[(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2]^{3/2}}
\]

\[
- \sum_{j \in S_i \setminus S_i^*} \frac{n_j(x_i - x_j)}{[(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2]^{3/2}}
\]

\[
= \frac{q^2}{4\pi\epsilon_0 \ mc^2} n_i \left[ \sum_{j \in S_i \setminus S_i} \frac{n_j(x_i - x_j)}{[(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2]^{3/2}} + \tilde{E}_x^n \right]
\]

\[
- \sum_{j \in S_i \setminus S_i^*} \frac{n_j(x_i - x_j)}{[(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2]^{3/2}} \right].
\]

(4.57)
The modified Eq. 4.56 can be obtained using Eq. 4.57.

\[
\hat{F}_{i}^{[2]}(\hat{Y}_{i}, \hat{t}) = \frac{q^2}{4\pi\epsilon_0 m c^2} n_i \left[ \hat{E}_x + \sum_{j \in S_i \setminus S_i} \frac{n_j (x_i - x_j)}{[(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2]^{3/2}} \right. \\
- \left. \sum_{j \in S_i \setminus S_i^*} \frac{n_j (x_i - x_j)}{[(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2]^{3/2}} \right] \\
\frac{q^2}{4\pi\epsilon_0 m c^2} n_i \left[ \hat{E}_y + \sum_{j \in S_i \setminus S_i} \frac{n_j (y_i - y_j)}{[(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2]^{3/2}} \right. \\
- \left. \sum_{j \in S_i \setminus S_i^*} \frac{n_j (y_i - y_j)}{[(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2]^{3/2}} \right] \\
\frac{q^2}{4\pi\epsilon_0 m c^2} n_i \left[ \hat{E}_z + \sum_{j \in S_i \setminus S_i} \frac{n_j (z_i - z_j)}{[(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2]^{3/2}} \right. \\
- \left. \sum_{j \in S_i \setminus S_i^*} \frac{n_j (z_i - z_j)}{[(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2]^{3/2}} \right] .
\] (4.58)

4.3.4.3 The FMM order in PHAD

The accuracy of PHAD results are FMM order dependent as shown in Fig. 4.3. Higher FMM orders increase the accuracy. However, they are time-wise expensive [45]. Also, the
Figure 4.3: As the FMM order increases, the accuracy of PHAD increases.

accuracy of PHAD can be enhanced by increasing the number of time steps and decreasing the time step size (Table 4.2) as shown in Fig. 4.4.

Table 4.2: The accuracy of the PHAD improves as the number of time steps increases by decreasing the time step size.

<table>
<thead>
<tr>
<th>Total time units</th>
<th>Time step Size</th>
<th>Time Step</th>
<th>Average x Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>2</td>
<td>100</td>
<td>$9.19 \times 10^{-9}$</td>
</tr>
<tr>
<td>200</td>
<td>1</td>
<td>200</td>
<td>$8.91 \times 10^{-9}$</td>
</tr>
<tr>
<td>200</td>
<td>0.5</td>
<td>400</td>
<td>$8.84 \times 10^{-9}$</td>
</tr>
</tbody>
</table>
Figure 4.4: As the number of time steps increases (while decreasing the time step size), the accuracy of PHAD increases.

4.3.4.4 Time Scales and Time Step Sizes

The simulation progresses on three time scales: The largest time step is the time taken by particles to traverse the cooling section once, which is equivalent to one-turn time scale. This is estimated by subdividing the total cooling time required to cool the beam completely in the relevant cooling section, in the booster or in the large booster, and denoted by $T$ in Eq. 4.59. The second is the time scale of slow varying forces, which is used in the FMM. This is denoted by $\Delta t_i \ (i=1, 2, ... , n)$ in Eq. 4.59. The third is the time scale of rapidly varying forces, and it is used in the Picard integrator as shown in Fig. 4.5 as $\Delta (\Delta t_i)$. This time scale depends on the particle-to-particle interaction force. Hence, the third time step size is adaptive. Within the time step used in the Picard integrator, we can perform many number
Figure 4.5: Each Picard time step $\Delta t_n$ is further divided into even smaller time steps of size $\Delta(\Delta t_n)$. The total time $T$ has $n$ number of time steps. The red arrow indicates the direction of the time progression.

Despite all the above discussed new methods and improvements, the serial version is still too slow for realistic electron cooling simulations. Therefore, it is inevitable that code needs parallelization.

we developed the MPI-based parallel version of PHAD since COSY Infinity 9.1 supports MPI. This is extensively discussed in Chapter 5.

Up to now, we have curtailed flexibility of the Picard integrator by specifying the Picard order and the Picard time step size at the beginning of the simulation. In order to exploit the full advantage of the Picard integrator, we can modify the PHAD code by introducing two variables, adaptive Picard order and adaptive Picard time step size. If the distance between particles is small, the Picard integrator can reduce the time step size accordingly
and improve the accuracy. On the other hand, if the distance between particles are com-
paratively large, the time step size can be increased, and it leads to enhance the efficiency.
These upgrades of the PHAD code will be done in future.
In this research, we developed three in-house codes. Their applications demand high accuracy; therefore, we verify the correctness and veracity of the codes by performing benchmarking tests. First, we developed the $N$-body code (Section 5.3) and compared results with those of the point-to-point method, and verified the accuracy. Therefore, we can justify that the Picard iteration based integrator, which is used in the $N$-body code to solve differential equations, performs accurately. Second, the stand-alone FMM was benchmarked against the point-to-point method. At higher orders of the FMM, results converge to the point-to-point method results as shown in 3.2.5.2. Finally, the PHAD code was benchmarked against the $N$-body code (detailed explanation in 5.3). These results showed a fine agreement signifying the correctness of implementation of the FMM and Strang splitting in the PHAD.

5.1 Data Structuring

Data structuring is needed to organize the configuration of the charged particles. This data structuring part is written in C++. As described in Chapter 3, given the input configuration of the targets and the sources along with the parameter value $q$ (the maximum number of sources allowed in the neighborhood of a particular target), the data structure can be fully defined. The primary data structure we used to implement the adaptive FMM algorithm was an octree (or quadtree in 2D). An octree is a tree data structure in which each internal node has eight children; it is typically used to partition a 3D space. Each node of
the octree represents a box. Establishing an octree to save all necessary information makes the code run fast because the traversal time of the octree is $O(h)$, where $h$ is the height of the tree or the number of levels.

**Step 1:** Construct the root box. In order to ensure that the unit box encloses all particles, we modify the scaling factor $D_0$ by adding a smaller value $10^{-11}$ to it. This root box of side length 1, which is denoted by the universal index (0, 0), contains all targets and sources. Construction of the root box is in the constituent code files, `particles.h` and `particles.cpp`, of the C++ code.

**Step 2:** Divide the root box once into equal octants and create 8 boxes in level 1. The box indices are (0, 1), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (6, 1), and (7, 1). The first digit in the parenthesis denotes the box number, and the second digit denotes the level. Divide each octant one more time to create 64 level 2 boxes. The level 2 box indices are $(n, 2)$, with $n = \{0, 63\}$. These boxes are called octree nodes, and the `subDivide()` function in `octree.cpp` is used to perform the subdividing process. The `OctreeNode` class is defined in `octree.h` and its members save the information of the node as shown in Table 5.1.

Once the level 2 boxes are created, the ones that contain targets and whose neighborhood has fewer than $q$ sources are added to the $D$-tree as level 2 boxes, a process called tagging. The other non-empty boxes with targets are divided to create level 3 boxes. Empty boxes are not subdivided any further. As was with level 2, the level 3 boxes containing targets and whose neighborhood has fewer than $q$ sources are tagged as level 3 boxes set in the $D$-tree. This repetitive box partitioning process must be continued until all boxes with targets satisfy the above condition. The highest or the finest level reached is denoted by $l_{\text{max}}$. This task is implemented by the `tagAndDivide()` function in `main.cpp`.

**Step 3:** Construct the $D$-tree. The process in step 2 determines that the lowest and the largest level of the target box set is 2 and $l_{\text{max}}$, respectively. Starting from the largest level
Table 5.1: Information saved in an OctreeNode class object.

<table>
<thead>
<tr>
<th>Member</th>
<th>Information Saved</th>
</tr>
</thead>
<tbody>
<tr>
<td>m_level</td>
<td>level of the box</td>
</tr>
<tr>
<td>m_num</td>
<td>box number</td>
</tr>
<tr>
<td>m_xmin</td>
<td>front-bottom-left x coordinate</td>
</tr>
<tr>
<td>m_ymin</td>
<td>front-bottom-left y coordinate</td>
</tr>
<tr>
<td>m_zmin</td>
<td>front-bottom-left z coordinate</td>
</tr>
<tr>
<td>m_length</td>
<td>side length of the box</td>
</tr>
<tr>
<td>m_ChildNodes</td>
<td>level and the box number of 8 children</td>
</tr>
<tr>
<td>m_ParentNode</td>
<td>level and the box number of the parent</td>
</tr>
<tr>
<td>m_IndexInAllCLsetNodes</td>
<td>the box index in the C-box array</td>
</tr>
<tr>
<td>m_IndexInAllDtreeNodes</td>
<td>the box index in the D-box array</td>
</tr>
<tr>
<td>m_source_particles</td>
<td>number of sources in the box</td>
</tr>
<tr>
<td>m_target_particles</td>
<td>number of targets in the box</td>
</tr>
<tr>
<td>m_neighbors</td>
<td>level and the box number of all neighbors</td>
</tr>
</tbody>
</table>

the child-parent relationship can be established by linking the level \( l \) child nodes with level \((l - 1)\) parent nodes. The `constructDtree()` function in `main.cpp` creates the \( D \)-tree.

**Step 4:** Construct the \( C \)-forest. The set of boxes with sources form the \( CL \)-set. As described in Chapter 3, \( C \)-trees are built by traversing through the \( CL \)-set boxes from the finest level to the coarser level while linking parent node with its child nodes.

## 5.2 Coulomb Force Calculation

Three simulation codes have been developed to calculate the Coulomb interaction, the \( N \)-body code, the stand-alone FMM, and the PHAD (Table 5.2). All of them are written in COSY INFINITY 9.1.
Table 5.2: The description of three codes.

<table>
<thead>
<tr>
<th>Code</th>
<th>Techniques used</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>N-body</td>
<td>Picard Integrator</td>
<td>Potential/field calculation</td>
</tr>
<tr>
<td>Stand-alone FMM</td>
<td>FMM, Picard Integrator, Strang splitting</td>
<td>Potential/field calculation</td>
</tr>
<tr>
<td>PHAD</td>
<td></td>
<td>Potential/field calculation</td>
</tr>
</tbody>
</table>

5.3 The N-body Code

Validation of the PHAD code was performed by means of another code called *N-body code*. This code calculates Coulomb interactions among particles as we do in the pair-wise method. Therefore, the N-body code does not use the FMM or Strang splitting. To solve the differential equations, however, it uses the Picard integrator.

5.3.1 Benchmarking the PHAD code against the N-body code

5.3.1.1 Particle Trajectories

Using 100 charged particles, each carrying a unit charge and a unit mass, the trajectories were measured as shown in Fig. 5.1. Since the N-body code is not efficient we used a small number of particles. The total number of time steps is 400, and the time step size is 0.5. According to Fig. 5.1, the PHAD and the N-body code trajectories behave in a similar manner. Also, the absolute errors calculated using the $x$ coordinates measured in the trajectories produced by the PHAD and the N-body code for each time step are shown in the Fig. 5.2. This error can further be mitigated by increasing the FMM order and decreasing
For the same time step size, we can increase the accuracy as depicted in Fig. 5.3 by increasing the FMM order in the PHAD code.

**Particle Trajectories for One Proton and One Electron**

We performed another test for $10^6$ time steps using one electron and one proton in a 1 T solenoidal magnetic field, which was in the $z$ direction. The time step size was 0.001 meters. As we expected, both particles make helical paths around the magnetic field lines (Fig. 5.5). This test further assures the accuracy of the PHAD code.
5.3.1.2 Position and Momentum Plots

Small time step size, 0.001 meters

The $x-z$ plot and the $p_y - p_x$ (momentum) plot are generated to compare the PHAD code with the $N$-body code (Fig. 5.4). For these plots, we used 100 particles with unit mass and charge. These plots show that each point on the $N$-body code plot perfectly coincides with the points on the PHAD plot. The parameters used for this test are listed in Table 5.3. The relative errors of $x$ coordinates of all particles measured after 0.001m (total time) are shown in Fig. 5.6. The maximum relative error is $\sim 10^{-8}$. If the FMM order is increased, the relative error can be further reduced. Therefore, if the time step size is small and the FMM order is high, the two codes converge to each other.

Large time step size, 2 meters

With the same Picard order and the FMM order, the time step size was increased to 2 meters. The $x-z$ plot and the $p_y - p_x$ (momentum) plot are generated to compare the
Figure 5.3: The absolute error calculated for the x values for FMM orders 2 and 12. The total number of time steps is 200 with time step size is 0.025 meters.

Figure 5.4: $z - x$ plot and the momentum in $x - y$ plane ($p_x - p_y$) for the N-body code and the PHAD. All points in both plots overlap indicating that the results are in good agreement. The time step size is 0.001 meters.
a. Proton trajectory  

b. Electron trajectory

Figure 5.5: Particle trajectories in the solenoidal magnetic field of 1 T.

Table 5.3: Parameters used in the $N$-body code and the PHAD code for the small time step size, 0.001 meters.

<table>
<thead>
<tr>
<th>Code</th>
<th>Parameter</th>
<th>Units</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$-body code</td>
<td>time step size</td>
<td>meters</td>
<td>0.0025</td>
</tr>
<tr>
<td></td>
<td>Picard order</td>
<td>-</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>Picard iterations</td>
<td>-</td>
<td>400</td>
</tr>
<tr>
<td></td>
<td>total time</td>
<td>meters</td>
<td>0.001</td>
</tr>
<tr>
<td>PHAD</td>
<td>time step size</td>
<td>meters</td>
<td>0.001</td>
</tr>
<tr>
<td></td>
<td>Picard order</td>
<td>-</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>Picard iterations</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>FMM order</td>
<td>-</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>total time</td>
<td>meters</td>
<td>0.001</td>
</tr>
</tbody>
</table>

PHAD code with the $N$-body code (Fig. 5.7). Table 5.4 shows the parameters used for this test. As in the previous test, the relative errors of $x$ coordinates of all particles measured after 2 meters (total time) are shown in Fig. 5.8. The maximum relative error is $\sim 10^{-7}$, and it is larger than the previous test with the small time step size. Therefore, we reiterate
that the PHAD and the $N$-body codes converge to each other if the time step size is small and the FMM order is high.

Table 5.4: Parameter used in the $N$-body code and the PHAD code for the large time step size, 2 meters.

<table>
<thead>
<tr>
<th>Code</th>
<th>Parameter</th>
<th>Units</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$-body code</td>
<td>time step size</td>
<td>meters</td>
<td>0.005</td>
</tr>
<tr>
<td></td>
<td>Picard order</td>
<td>-</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>Picard iterations</td>
<td>-</td>
<td>400</td>
</tr>
<tr>
<td></td>
<td>total time</td>
<td>meters</td>
<td>2</td>
</tr>
<tr>
<td>PHAD</td>
<td>time step size</td>
<td>meters</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Picard order</td>
<td>-</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>Picard iterations</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>FMM order</td>
<td>-</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>total time</td>
<td>meters</td>
<td>2</td>
</tr>
</tbody>
</table>
a. $z - x$ plot  

Figure 5.7: $z - x$ plot and momentum in the $x - y$ plane ($p_x - p_y$) for the $N$-body code and the PHAD. All points in two plots overlap indicating that the results are in good agreement. The time step size is 2 meters.

b. $p_x - p_y$ plot

Figure 5.8: Relative errors of $x$ values in the PHAD and the $N$-body code. The time step size is large (2 meters) and the relative errors are larger compared to the small time step size test.
5.4 MPI Based Parallelization of PHAD

The PHAD is a dynamic simulation code written in C++ and COSY Infinity. To reach the desired cooling, the proton beam has to make many revolutions together with the electron beam in the cooling sections. The time step size cannot be made arbitrarily large in favor of expediting the revolution rate since larger time steps will mask important and highly probable physics such as close encounters of particles. On the other hand, many small time steps are required to obtain the desired results. Serial execution of the code cannot reach the number of revolutions required for cooling the proton or the ion beam within a reasonable time period. To meet the compelling requirements, the parallel version of PHAD has been developed. The main parts of parallel programming are distributing the task/data among different nodes/processors, synchronization, communication between nodes/processors, and finally gathering the final results.

As shown in Table 5.5, the data structuring part of the code is written in C++, and it takes a small fraction of the total run time (Fig. 5.9). The FMM and the Picard iteration-based integrator, which are written in COSY INFINITY and take long runtime, demand parallel implementations. COSY INFINITY 9.1 supports MPI based parallelization [46]. MPI is one of the popular models for parallel computing. COSY INFINITY implements a super set of the usual MPI commands (such as PLOOP...END PLOOP) that make MPI similar to OPEN-MP parallel loop constructions, but much simpler to use due to its integration with COSYScript. Using MPI, we can generate precise and fast output in simulation.

In COSY INFINITY, we can dynamically determine the available number of processor for the application by the PNPRO M statement, where M is an integer variable and PNPRO is a COSY INFINITY function. If we assume that we have N number of MPI processes (For example, in our data structure number of Trees or D-Tree nodes) to process, then we can
find the number of MPI processes per processor given by $\frac{N}{M}$. $N$ may not always be exactly divisible by $M$, and if $N > M$, then we allocate the remaining number of processes ($N - \left\lfloor \frac{N}{M} \right\rfloor$) to each processor starting from the processor number one. We iterate each process through COSY INFINITY main parallel processing construct `PLOOP-ENDPLOOP`. Syntax for `PLOOP` is `PLOOP m n`, where $m$ and $n$ are integers and $m$ is the starting process number and $n$ is ending process number. In our application, we created a COSY function `GET LOOP LIMITS` to identify $m$ and $n$ values. Syntax of `ENDPLOOP` is `ENDPLOOP c X`, where $c$ is an integer variable and can have values 1 through 8 and $X$ is a multidimensional array and its last column must have length $M$. Data saved in multidimensional array of $X$ can be used to share among processes or subsequent processing in the application. If there are $Q$ groups ($Q = \frac{N}{M}$) of MPI processes executing in the parallel loop block, there are $Q$ root processes associated with $Q$ different groups. For example 1, 6 and 11 are root processes in Fig. 5.10. `PLOOP-ENDPLOOP` exits after all MPI processes are completed. Therefore, we can use it as an application flow controlling mechanism. The COSY INFINITY intrinsic procedure `PROOT` is used to identify the root process in `PLOOP-ENDPLOOP` block. We execute IO operation or call the external application (C++ application) to the COSY program if the process is the root process.

Table 5.5: Time taken by each part of the FMM.

<table>
<thead>
<tr>
<th>Component</th>
<th>Time (sec)</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>data structuring</td>
<td>0.291</td>
<td>1.2</td>
</tr>
<tr>
<td>fmm up</td>
<td>0.578</td>
<td>2.4</td>
</tr>
<tr>
<td>fmm down</td>
<td>10.144</td>
<td>41.6</td>
</tr>
<tr>
<td>fmm direct</td>
<td>13.059</td>
<td>53.5</td>
</tr>
</tbody>
</table>
The loop length $M$ denotes the number of processors. If there are $N$ processes in the execution of the parallelized COSY program, we find $Q = \left\lfloor \frac{N}{M} \right\rfloor$, which defines the number of processes that must be allocated to each processor. Since $(N - M \cdot Q) \neq 0$, then $(N - M \cdot Q)$ processes can be allocated among $M$ processors [46]. For instance, if $N = 14$ and $M = 3$, then the first two processors get 5 processes and the third one gets 4 processes as shown in Fig. 5.10.

$N = 14$

1 2 3 4 5 6 7 8 9 10 11 12 13 14.
**PNPRO** is the procedure that allows the program to identify the number of processes we intend to use. **PROOT** procedure is used to identify the root process, and it returns 1 for the root process and 0 for the non-root processes [46].

For instance,

```plaintext
PROCEDURE GET_MPI_INFO M ip ipr;
VARIABLE I 1;
PNPRO M;
    PLOOP I 1 M;
    ip := I;
    ENDPLOOP;
PROOT ipr;
ENDPROCEDURE;
```

### 5.4.1 Parallel Implementation

The C-forest trees in the data structure can be treated as independent entities. Each tree has a certain number of nodes, and that number differs from tree to tree and hence the size of memory varies. The trees can be sorted according to the number of nodes or memory each
tree acquires. Since the number of available processors is specified before execution, dynamic load balancing is achieved by distributing the trees approximately evenly among the MPI processes. The MPI processes are mapped as one process per each machine processor. Once all processes finish their tasks, the final outcome is obtained by gathering the individual results. All processes must begin the next split task simultaneously. Hence, it is crucial to properly maintain the communication among processes. In PHAD, in order to prevent asynchronous advancement, we place a barrier between two consecutive tasks. For instance, we place a barrier between the upward pass and the downward pass. Each process halts at the barrier waiting for all other processes to finish the upward pass and then resumes the downward pass together. This barrier is also another PLOOP command. Similarly, the $D$-tree nodes can be divided among the MPI processes.

\begin{verbatim}
PROCEDURE BARRIER;
    VARIABLE X 1 M;
    VARIABLE I 1;
    VARIABLE Y 1;
    PLOOP I 1 M;
        Y := 1;
    ENDPLOOP 1 X;
ENDPROCEDURE;
\end{verbatim}

When the PHAD code runs, it first calls the C++ program for data structuring. In order to prevent asynchronous communication, we designate the root process to call the C++ program with the specified parameters such as the source/target file name, the particle limit $q$, the relativistic gamma value, number of processes and the process ID. Therefore, we use the following PLOOP, and it guarantees that only the root process is allowed to call the C++ program.
PLOOP I 1 M;
    PROOT prank;
    IF (prank = 1);
        CALL THE C++ EXECUTABLE
    ENDIF;
ENDPLOOP 1 ploopControl;

All other processors need to wait until the root process finishes retrieving the information in the C++ program. Therefore, we perform a status checking as shown below.

fmmstatus := 0;
WHILE (fmmstatus = 0);
    PLOOP I 1 M;
        PROOT prank;
            IF (prank = 1);
                OPENF outfile 'fmmcpp.status' 'UNKNOWN';
                READ outfile fmmstatus;
                CLOSEF outfile;
            ENDIF;
        ENDPLOOP 1 ploopControl;
    ENDPLOOP 1 ploopControl;
ENDWHILE;

We implement parallelization in Gaea, a high performance 60-node computing cluster at Northern Illinois University (NIU) (more details in Section 5.4.5). We extensively used this system for our simulations.

According to the preliminary tests done to identify the best MPI configuration, Table 5.6 shows that a multi-processor simulation incurs a significant efficiency loss due to collective communication calls among processors. Therefore, it is desirable to use as many nodes and a single processor from each node. For example, a simulation with 8 nodes with 1 processor
Figure 5.11: Time taken for 1000 time steps by the MPI-based parallel version with different number of nodes.

from each node finishes faster than a single node with 8 processors. The test results show that

Table 5.6: Runtime measured for 1000 time steps by varying the number of nodes and the processors per node (ppn).

<table>
<thead>
<tr>
<th>Test</th>
<th>Nodes</th>
<th>ppn</th>
<th>MPI Procs</th>
<th>Time (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>262</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>136</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>79</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>1</td>
<td>8</td>
<td>56</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>1</td>
<td>12</td>
<td>55</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>2</td>
<td>8</td>
<td>94</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td>174</td>
</tr>
</tbody>
</table>

inter-node communication, which is the communication between two processors on different nodes, is the best suited for our problem (Fig. 5.11).
5.4.2 Parallelizing Data Structuring with Intel Cilk Plus

Intel Cilk Plus is implemented by the Intel C++ Compiler [47]. It is known as an extension of C or C++ and will be useful when the application runs on a multi-core machine. Cilk supports multi-threads and shared memory. Every thread can globally access all memory. Since Cilk shares memory, a computer node with multi-processors or a work station with a multi-core processor is an appropriate environment for a Cilk application to run.

We developed the parallel version of the data structuring part written in C++ using Intel Cilk Plus. However, as shown in Fig. 5.9, the time taken to build the data structure is about $\sim 1\%$ of the total runtime (Table. 5.5). If the data set becomes large, the parallel version will reduce the runtime considerably.

The word `cilk` identifies the cilk procedure [48]. Intel Cilk Plus introduces the following three keywords to the C++ code.

**cilk_spawn**
This keyword invokes the parallel task. In high degree parallelism, the first cilk procedure (parent) can spawn off many other procedures (children).

**cilk_sync**
The use of the keyword forces all tasks to finish their previous job before starting the next job. In other words, if all children have not completed their tasks when the program executes `cilk_sync`, the procedure stops and waits until all of its children have completed. Essentially, `cilk_sync` is a local barrier. In `main.cpp`, we have used these keyword as follows.

```cpp
    cilk_spawn sortAndCalcParents(source_particles, sources_boxnum_table);
    sortAndCalcParents(target_particles, targets_boxnum_table);
    cilk_sync;
```
cilk_for

This converts a serial for loop into a parallel for loop. In main.cpp and particle.cpp we have converted several regular for loops into parallel version. For example,

cilk_for (uint32_t i = 0; i < 8; i++) {
   subDivide children[i]
}

5.4.3 Parallelizable Procedures in the PHAD

Picard Integrator

In the Picard integrator point-to-point interactions of particles in each $D$-leaf node are calculated. Since each node has its unique neighborhood, for a given time step or a temporal resolution $\Delta t$, we can evenly distribute $D$-leaf nodes ($D_{\text{leafNodes}}$) among all processors. Accordingly, calculations involving $D_{\text{leafNodes}}$ can be independently performed.

FMM Up

In the $C$-forest, there are many trees and they are independent. Therefore, we can evenly distribute the trees among processors and PLOOP block can be used to process them. In the PROCEDURE FMMUP we calculate the multipole expansions based on the information provided by the trees in the $C$-forest.

FMM Down

Multipole-to-local expansion must be performed for each node in the $D$-tree. Since we know the total number of $D$-tree nodes beforehand, they can be distributed among the processors.
In the **PROCEDURE MODLNGRNGINT**, the neighborhood of each $D$-leaf node needs to be organized after each Picard iteration. Hence, these independent $D$-leaf nodes can be assigned to various processors.

Table 5.6 shows that the parallel version of the PHAD (Test 4) gives an almost five-fold speed up over the baseline (Test 1).

### 5.4.4 Serial Procedures in the PHAD

**FMM DIRECT** and long range integrator (**LNGRNGINT**) procedures in the PHAD still performs serial executions. The time taken by each procedure for a single time step in the serial version of the PHAD code was measured. Table 5.7 shows that the **FMM DIRECT** and the long range integrator take the least amount of time. MPI parallelization incurs inter-process communication latency, and it will be added to the processing time. Therefore, we excluded **FMM DIRECT** and long range integrator from parallelization.

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Time (s)</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmm up</td>
<td>$7.70 \times 10^{-2}$</td>
<td>0.57</td>
</tr>
<tr>
<td>fmm down</td>
<td>0.71</td>
<td>5.21</td>
</tr>
<tr>
<td>fmm direct</td>
<td>$6.0 \times 10^{-2}$</td>
<td>0.44</td>
</tr>
<tr>
<td>Picard integrator</td>
<td>12.464</td>
<td>91.79</td>
</tr>
<tr>
<td>long range integrator</td>
<td>$4.0 \times 10^{-3}$</td>
<td>0.03</td>
</tr>
<tr>
<td>modified long range integrator</td>
<td>0.26</td>
<td>1.96</td>
</tr>
</tbody>
</table>
5.4.5 Cluster Information

*Gaea* is the NIU’s 60-node CPU and GPU hybrid cluster (Fig. 5.12). These 60 nodes are connected via Full 1:1 non-blocking InfiniBand and Ethernet switch connectors. The configuration of individual nodes in Gaea are shown in Table 5.8.

![Figure 5.12: NIU computing facility, Gaea.](image)

The serial version of the code is run on the ARC cluster at NIU. ARC is a virtualized, or cloud, compute cluster (Table 5.9). The cluster has two compute nodes with a total of 1TB RAM, 48 CPU cores (about 260 GHz compute power) and 20 TB raw storage all connected by twin bonded 10 GB network links across a low latency pair of 10GB switches.
Table 5.8: Gaea Cluster’s Compute Node Specification.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server</td>
<td>HP ProLiant SL390 Generation 7 (G7)</td>
</tr>
<tr>
<td>Processor</td>
<td>6-core Intel Xeon X5650 w/ HyperThreading @ 2.66 GHz</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>32KB I-Cache, 32KB D-Cache</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>256KB</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>12MB Shared</td>
</tr>
<tr>
<td>DRAM</td>
<td>72 GB</td>
</tr>
<tr>
<td>Storage</td>
<td>4 x 500 GB 2.5 SATA (i.e., 2 TB each node)</td>
</tr>
<tr>
<td>GPU</td>
<td>2 x NVIDIA M2070 FERMI (each with 6GB RAM) and connected via PCIe</td>
</tr>
</tbody>
</table>

Table 5.9: ARC Cluster’s Compute Node Specification.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server</td>
<td>Virtual server on VMware</td>
</tr>
<tr>
<td>Processor</td>
<td>6-core Intel Xeon E5-2697 w/ HyperThreading @ 2.70 GHz</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>32KB I-Cache, 32KB D-Cache</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>256KB</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>30MB Shared</td>
</tr>
<tr>
<td>DRAM</td>
<td>72 GB</td>
</tr>
</tbody>
</table>
CHAPTER 6
SIMULATION DETAILS, RESULTS AND APPLICATIONS

6.1 Energy Recovery Linac in JLEIC

The Jefferson Lab Electron Ion Collider (JLEIC) is a billion-dollar project approved by the federal Nuclear Science Advisory Committee (NSAC) as the highest priority new construction for nuclear physics after the Facility for Rare Isotope Beams (FRIB) is finished [49]. Electron cooling is a single point of failure of the JLEIC. In other words, if the uncertainty is off by a factor of 2, the entire design will be unsuccessful. Most often computer models are used to simulate experiments before performing highly expensive physical experiments. Therefore, simulations results must be highly accurate. As shown in Fig. 6.1, the JLEIC consists of two figure-8 rings: one for protons and the other for electrons [2, 50]. The objective of this research is to simulate electron cooling of heavy ion beams in the straight cooling sections housed in the booster and the collider ring as shown in Fig. 6.2.

Figure 6.1: The ion booster, the ion collider ring and the electron collider ring of the JLEIC.
6.2 Particle-based Proof of Principle Numerical Simulation of Electron Cooling

6.2.1 Parameter Estimation

As we have mentioned earlier, our simulation employs real particles—electrons and protons or heavy ions. In a straight section of the ring (booster or collider as shown in Fig. 6.2), the proton beam is brought into overlap with an electron beam, which propagates with the same velocity of protons. After energy boosting (~8GeV), the high energy protons are injected into the large collider ring. The length of the cooling sections in each ring is different. Given the strength of the solenoidal magnetic field, it is necessary to estimate the correct time step size in each section to match with the energy of ions and the length of the cooling section. In the booster ring, the length of the cooling section is 3m, and in the collider ring it is 60m. In order to estimate the time step size for each cooling section, the length of the cooling section is needed.

Figure 6.2: Two cooling sections - one in the booster ring for DC cooling and the other in the collider ring for BB (bunched beam) cooling.
6.2.1.1  Booster Ring

In the booster ring, the energy of protons at the injection is 280 MeV, and the corresponding average velocity of them is 0.64c. c is the speed of light. In order to co-propagate with the proton beam, the electrons must have the same average longitudinal velocity as protons. In the conceptual design, the length of the cooling section in the booster ring is given as 3 m. Accordingly, electrons moving at the speed of 0.64c take 3.6 ns to traverse the cooling section. Typically, the cooling section is immersed in a longitudinal magnetic field \( B \) of 1 T (Fig. 3.22), and it has a significant affect on collision dynamics of electrons and ions. The electrons execute a helical motion (Fig. 6.3) in this magnetic field and the time taken for one revolution or the period can be given by

\[
T = \frac{2\pi m}{qB},
\]

where \( m \) and \( q \) denote the mass and the charge of an electron, respectively. Hence, the period is \( T = 35.77 \times 10^{-12} \) s. Each electron, approximately makes 100 revolutions under the influence of the magnetic field before exiting the cooling section. It is customary to assume that electrons make 10 small time steps to complete one rotation. Then a single electron must complete 1000 time steps of size 3.6 ps in one pass in the cooling section. In the booster ring the ion or the proton beam must be cooled within 16 ms. Therefore, the proton and the electron beam can pass the cooling section approximately about \( 10^6 \) times.

6.2.1.2  Collider Ring

The collider ring receives the accelerated protons of energy of 100 GeV/u from the booster ring. Hence, we need a long cooling section in the collider ring of about 60 m immersed in
Figure 6.3: The schematic layout of the cooling section in the pre-booster is 3 m long. Protons and electrons enter the cooling section and co-move in the longitudinal direction with velocity $\vec{v}_z = 0.64c$. The strength of the applied magnetic field in the longitudinal direction $B_z = 1$ T.

A longitudinal solenoidal magnetic field of 1 T. As in the booster ring, we can perform a similar calculation for the collider ring to decide the appropriate time step size. It can be shown that each electron, approximately makes 5,592 revolutions before exiting the cooling section in the collider ring. Also, if it takes 10 small time steps to complete one rotation, then a single electron must complete 55,920 time steps of size 3.6 ps in one pass in the cooling section. According to our estimation, the proton and the electron beam must pass the cooling section in the collider ring approximately about $10^8$ times.
6.2.1.3 Simulation Details

To use as the input data for our simulation, a set of particles with space coordinates characterized by a Gaussian distribution is randomly generated. The maximum number of particles that can be used in the simulation is limited by the large runtime. Hence, we used 100 protons and 1000 electrons. We noticed, in our preliminary studies, that the cooling rate is very slow with regular electrons. In order to enhance the cooling rate, the electron bunch intensity must be increased (Eq. 2.31). Therefore, the regular electrons were replaced by macro-electrons (Table 6.1). We performed cooling in 3D, where $x$ and $y$ in the transverse direction and $z$ in the longitudinal direction or in the direction of the beam propagation. Both protons and electrons lie on a disk in the $x - y$ plane with a radius of 0.01 m and 0.005 m (more details are given in 6.2.9), respectively, as shown in Fig 6.8. The bunch length is 1 mm since the JLEIC design requires very short bunches for both electron and ion beams [2]. The other beam parameters used in the simulation are listed in Table 6.1.

6.2.2 Selection of Optimum Parameters

There are several parameters need to be optimized before launching the simulation. Among them are the time step size for the Picard Integrator, the Picard order, the FMM order, and the particle limit $q$. The first step of simulation is choosing the optimum values for the parameters. Therefore, we performed numerous tests and investigated them to identify the optimum value for each parameter.
Table 6.1: Parameters used in the simulation.

<table>
<thead>
<tr>
<th>Element</th>
<th>Parameter</th>
<th>Unit</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>proton</td>
<td>x-y radius</td>
<td>m</td>
<td>0.01</td>
</tr>
<tr>
<td>proton</td>
<td>mass</td>
<td>amu</td>
<td>1</td>
</tr>
<tr>
<td>proton</td>
<td>charge</td>
<td>e</td>
<td>1</td>
</tr>
<tr>
<td>proton</td>
<td>longitudinal momentum $p_z$</td>
<td>scaled with $\frac{1}{mc}$</td>
<td>0.83</td>
</tr>
<tr>
<td>proton</td>
<td>horizontal momentum $p_x$</td>
<td>scaled with $\frac{1}{mc}$</td>
<td>%0.1of $p_z$</td>
</tr>
<tr>
<td>proton</td>
<td>vertical momentum $p_y$</td>
<td>scaled with $\frac{1}{mc}$</td>
<td>%0.1of $p_z$</td>
</tr>
<tr>
<td>macro-electron</td>
<td>x-y radius</td>
<td>m</td>
<td>0.005</td>
</tr>
<tr>
<td>macro-electron</td>
<td>mass</td>
<td>amu</td>
<td>17.73</td>
</tr>
<tr>
<td>macro-electron</td>
<td>charge</td>
<td>e</td>
<td>-32552.08</td>
</tr>
<tr>
<td>macro-electron</td>
<td>longitudinal momentum $p_z$</td>
<td>scaled with $\frac{1}{mc}$</td>
<td>14.73</td>
</tr>
<tr>
<td>macro-electron</td>
<td>horizontal momentum $p_x$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>macro-electron</td>
<td>vertical momentum $p_y$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>macro-electron</td>
<td>current</td>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>bunch length</td>
<td></td>
<td>mm</td>
<td>1</td>
</tr>
<tr>
<td>longitudinal magnetic field $B_z$</td>
<td></td>
<td>T</td>
<td>1</td>
</tr>
<tr>
<td>time step size</td>
<td>$h$</td>
<td>meters</td>
<td>0.001</td>
</tr>
<tr>
<td>number of Picard iterations</td>
<td>$niter$</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Picard time step</td>
<td>$\Delta t = \frac{h}{niter}$</td>
<td>meters</td>
<td>0.001</td>
</tr>
</tbody>
</table>

### 6.2.3 Particle Limit $q$

The runtime for 100 time steps was measured by varying $q$. The optimum $q$ value for Gaussian-distributed 1100 particles was found to be 400 as shown in Fig. 6.4.

### 6.2.4 Time Step For the Picard Integrator

We set the time step size to 0.001 meters (here we use scaled time as shown in 4.3.1 and hence the unit in meters), which is the time step we calculated for the booster ring. However, if the time step is bigger than 0.001 meters, we get some unphysical behavior in the emittance plot. For instance, to speed up the simulations we tested a simulation with time step size...
Figure 6.4: The optimum value for the $q$ (particle limit) value is 400, since the runtime is minimum at $q=400$.

increased to 0.004 meters. According to Fig. 6.5, the calculated transverse emittances of protons showed a rapid increase. Plots of $x - y$ cross sections of the proton beam as well as the electron beam indicated some outliers and the noticed rapid emittance growth can be attributed to these outliers. By further investigation, we found that two protons (26 and 54) are very far from the rest of the protons as shown in Fig. 6.7. This reflects as the emittance blow up in the emittance plot. This problem was avoided by reducing the time step size to 0.001 meters. In all simulations, we used the same time step size, 0.001 meters, to avoid such issues. In addition, close encounters of particles may cause unphysical behavior of the proton beam. Therefore, with 100 particles of unit mass and different charges, we performed another test to investigate the close encounter of particles [45]. We used 100 equal time steps within a total time span of 5. As time progresses, the distance to all particles were measured from particle 1 (the reference particle). Two particles, 58 and 77, showed a atypical behavior in the distance against the time step plot. The distance plot for particle 57 (typical) and 58 (atypical) is shown in Fig. 6.6.a. The distance plot for 57 and 77 is similar
Figure 6.5: Emittances of proton with the time step size=0.004 meters.

to this. All particles, except for 58 and 77, show a behavior similar to particle 57. Due to this close encounter they experience a large force and the distance between particle 58 and 77 instantly increases at the time step 49 (Fig. 6.6.b). This type of behavior can adversely affect on the emittance calculation of the proton beam.

6.2.5 Picard Order

If we add more higher-order Picard terms, the solution of the differential equation will converge to the exact solution. Similarly, as the number of Picard iterations increases, the solution approaches the exact solution. However, since this simulation requires millions of time steps, we cannot afford for higher order Picard terms or large number of Picard iterations. Hence, it is crucial to find the optimum values for these two parameters. The
(a) Distance of particle 57 (typical) and 58 (atypical) with respect to particle 1.

(b) Distance between particle 77 and 58. The distance between them suddenly increases.

Figure 6.6: Close encounter of particles at the step 49.

Runtime for a single time step with a single Picard iteration was measured by varying the Picard order. According to Table 6.2, it is apparent that a single Picard iteration with Picard order 7 has the shortest time. The difference between the results obtained in the Picard order 16 and 7 is in the order of $10^{-19}$. Therefore, we conveniently reached the conclusion that a single Picard iteration with the Picard order 7 is sufficient to solve the differential equations without forsaking accuracy.

Table 6.2: The runtime for different FMM orders and Picard orders.

<table>
<thead>
<tr>
<th>Number of time steps</th>
<th>Time step size (meters)</th>
<th>FMM order</th>
<th>Picard order</th>
<th>Time(min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.001</td>
<td>9</td>
<td>16</td>
<td>1.14</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>16</td>
<td>1.07</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>12</td>
<td>0.52</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>10</td>
<td>0.36</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>7</td>
<td>0.24</td>
</tr>
</tbody>
</table>
6.2.6 **FMM Order**

The impact of the FMM order on runtime was discussed in section 3.2.5.2. By measuring the runtime for 100 time steps with different FMM orders, the desirable value for the FMM order is found to be 5 as shown in Table 6.2.

6.2.7 **Simulation 1: Proton Beam with Small Transverse Emittances**

In the first simulation we used regular electrons to cool the proton beam. The proton beam did not show any noticeable emittance reduction. Therefore, we replaced the regular electrons with macro-electrons. For this simulation, we used a proton beam with extremely
small initial transverse momentum, and the ratio between the longitudinal and transverse momentum, $\frac{p_{x,y}}{p_z}$ was $\approx 10^{-6}$.

### 6.2.7.1 Emittance Calculation

The Eq. 2.2 was used to calculate the emittance.

$$
\epsilon_{x,\text{rms}} = \sqrt{\langle x^2 \rangle \langle x'^2 \rangle - \langle xx' \rangle^2}.
$$

As previously explained, $x$ is the position and $x' = \frac{p_x}{p_z}$. $p_x$ and $p_z$ are the horizontal and longitudinal momentum, respectively. Initially, transverse momenta of protons are non-zero while those of electrons are zero. When electrons propagate in the longitudinal direction with the same average velocity, as a result of Coulomb collision, electrons gain momenta in the transverse direction. Therefore, transverse momenta of protons gradually decrease. In
other words, the proton beam gets cooled. We measured the position and the momentum of each proton and electron after each time step and calculated the emittance using Eq. 2.2. The horizontal and vertical emittance plots for protons are given by Fig. 6.9 and Fig. 6.10. Both plots indicate that the transverse emittance of protons diminishes very slowly.

Figure 6.9: Emittance ($\epsilon_x$) of protons after $10^6$ time steps. The dashed black line is a reference line. The red line is the trend line to represent the behavior of horizontal emittance.
6.2.8 Problems Discovered and Remedies

6.2.8.1 Electron Bunch Refreshing

A noteworthy issue that arose during the simulation is the spread of electrons in the longitudinal direction causing a bunch lengthening. As a consequence of this, degradation of intensity occurs over time. In order to correct this, we refreshed the electron bunch after every 3 m by replacing the existing electron bunch with a new batch of ‘cold’ electrons. In our simulation, 3 m length is equivalent to 4700 time steps with time step size 0.001 meters.
6.2.8.2 **Softening Parameter**

We have applied minimal softening. The softening parameter $\lambda = 0.5 \times 10^{-4}$ m, which is compatible with the configuration data and the average distance between particles, was introduced as a precaution such that any close encounter (discussed in 6.2.4) that would require smaller time steps than the already very small time step size imposed by the strong magnetic field on electrons do not blow up the simulations. As a result of this, outliers were not detected in the $x - y$ plot of protons.

We introduced the electron refreshing and the softening parameter in all subsequent simulations.

6.2.9 **Different Initial Particle Configurations**

In order to identify the most appropriate proton configuration to detect fast cooling, we performed three tests—protons and electrons are evenly distributed in a disk of radius 1 cm, electron beam is surrounded by the proton beam of 1 cm radius, proton beam is surrounded by the electron beam of 1 cm radius. The following are the simulation details.

6.2.9.1 **Protons and Electrons are Evenly Distributed**

The configuration of protons was changed such that all particles are evenly distributed on the disk of radius 1 cm in the $x - y$ plane (Fig. 6.11). The bunch length is 1 mm. The calculated horizontal and vertical emittances are shown in Fig. 6.12. With respect to the dashed horizontal line in black, we can see the horizontal emittance decreases (peaks of the
Figure 6.11: Cross section of the proton and the electron beam. Both types of particles are uniformly distributed on a disk of 1cm radius in the $x - y$ plane. 100 protons are in red and 1000 electrons are in blue.

Figure 6.12: Evenly distributed protons and electrons. Emittances of protons at the time step 132,000. The green and black dashed lines are reference lines, for $x$ and $y$ emittance values.

red curve) with time. Similarly, with respect to the dashed horizontal line in green, the vertical emittance (peaks of the blue curve) decreases with time.
In order to further examine the behavior of emittances, we can study the trend of maxima of each curve as shown in Fig. 6.13. These two plots show that $x$ maxima as well as $y$ maxima decline with time assuring the transverse emittance reduction.

6.2.9.2 Proton Spread in 0.5 cm Radius

We changed the configuration to limit the proton spread in a disk of radius 0.5 cm while electron spread is in a disk of radius 1 cm. The bunch length is 1 mm. The emittance plot for protons did not show emittance reduction.

6.2.9.3 Proton Spread in 1 cm Radius

In the next test, we changed the configuration to limit the electron spread in a disk of radius 0.5 cm while proton spread in a disk of radius 1 cm. Again, the bunch length is 1 mm. With this configuration, the emittance plots were similar to those of the evenly distributed particle configuration. Therefore, we continued our simulations with this configuration.
### 6.2.10 Simulation 2: Protons with 0.1% Momentum

In this simulation the transverse momentum of protons was increased to 0.1% of longitudinal momentum. Therefore, $\frac{p_{x,y}}{p_z} \approx 10^{-3}$. Also, protons are spread in a disk of radius 1 cm in the $x-y$ plane while electrons are confined to a radius of 0.5 cm. The $x-y$ cross section and the bunch length are similar to Fig. 6.8. Also, we employed the electron bunch refreshing after every 4700 time steps and a softening parameter of $0.5 \times 10^{-4}$ m. The strength of the longitudinal (solenoidal) magnetic field is 1 T.

![Graphs showing transverse electron emittances](image)

Figure 6.14: Emittances of electrons at the time step 100,000 (with softening and electron bunch refreshing). The longitudinal magnetic field is 1 T.

The transverse electron emittances are shown in Fig. 6.14. The dips at every integer multiple of 4700 time steps correspond to electron bunch refreshing. The transverse emittance of protons during the first $10^5$ time steps is shown in Fig. 6.15. This plot indicates a small emittance reduction in both transverse directions.

There are some other parameters that we can optimize to detect a fast cooling process. Therefore, we performed simulations with different solenoidal magnetic fields.
According to the Eq. 6.1, the time step size depends on the strength of the solenoidal magnetic field described in 6.2.1.1. Therefore, two tests were performed with low and high magnetic fields. For the low magnetic field case, it was reduced to 0.8 T. However, the proton emittances did not show a noticeable change in the cooling rate. In the next test, the magnetic field was increased to 1.5 T, and the emittance curves for protons showed a slight decrease. Therefore, we proceeded with the high magnetic field of 1.5 T (Table 6.3).
6.2.12 Simulation 3: Proton Beam with 0.1% Transverse Emittances and High Magnetic field

In this simulation the transverse momentum of protons was increased to 0.1% of longitudinal momentum. Therefore, \( \frac{p_{x,y}}{p_z} \approx 10^{-3} \). Also, protons are spread in a disk of radius 1cm in the \( x-y \) plane while electrons are confined to a radius of 0.5 cm. The \( x-y \) cross section and the bunch length are similar to Fig. 6.8. Also, we employed the electron bunch refreshing after every 4700 time steps and a softening parameter of \( 0.5 \times 10^{-4} \) m. The strength of the longitudinal (solenoidal) magnetic field is 1.5 T.

Table 6.3: Parameters used for Simulation 3.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>unit</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>proton beam radius</td>
<td>m</td>
<td>0.01</td>
</tr>
<tr>
<td>electron beam radius</td>
<td>m</td>
<td>0.005</td>
</tr>
<tr>
<td>bunch length</td>
<td>mm</td>
<td>1</td>
</tr>
<tr>
<td>magnetic field</td>
<td>T</td>
<td>1.5</td>
</tr>
<tr>
<td>time step size</td>
<td>meters</td>
<td>0.001</td>
</tr>
<tr>
<td>horizontal momentum ( p_x )</td>
<td>scaled with ( \frac{1}{mc} )</td>
<td>%0.1 of ( p_z )</td>
</tr>
<tr>
<td>vertical momentum ( p_y )</td>
<td>scaled with ( \frac{1}{mc} )</td>
<td>%0.1 of ( p_z )</td>
</tr>
</tbody>
</table>

The proton emittances for 1,005,000 time steps are shown in Fig. 6.16. A small emittance reduction in both transverse directions can be observed. As we explained in a previous simulation, the peak emittance values decrease with time as depicted in Fig. 6.17. The results of this simulation were generated by the MPI-parallel PHAD code.

Most of our simulations were performed for only a small fraction (< 1%) of the expected cooling time. In real time, these runs took significant amounts of time with the serial version of PHAD. We can conclude that the fully parallelized version of PHAD will be needed for successful complete electron cooling demonstrations. As described earlier, we
Figure 6.16: Emittances of protons for 1,005,000 time steps. (with softening and electron bunch refreshing). The longitudinal magnetic field is 1.5 T.

Figure 6.17: Maxima of the vertical and the horizontal emittance plots. The negative slope on both plots indicates that the transverse emittance decreases with time.

already implemented several parallel modules. However, to achieve fully parallel, scalable performance it is essential to implement GPU-based hybrid parallelization of the FMM, adaptive time steppin, and automatic order selection in the Picard iteration based integrator.
CHAPTER 7
CONCLUSIONS

Even with high-performance computers, solving the $N$-body problem becomes impractical for very large $N$. There are some methods available to solve the $N$-body problem. However, the caveat lies in maintaining high accuracy. The fast multipole method yields high efficiency as well as high accuracy. In order to harness the advantage of the FMM, it must be tailored to suit the particle configuration. The novel FMM algorithm—adaptive multi-level FMM—implemented in this dissertation is efficient and accurate. It is applicable for any complex particle distribution in 3D or 2D, and it is platform independent. The two major parts of the FMM code, data structuring and potential calculation, were implemented in C++ and COSY INFINITY 9.1, respectively. Since any charged particle beam in accelerators typically contains a large number of particles ($\sim 10^{11} - 10^{13}$), the adaptive multi-level FMM can be used to calculate the Coulomb interactions among the particles with appreciable accuracy and efficiency.

The FMM code was well tested and measured the potential and field with different types of particle distributions of different sizes. The FMM results were compared with the point-to-point (or direct) method results and verified the accuracy. Another notable advantage of the FMM is that it has guaranteed apriori error bounds.

Studying the time evolution of particle distributions and their emittances is crucial in accelerator and beam physics. Therefore, we developed a new high performance computer code (PHAD). In addition to the FMM, it includes the Picard iteration-based integrator and the Strang operator splitting method. The Picard integrator resolves the close encounters. Strang splitting method enhances the efficiency while maintaining high accuracy and pre-
serving symplecticity. The PHAD code can be utilized to understand and study the behavior of the particle beam’s temporal evolution.

In order to ascertain the correctness of the PHAD code, the results were compared with the $N$-body code, which uses the conventional point-to-point method to calculate the potential/field of charged particles. The $N$-body code employs the Picard integrator to solve the differential equations. However, it does not use the FMM. This $N$-body code was benchmarked against the point-to-point method. The PHAD results and the $N$-body results agree excellently. Therefore, the PHAD code assures its accuracy by benchmarking against the $N$-body code.

Our initial implementation of the serial version of the PHAD code takes a long time to show any signs of the cooling effect. The beam propagated distance depends on the time step size. In order to speedup the beam propagation, we cannot increase the time step size arbitrarily because it is parameter-dependent. Therefore, a viable solution to overcome the long runtime issue is developing the parallel version of the PHAD code. As COSY INFINITY 9.1 supports MPI based parallelization, we developed the MPI-parallel PHAD code. With this parallel version, we investigated the long term behavior of the beam propagation. The results of the MPI version were compared with the serial version, and we confirmed that they are in good agreement.

The current version of MPI-parallel PHAD, compared to the serial version, yields a speedup of factor 4-5 (For example, 1000 time steps need 248 minutes in the serial version and 55 minutes in the parallel version). We expect a factor of $\sim 100$ times the performance increase by full parallelization of this code.

With the current version of PHAD, we performed the first ever particle-based proof of principle simulation of electron cooling of protons. The long cooling time and stringent accuracy requirements have limited us to simulate a small fraction of the cooling time to date. Nevertheless, this seems sufficient to show the first signs of emittance reduction, which
is promising regarding the future: The complete cooling simulations with the upgraded, fully parallel PHAD as described in this dissertation are still ongoing.
BIBLIOGRAPHY


[47] [https://www.cilkplus.org/](https://www.cilkplus.org/).


[50] [https://indico.hep.anl.gov/indico/getFile.py/access?contribId=6&sessionId=0&resId=0&materialId=slides&confId=1002](https://indico.hep.anl.gov/indico/getFile.py/access?contribId=6&sessionId=0&resId=0&materialId=slides&confId=1002).
APPENDIX A

MULTIPOLE-TO-LOCAL TRANSLATION IN 1D AND 3D
Changing the 3DM2L translation to 1DM2L involves four coordinate systems. They are listed below.

Original multipole expansion in \((X, Y, Z)_{m,o}\)

Rotated multipole expansion in \((X, Y, Z)_{m,r}\)

Original local expansion in \((x, y, z)_{l,o}\)

Rotated local expansion in \((x, y, z)_{l,r}\)

The subscripts \(l, m, o\) and \(r\) denote local, multipole, original and rotated, respectively (Fig. A.1).

![Figure A.1: Rotation of the coordinate system to perform 3DM2L to 1DM2L translation.](image)

The rotation about any arbitrary axis \(\hat{\omega}\) by an angle \(\theta\) can be described by the matrix \(R\).

\[
R = \begin{bmatrix}
  t\omega_x^2C & t\omega_x\omega_y - S\omega_z & t\omega_x\omega_z + S\omega_y \\
  t\omega_x\omega_y + S\omega_z & t\omega_y^2C & t\omega_y\omega_z - S\omega_x \\
  t\omega_x\omega_z - S\omega_y & t\omega_y\omega_z + S\omega_x & t\omega_z^2C
\end{bmatrix}
\]
where $C$, $S$ and $t$ denote $\cos \theta$, $\sin \theta$ and $(1-\cos \theta)$, respectively.

If $\vec{r}=(\vec{r}_x, \vec{r}_y, \vec{r}_z)$, $\vec{\omega}=(\vec{\omega}_x, \vec{\omega}_y, \vec{\omega}_z)$ and $z = (0, 0, 1)$, then $\omega_x = -\frac{r_x}{r}$, $\omega_y = -\frac{r_y}{r}$, $\omega_z = 0$.

The relationship between rotations can be given as follows:

\[
\begin{pmatrix}
X \\
Y \\
Z
\end{pmatrix}_{m,r} = R \begin{pmatrix}
X \\
Y \\
Z
\end{pmatrix}_{m,o} \tag{A.1}
\]

and

\[
\begin{pmatrix}
x \\
y \\
z
\end{pmatrix}_{l,r} = R \begin{pmatrix}
x \\
y \\
z
\end{pmatrix}_{l,o} \tag{A.2}
\]

The translation of the multipole expansion $M_o$ to local expansion $L_o$ can be expressed as a composition between $M_o$ and $M2L_o$.

\[
L_o \begin{pmatrix}
x \\
y \\
z
\end{pmatrix}_{l,o} = M_o \circ M2L_o \begin{pmatrix}
x \\
y \\
z
\end{pmatrix}_{l,o} \tag{A.3}
\]

$r$ is the distance between the centers of the boxes (invariant under $R$). Hence, we can establish the following relationships:

\[
\frac{1}{r^2} \begin{pmatrix}
X \\
Y \\
Z
\end{pmatrix}_{m,o} = M2L_o \begin{pmatrix}
x \\
y \\
z
\end{pmatrix}_{l,o} \tag{A.4}
\]
Also, the new operator $M2L_n$ can be introduced as,

$$
\frac{1}{r^2} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}_{m,r} = M2L_n \begin{pmatrix} x \\ y \\ z \end{pmatrix}_{i,r}
$$

(A.5)

The subscript $n$ stands for new. The two operators, $M2L_o$ and $M2L_n$, must give the same local expansion around centers of the nodes. Hence,

$$
L_n \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}_{i,o} = L_o \begin{pmatrix} x \\ y \\ z \end{pmatrix}_{i,o}
$$

(A.6)

Further simplification gives $L=M \circ R^{-1} \circ M2L_o \circ R$.

where $L_o = L_n = L$ and $M_o = M$. The rotation matrix is orthogonal. Hence, $R^{-1} = R^T$.

One can notice that the runtime taken with 1DM2L is slightly higher than that of 3DM2L until the order reaches 5 (Fig. A.1). On the other hand, after order 5 the runtime is significantly less with 1DM2L. Therefore, allowing the FMM code to select the appropriate method of multipole-to-local translation according to the required FMM order is a viable option to decrease the runtime.
APPENDIX B

ALGORITHMS
Algorithm 1 module:paricle.cpp Function: createParticles()

function: createParticles()
Input:input_data_file:string, nparticles:integer, relg:double
Output:particles: vector of particles
open input_data_file
if (input_data_file not open) then
    print “error: cannot open”
    exit
end if
define variable oneline
define variable num_available
while end of input_data_file do
    read a record into oneline
    num_available ← num_available + 1 ▷ Count number of particles in the input data file.
end while
if (num_available=0) then
    print “error: has no valid particle data”
    exit
end if
if (nparticles=0 or nparticles > num_available) then
    nparticles ← num_available
end if
if (relg <1) then
    print “The relativistic gamma factor cannot be less than 1”
    exit
end if
if (relg >1) then
    for n = 0 → nparticles do
        if (any of x, y, z coordinates of the point is missing) then
            print “error: incomplete read at line n+1”
            exit
        end if
        \( p_z \leftarrow p_z \times relg \) ▷ multiply \( p_z \) by \( relg \\
        \) particles[n] ← p ▷ Add particle \( \mathbf{p} \) to particles vector
    end for
**Algorithm 1** Module:paricle.cpp Function: createParticles() (Part-2)

```plaintext
else
    for n = 0 → nparticles do
        if (any of x, y, z coordinates of the point is missing) then
            print “error: incomplete read at line n + 1”
            exit
        end if
        particles[n] ← p
    end for
end if
```

End

---

**Algorithm 2** Module:octree.cpp FMM: Main Data Structure (Octree Node)

```plaintext
Begin
    Call OctreeNode() ▷ OctreeNode class is the application main data structure and it holds all particle information
    Call addNeighbor() ▷ Neighbor node is a OctreeNode type vector. Call this function to add neighbors to a particular node
    Call subDivide() ▷ Create 8 child node objects for a particular node
    Call getParentNode() ▷ Return parent node
    Call getChildNodes() ▷ Return the array of child nodes
    Call getLevel() ▷ Return the node level
    Call getNum() ▷ Return the node index, for a given level. This value is unique
    Call getNeighbors() ▷ Return all neighbors for a given node
    Call getSourceParticles() ▷ Return the source particle indices in a box
    Call getTargetParticles() ▷ Return the target particle indices in a box
    Call addAllParticles() ▷ Add both source and target particles
    Call addSourceParticles() ▷ Add source particles to OctreeNode structure
    Call addTargetParticles() ▷ Add target particles to OctreeNode structure
    Call getBoxCenter() ▷ return the coordinates of the center of the box
    Call setCArrayIndex() ▷ Set indices for CLset nodes
    Call setDArrayIndex() ▷ Set indices for Dtree nodes
End
```
Algorithm 3 Module:Octree.cpp Function: subDivide()

return immediately if child nodes already exist
if (child nodes exist) then
    return
else
    create child nodes
    allocate memory
end if
for \( j = 0 \rightarrow < 8 \) do
    save parent nodes of all 8 children
end for

Algorithm 4 Module:Octree.cpp Function: getBoxCenter()

Input : \( m_{x_{min}}, m_{y_{min}}, m_{z_{min}}, m_{length} \)
Output : box centers: \( cx, cy, cz \)
\[ cx \leftarrow \frac{(m_{x_{min}} + m_{length})}{2} \]
\[ cy \leftarrow \frac{(m_{y_{min}} + m_{length})}{2} \]
\[ cz \leftarrow \frac{(m_{z_{min}} + m_{length})}{2} \]

Algorithm 5 Module:Octree.cpp Function: addAllParticles()

Input : \( numSources, numTargets \)
Output : \( numSources, numTargets \)
for \( n = 0 \rightarrow numSources \) do
    \( m_{source\_particles}[n] \leftarrow n \)
end for
for \( n = 0 \rightarrow numTargets \) do
    \( m_{tagret\_particles}[n] \leftarrow n \)
end for
Algorithm 6 module:main.cpp Function: main()

Input: particle file (file with particles x, y, z coordinates), relg (value of the relativistic gamma), q (the maximum number of sources in the neighborhood of a particular target)

Output: FMM data files for PHAD

initialize parameters
call addAllParticles() ▷ add source and target particles to zeroBox

if (zeroBox = null) then
  print the error message “error creating zeroBox”
  exit
end if

if (zeroBox is not sub-divided) then
call distributeParticles()
end if

sub-divide level 1 boxes to level 2
call getChildrenNodes()

for i = 1 → 8 do
  if (zeroBox is not sub-divided) then
    call distributeParticles()
  end if
end for

call tagAndDivide()

for i = 1 → 8 do
  call removeSourcesandTargets()
end for

call constructDtree()
call constructCLTreeSet() ▷ Build C-forest trees
call topTomiddleandBottomTrees()
createCosyFile1()
createCosyFile2()
createCosyFile3()
createCosyFile4()

exit
End
**Algorithm 7** Module:main.cpp Function: constructDtree

**Input:** target_boxes_list: OctreeNode type 2D vector

for $i = (\text{maxlevel} - 1) \rightarrow 0$ do
    for $j = 0 \rightarrow < \text{number of elements in target_boxes_list}[i]$ do
        save target_boxes_list[i][j] in dtree[i]
    end for
end for

if ($i \neq \text{maxlevel} - 1$) then
    for $j = 0 \rightarrow < \text{number of elements in dtree}[i+1]$ do
        parent_box ← parentBox of dtree[i+1][j]
        inDtree ← false
        for $k = 0 \rightarrow < \text{number of elements in dtree}[i]$ do
            if (dtree[i][k] = parent_box) then
                inDtree ← true
            end if
        end for
        if (inDtree $\neq$ true) then
            dtree[i + 1][j] ← parent_box
        end if
    end for
end if

for $k = 0 \rightarrow < \text{number of elements in dtree}[i]$ do
    counter ← $\text{counter} + 1$
    save dtree[i][k] in AllDtreeNodes
    dtree[i][k] → darrayIndex(counter)
end for

**Algorithm 8** Module:main.cpp Function: getMaxLevel()

**Input:** target_boxes_list: OctreeNode type vector

**Output:** maxlevel: integer

maximumLevel ← 21

for $i = \text{maximumLevel} - 1 \rightarrow 0$ do
    if (number of target boxes in target_boxes_list[i] $> 0$) then
        maxlevel ← i
        break
    end if
end for
Algorithm 9 Module:main.cpp Function: giveBoxNumber()

Input: particle coordinates nx, ny, nz
Output: boxNumber:unsigned integer

dimension, d ← 3, precision ← 21, mask ← 0x1L, fn_integer ← 0x0L

Multiply nx, ny and nz by $2^{21}$

for $i \leftarrow (\text{precision} - 1) \rightarrow \geq 0$ do

result ← shift left i positions of mask and mask it with nx

if (result = 0) then
    shift left 1 position of fn_integer
else
    shift left 1 position of fn_integer and append 1
end if

result ← shift left i positions of mask and mask it with ny

if (result = 0) then
    shift left 1 position of fn_integer
else
    shift left 1 position of fn_integer and append 1
end if

result ← shift left of i positions and mask it with nz

if (result = 0) then
    shift left 1 position of fn_integer
else
    shift left 1 position of fn_integer and append 1
end if

end for

fn_integer ← 0x0L

boxNumber ← shift right (precision*d - m) positions of fn_integer
Algorithm 10 Module:main.cpp Function: getInteractionList()

Input: zeroBox:OctreeNode type vector, selfBox:OctreeNode type vector
Output: InteractionList:OctreeNode type vector

for \( p = 1 \rightarrow \) size of the parent’s neighbor set do
  \( \text{neighbor}_\text{children} \leftarrow \text{parentNeighborSet}[p] \)
  if (\( \text{neighbor}_\text{children} \) exists) then
    for \( m = 0 \rightarrow 8 \) do
      save \( \text{neighbor}_\text{children}[m] \) in parentNeighborChildSet
    end for
  end if
end for

generate self neighbors
get self neighbors

for \( pnc = 0 \rightarrow \) size of the parent’s neighbor child set do
  for \( sn = 0 \rightarrow \) size of the self neighbors do
    if (\( \text{parentNeighborChildSet}[\text{pnc}] \leftarrow \text{selfNeighborSet}[\text{sn}] \)) then
      get the next \( pnc \) and check \( \text{parentNeighborChildSet}[\text{pnc}] \)
      against \( \text{selfNeighborSet}[\text{sn}] \)
    end if
    if (number of sources in the parent neighbor child set > 0) then
      save \( \text{parentNeighborChildSet} \) in the InteractionList vector
    end if
  end for
end for

Algorithm 11 Module:main.cpp Function: quickSort()

Input: dtrow:OctreeNode type vector, left:integer, right:integer

\( i \leftarrow \text{left}, j \leftarrow \text{right} \)
\( \text{pivot} \leftarrow \text{left} + ((\text{right}-\text{left}) / 2) \)
\( \text{pivotBox} \leftarrow \text{box number of dtrow}[\text{pivot}] \)

while \( i \leq j \) do
  while \( i \) (box number of dtrow[\( i \]) < pivotBox) do
    \( i \leftarrow i + 1 \)
  end while
  while \( i \) (box number of dtrow[\( j \]) > pivotBox) do
    \( j \leftarrow j - 1 \)
Algorithm 11 Module: main.cpp Function: quikSort() (part-2)

\[
\text{end while}
\]
\[
\text{if } i \leq j \text{ then}
\]
\[
\text{tmp} \leftarrow \text{dtrow}[i]
\]
\[
\text{dtrow}[i] \leftarrow \text{dtrow}[j]
\]
\[
\text{dtrow}[j] \leftarrow \text{tmp}
\]
\[
i \leftarrow i + 1
\]
\[
j \leftarrow j - 1
\]
\[
\text{end if}
\]
\[
\text{end while}
\]
\[
\text{if } left < j \text{ then}
\]
\[
\text{quicksort(dtrow, left, j)}
\]
\[
\text{end if}
\]
\[
\text{if } i < right \text{ then}
\]
\[
\text{quicksort(dtrow, i, right)}
\]
\[
\text{end if}
\]

Algorithm 12 Module: main.cpp Function: removeDuplicates()

\textbf{Input:} induplicates\_list: OctreeNode type vector
\textbf{Output:} noduplicate\_list: OctreeNode type vector

Clear noduplicates\_list
\[
\text{if (number of elements in duplicates\_list > 0) then}
\]
\[
\text{save duplicates\_list[0] in noduplicates\_list}
\]
\[
\text{end if}
\]
\[
\text{for } i = 1 \rightarrow < \text{number of elements in duplicates\_list} \text{ do}
\]
\[
\text{if (box number of duplicates\_list[i - 1] \neq box number of duplicates\_list[i]) then}
\]
\[
\text{save duplicates\_list[i] in noduplicates\_list}
\]
\[
\text{end if}
\]
\[
\text{end for}
\]
Algorithm 13 Module:main.cpp Function: findBox()

Input: zeroBox:OctreeNode type vector, level:integer, number:integer
Output: currBox:OctreeNode

dimension ← 3
currBox ← zeroBox
parentBox.num ← shift right 3 positions of currBox.num
unsigned mask ← 0x7L
next_level ← 1
while (no box_found) do
    if (level of the currBox = level) and (box number of the currBox = num) then
        box_found ← true
    end if
    if (no box_found) then
        currBox.children ← child nodes of currBox
        idx ← shift right (level − nextlevel) * dimension number of positions
        and append the mask
        if (currBox.children = 0) then
            stop
        end if
        currBox ← currBox.children[idx]
        nextlevel ← nextlevel + 1
    end if
end while

Algorithm 14 Module:main.cpp Function: tagAndDivide()

Input: zeroBox:OctreeNode type vector, particle_limit_loc:integer
for i = 0 →< 8 do
    currBox.child_nodes ← child nodes of zeroBox.children[i]
for j = 0 →< 8 do
    save currBox.child_nodes[j] in bfs-queue
end for
end for
while (bsf-queue is not empty) do
    currBox ← front element of bfs-queue
    currBox.num_target_particles ← number of target particles in the currBox
    if (currBox.num_target_particles > 0) then

**Algorithm 14** Module:main.cpp Function: tagAndDivide() (part-2)

if (number of Sources in the neighborhood of currBox) ≤ (particle_limit) then
    tag the currBox
    save currBox in target_boxes_list[level of currBox - 2]
    target-rem number of target particles in currBox - 1
end if
end if
if (level of the currBox = max_level) then
    stop the program
end if
for i = 0 → < number of element in currBox.neighbors do
    if (currBox.neighbors[i] box is not divided) then
        distributeParticles in currBox.neighbors[i]
    end if
    if (level of currBox.neighbors[i] ≠ level of currBox) then
        print an error message: “neighbor is not at the same level as currBox”
    end if
    currBox.child_nodes ← child nodes of currBox
    for j = 0 → < 8 do
        save currBox.child_nodes[j] in bfs-queue
    end for
end for
remove currBox from the queue
end while

**Algorithm 15** Module:main.cpp Function: generateNeighbors()

Input: zeroBox:OctreeNode type vector
if (nx = 0) then
    save nx, nx+1 in NBx
end if
if (nx = ((mask << currBox.level) – 1)) then
    save nx-1, nx in NBx
else
    save nx-1, nx, nx+1 in NBx
end if
**Algorithm 15** Module:main.cpp Function: generateNeighbors() (part-2)

```plaintext
if (ny = 0) then
    save ny, ny+1 in NBy
end if
if (ny = ((mask << currBoxlevel) - 1)) then
    save ny-1, ny in NBy
else
    save ny-1, ny, ny+1 in NBy
end if
if (nz = 0) then
    save nz, nz+1 in NBz
end if
if (nz = ((mask << currBoxlevel) - 1)) then
    save nz-1, nz in NBz
else
    save nz-1, nz, nz+1 in NBz
end if
for i = 0 → < number of elements in NBx do
    for j = 0 → < number of elements in NBy do
        for k = 0 → < number of elements in NBz do
            for p = (currBoxlevel - 1) → ≤ 0 do
                result = shift left p positions of mask and mask it with NBx[i]
                if (result ≠ 0) then
                    tmpneighbor = shift left 1 position of tmpneighbor and append 1
                else
                    tmpneighbor = shift left 1 position of tmpneighbor
                end if
                result = shift left p positions of mask and mask it with NBy[j]
                if (result ≠ 0) then
                    tmpneighbor = shift left 1 position of tmpneighbor and append 1
                else
                    tmpneighbor = shift left 1 position of tmpneighbor
                end if
                result = shift left p positions of mask and mask it with NBz[k]
                if (result ≠ 0) then
                    tmpneighbor = shift left 1 position of tmpneighbor and append 1
                else
                    tmpneighbor = shift left 1 position of tmpneighbor
                end if
            end for
        end for
    end for
end for
```

Algorithm 16 Module:main.cpp Function: sortAndCalcParents()

Input: finestlevel = 19, dimension = 3

for $i = 0 \rightarrow < \text{number of elements in sourceparticles vector}$ do
  number $\leftarrow$ box number of the source particle $i$ with coordinates($x$, $y$, $z$)
  at the finest level
  save number in correspondenceVec
  save number in parentvecsources[finestlevel]
end for

for $i = 0 \rightarrow < \text{number of elements in source-particles vector}$ do
  parray_sources[$i$] = $i$
end for

sort parray_sources with correspondenceVec
release the memory allocated to parray_sources array

for $i = \text{finestlevel} - 1 \rightarrow \geq 0$ do
  for $j = 0 \rightarrow < \text{number of elements in correspondenceVec}$ do
    parentboxnumber $\leftarrow$ shift right (($\text{finestlevel} - i) \times \text{dimension}$)
    places of correspondenceVec[$j$]
    save parentboxnumber in parentvecsources[$i$]
  end for
end for

clear correspondenceVec

for $i = 0 \rightarrow < \text{number of elements in targetparticles vector}$ do
  number $\leftarrow$ box number of the target particle $i$ with coordinates($x$, $y$, $z$) at the finest level
  save number in correspondenceVec
  save number in parentvectarget[finestlevel]
end for

for $i = 0 \rightarrow < \text{number of elements in targetparticles vector}$ do
  parraytargets[$i$] = $i$
end for

sort parray_targets with correspondenceVec
release the memory allocated to parray_targets

for $i = \text{finestlevel} - 1 \rightarrow \geq 0$ do
  for $j = 0 \rightarrow < \text{number of elements in correspondenceVec}$ do
    parentboxnumber $\leftarrow$ shift right (($\text{finestlevel} - i) \times \text{dimension}$)
    places of correspondenceVec[$j$]
    save parentboxnumber in parentvectargets[$i$]
  end for
end for
Algorithm 17 Module:main.cpp Function: topTomiddleandBottomTrees()

**Input:** CLset : 2d vector of OctreeNode*
**output:** T2Btrees : Matrix of OctreeNode*

Declare vector v1, v2, nodevector3 : vector of OctreeNode*
Declare TreeGrow: 2d vector of OctreeNode*
Declare variable e1 : OctreeNode*
Declare variable children : pointer of OctreeNode*
Declare variable nd3, curr\_tree, MaxiLevel : integer
MaxiLevel = call getMaxLevel()

for $k = 0 \rightarrow CLset[0].size$ do
  initialize vector v1
  initialize vector v2
  $e1 \leftarrow CLset[0][k]$
  T2Btrees.add (OctreeNode*)
  T2Btrees.add (e1)
  if ($(e1\rightarrow getChildNodes()) == null)$ then
    $curr\_tree \leftarrow curr\_tree + 1$
    continue
  end if
  v1.add (e1)
  for $n = 0 \rightarrow MaxiLevel$ do
    for $s = 0 \rightarrow size of v$ do
      $children \leftarrow (v1\rightarrow getChildNodes())$
      if $(children == null)$ then
        continue
      end if
      $onechildfound = 0$
      for $m = 0 \rightarrow 8$ do
        for $b = 0 \rightarrow CLset[n].size$ do
          if $(children[m] == CLset[n][b])$ then
            v2.add(children[m])
            TreeGrow[n].add(children[m])
            T2Btrees[curr\_tree].add(children[m])
            $onechildfound \leftarrow onechildfound + 1$
          end if
        end for
      end for
    end for
  end for
  v1.clear()
  for $g = 0 \rightarrow size of v2$ do
    v1.add(v2[g])
  end for
Algorithm 17 Module:main.cpp Function: topTomiddleandBottomTrees() (part-2)

\[
v2\text{.clear()}
\]
end for
\[
curr\_tree \leftarrow curr\_tree + 1
\]
end for
for \( v = 0 \rightarrow \langle MaxiLevel \) do
if \( (CLset[v].size == TreeGrow[v].size) \) then
continue
end if
\[
growth \leftarrow TreeGrow[v].size
\]
if \( (growth > 1) \) then
call quicksort(TreeGrow[v], 0, growth - 1)
end if
for \( u = 0 \rightarrow \langle CLset[v].size() \) do
OctreeNode* \( \text{tem} = CLset[v][u] \)
if not \( (\text{findElement(TreeGrow[v], \text{tem}, 0, TreeGrow[v].size - 1)}) \) then
clear v1
clear v2
add(tem) to v1
add(tem to \( T2Btrees[curr\_tree] \))
if \( (v = MaxiLevel) \) then
\( \text{parentnotexist} \leftarrow true \)
OctreeNode* \( \text{te} \rightarrow (\text{tem} \rightarrow getParentNode()) \)
for \( j = 0 \rightarrow \langle CLset[v - 1].size \) do
if \( (\text{te} = CLset[v - 1][j]) \) then
\( \text{parentnotexist} \leftarrow false \)
break
end if
end for
if \( (\text{parentnotexist} = true) \) then
add(tem) to nodevector3
\( nd3 \rightarrow nd3 + 1 \)
end if
end if
end for
for \( n = v + 1 \rightarrow \langle MaxiLevel \) do
for \( s = 0 \rightarrow \langle sizeofv \) do
children \( \leftarrow (v1[s] \rightarrow getChiledNodes()) \)
if \( (children = null) \) then
OctreeNode* \( \text{te} \leftarrow (v1[s] \rightarrow getParentNode()) \)
if not(\( \text{findElement(CLset[n - 2], \text{te}, 0, CLset[n - 2].size - 1) \) then
add \( (v1[s] \rightarrow \text{nodevector3}) \)
\( nd3 \leftarrow nd3 + 1 \)
end if
continue
end if
end if
Algorithm 17 Module:main.cpp Function: topTomiddleandBottomTrees() (part-3)

\[\begin{align*}
on\text{onechiled} & \leftarrow 0 \\
on\text{set}_n\text{.size} & \leftarrow CL\text{set}[n]\text{.size}() \\
\text{for } m = 0 \rightarrow < 8 \text{ do} & \\
& \text{if } (\text{findElement}(CL\text{set}[n], children[m], 0, cl\text{set}_n\text{.size} - 1)) \text{ then} \\
& \quad v2\text{.add(children[m])} \\
& \quad Tree Grow[n]\text{.add(children[m])} \\
& \quad T2Btrees[\text{curr_tree}]\text{.add(children[m])} \\
& \quad \text{onechild} \leftarrow \text{onechild} + 1 \\
& \text{end if} \\
& \text{end for} \\
& \text{if } (\text{onechiled} = 0) \text{ then} \\
& \quad \text{OctreeNode}^{*} \text{te} \leftarrow v1[s] \rightarrow \text{getParentNode}() \\
& \quad \text{if not(findElement}(CL\text{set}[n - 2], \text{te}, 0, CL\text{set}[n - 2]\text{.size}() - 1)) \text{ then} \\
& \quad \quad \text{node}\text{vector}3\text{.add(v1[s])} \\
& \quad \quad \text{onechild} \leftarrow \text{onechild} + 1 \\
& \quad \text{end if} \\
& \text{end if} \\
& v1\text{.clear}() \\
& \text{for } (g = 0 \rightarrow g < v2\text{.size}()) \text{ do} \\
& \quad v1\text{.add(v2}[g]) \\
& \text{end for} \\
& v2\text{.clear}() \\
& \text{end for} \\
& \text{curr}\_\text{tree} \leftarrow \text{curr}\_\text{tree} + 1 \\
& \text{end if} \\
& \text{end for} \\
& ntrees\_\text{created} \leftarrow T2Btrees\text{.size}() \\
& \text{for } tree = 0 \rightarrow < ntrees\_\text{created} \text{ do} \\
& \quad \text{tree}\text{Array}[\text{tree}]\text{.min} \leftarrow (T2Btrees[\text{tree}][0] \rightarrow \text{getLevel}()) \\
& \quad \text{tree}\text{Array}[\text{tree}]\text{.max} \leftarrow (T2Btrees[\text{tree}][T2Btrees[\text{tree}]\text{.size}() - 1] \rightarrow \text{getLevel}()) \\
& \text{end for} \\
& MaxC\text{ForestCurrentLevelNodes} \leftarrow 0 \\
& \text{for } level = 0 \rightarrow <= \text{MaxLevel} \text{ do} \\
& \quad \text{if } (\text{Tree Grow}[\text{level}]\text{.size}() > MaxC\text{ForestCurrentLevelNodes}) \text{ then} \\
& \quad \quad MaxC\text{ForestCurrentLevelNodes} \leftarrow \text{Tree Grow}[\text{level}]\text{.size}() \\
& \quad \text{end if} \\
& \text{end for}
\end{align*}\]
Algorithm 17 Module:main.cpp Function: topToMiddleAndBottomTrees() (part-4)

for level = 0 →<= MaxiLevel do
    n31 ← 0
    levelNodesCount ← 0
    for a → nodevector3.size() do
        n31 ← nodevector3[a] → getLevel()
        if (n31 = level) then
            levelNodesCount ← levelNodesCount + 1
        end if
    end for
    if (levelNodesCount > MaxCForestCurrentLevelNodes) then
        MaxCForestCurrentNodes ← levelNodesCount
    end if
end for

Algorithm 18 Module:PHAD Procedure: MILTIPOLE

Input: particle charge $qs$, coordinates of the target $i$ $(x_i, y_i, z_i)$, and coordinates of the box center $c$ $(x_c, y_c, z_c)$

Output: $m$: DA vector type element

$sx ← x_i - x_c$, $sy ← y_i - y_c$, $sz ← z_i - z_c$

$m ← (da(1)^2 + da(2)^2 + da(3)^2)$

$m ← qs/\sqrt{(1 + sx^2 + sy^2 + sz^2) * m - 2 * sx * da(1) - 2 * sy * da(2) - 2 * sz * da(3)}$

End of procedure MULTIPOLE

Algorithm 19 Module:PHAD Procedure: L2L

Input: coordinates of the child box center $cbc_x, cbc_y, cbc_z$, coordinates of the parent box center $(pbc_x, pbc_y, pbc_z)$, local expansion of the parent $locexp$

$a(1) ← 1$, $a(2) ← 1$, $a(3) ← 1$

call:datrn $locexp, a, 1, 3, partochild$  \(\triangleright\) translate local expansion of parent to child

return $partochild$
**Algorithm 20** Module: PHAD Procedure: M2M

**Input:** coordinates of the parent box center \( x_p, y_p, z_p \),
coordinates of the child box center \( (x_c, y_c, z_c) \),
multipole expansion of the child \( m_{ul} \)

\[
x \leftarrow (x_p - x_c), y \leftarrow (y_p - y_c), z \leftarrow (z_p - z_c)
\]
\[
r \leftarrow (da(1)^2 + da(2)^2 + da(3)^2)
\]
\[
r \leftarrow 1/(1 + (x^2 + y^2 + z^2) * r + 2 * x * da(1) + 2 * y * da(2) + 2 * z * da(3))
\]
\[
\text{map}_\text{trans}(1) \leftarrow (da(1) + x * (da(1)^2 + da(2)^2 + da(3)^2)) * r \quad \triangleright \text{map_trans is a differential algebraic vector}
\]
\[
\text{map}_\text{trans}(2) \leftarrow (da(1) + y * (da(1)^2 + da(2)^2 + da(3)^2)) * r
\]
\[
\text{map}_\text{trans}(3) \leftarrow (da(1) + z * (da(1)^2 + da(2)^2 + da(3)^2)) * r
\]
\[
mefun \leftarrow \text{mulexp}
\]
\[
call: \text{polval} \text{mefun }1 \text{ map}_\text{trans} \text{3 mefun }1 \quad \triangleright \text{translate multipole expansion of child to parent}
\]
\[
mm \leftarrow \text{mefun}(1) * \sqrt{r}
\]
return \( mm \)

**Algorithm 21** Module: PHAD Procedure: M2L

**Input:** coordinates of box center \( \text{binx, biny, binz} \),
coordinates of the Interaction list box \( \text{(ix, iy, iz)} \),
multipole expansion of the interaction list \( \text{mulomul} \)

translate multipole to local \( \text{locexp} \)
\[
xx = \text{binx - ix}, yy = \text{biny - iy}, zz = \text{binz - iz}
\]
if \((xx=0)\text{and}(yy=0)) \text{ then}

\[
evaluate \text{mulomul} \text{ get } \text{locexp} \quad \triangleright \text{“mulomul” : multipole to multipole}
\]
else

calculate the distance between the boxes

calculate angle between \( z \) axes of the boxes
perform inverse rotation
translate multipole to local
perform rotation
\[
evaluate \text{mulomul} \text{ get } \text{locexp}
\]
end if

return \( \text{locexp} \)
Algorithm 22  Module: PHAD Procedure: L2P

Input: coordinates of target, coordinates of the box center, local expansion of the box (local), particle limit
calculate spacial derivative of local expansion $E_x, E_y, E_z$
evaluate $E_x, E_y, E_z$ and local at the target
return potential and electric field at the target

Algorithm 23  Module: PHAD Procedure: RELGCALC (Relativistic Gamma Calculation)

initialize variables
sumBetaProton ← 0, sumBetaElectron ← 0
averageBetaProton ← 0, averageBetaElectron ← 0
for $i = 1 \rightarrow$ number of particles do
    if ($i = 0 < \text{num of protons} + 1$) then
        sumBetaProton ← sum(proton_velocity(i))
    else
        sumBetaElectron ← sum(electron_velocity(i))
    end if
end for
calculate averageBetaProton
calculate averageBetaElectron
calculate average velocity of particles, averageBETA
relativistic gamma= $\sqrt{1 - \text{averageBETA}^2}$
return relativistic gamma
End of procedure RELGCALC

Algorithm 24  Module: PHAD Procedure: STATER

Initialize variables
$h \leftarrow$ timestepsize
$nts \leftarrow \text{num of timesteps} - 1$
call Procedure:fmmwithintegrators
End of procedure STATER
Algorithm 25 Module: PHAD Procedure: PICARDINT

Input: Initial unscaled particle coordinates and momenta(Y) from FMM, number of iterations, number of particles, largest time step size, Picard order, FMM order

for node = 1 → dLeafNodes do
  for node = 1 → num_of_selfsources do
    index ← index of sourcenumbs(node)
    get un-scaled coordinates of sources
  end for
end for

for J = 1 → number of iterations do
  for node = 1 → dLeafNodes do
    for I = 1 → num_of_selfsources do
      Transform Y to Φ
    end for
  end for
  for L = 1 → order do
    danot L ▷ danot is a COSY INFINITY intrinsic function
    for node = 1 → dLeafNodes do
      for I = 1 → num_of_selfsources do
        get the spacial derivatives gg(1, 2, 3) ← Φ(1, 2, 3)
      end for
      for K = 1 → num_of_selfsources do
        get the momentum derivatives gg(4, 5, 6) ← Φ(4, 5, 6)
      end for
      for K = 1 → countsourceswithoutself do
        get the momentum derivatives gg(4, 5, 6) ← Φ(4, 5, 6)
      end for
      gg(4, 5, 6) ← gg(4, 5, 6)/relg ▷ modify with relativistic gamma
    end for
  end for
end for

for node = 1 → dLeafNodes do
  for I = 1 → num_of_selfsources do
    assign Φ2 to Φ
    Φ(1, 2, 3, 4, 5, 6) ← Φ2(1, 2, 3, 4, 5, 6)
  end for
end for
Algorithm 25 Module:PHAD Procedure: PICARDINT (part-2)

for node = 1 → dLeafNodes do
    for ie = 1 → num_of_selfsources do
        $gg(1, 2, 3, 4, 5, 6) \leftarrow \Phi(1, 2, 3, 4, 5, 6)$
        evaluate $gg$ ▷ compute $(x, y, z, px, py, pz)$ for the next step
        $Y(1, 2, 3, 4, 5, 6) \leftarrow gg(1, 2, 3, 4, 5, 6)$
    end for
end for

$elaptm \leftarrow (elaptm + delt)$; $J \leftarrow J + 1$
end for
End of procedure PICARDINT

Algorithm 26 Module:PHAD Procedure: RUN wrap other procedures

call procedure: initializevariables ▷ This procedure includes the main controlling parameters of the application.
call procedure: get_mpi_info ▷ This procedure passes mpirun command line parameters to the application.
call procedure: callconstants ▷ This procedure defines all constants used in the application.
call procedure: get_fmmcpp_params ▷ This procedure reads and assigns input parameters from the input file.
resolutioncounter $\leftarrow (outputresolution - 1)$ ▷ outputresolution - input parameter defines the number of time steps the application executed before writing output data to files.
nel $\leftarrow (num_of_particles - npr)$ ▷ nel - Number of electrons, num_of_particles : total number of particles, npr : Number of protons.
nexttimestep $\leftarrow numtimesteptoreplacecoordinates$ ▷ numtimesteptoreplacecoordinates - number of time steps program must execute before refreshing the electron bunch
call procedure: open_initialoutput_files
call procedure: starter ▷ This procedure calls procedures that include the application controlling logic.
call procedure: close_initialoutput_files
if $(ip = 1)$ then
    write run summary to files
end if
End of procedure RUN
Algorithm 27 Module: PHAD Procedure: FMMWITHINTEGRATORS

Input: Application configuration data file, initial particles file

Initialize variables

\[ \text{initial} \leftarrow 1 \]

call procedure: input_initial_config \( \triangleright \) This procedure reads initial particle information from an input file and store them in vectors.

call procedure: input_initial_momenta \( \triangleright \) This procedure reads input momenta from a file and store them in a vector.

call procedure: input_initial_massandcharge \( \triangleright \) This procedure reads initial mass and charge from a file, and store in a vector.

\[ \text{pic\_order} \leftarrow \text{max\_pic\_order} \]

\[ q \leftarrow q_{\text{max}} \]

call procedure: open_initialvelocity_outfiles

for \( i = 1 \rightarrow \text{np} \) do

\[ \text{call procedure: relgc} \]

\( \triangleright \) This implementation guarantees that each processor runs this application in parallel (MPI environment) to calculate the relativistic gamma value.

end for

call procedure: close_initialvelocity_outfiles

\[ \text{numtimestepstowriteaveragevalues} \leftarrow 0 \]

call procedure: open_betacalc_outfiles

call procedure: fmm

\[ h \leftarrow h/2 \]

call procedure: lngrngint

\[ \text{initial} \leftarrow 0 \]

for \( timestepcounter = 1 \rightarrow \text{nts} \) do

\[ \text{numtimestepstowriteaveragevalues} \leftarrow \text{numtimestepstowriteaveragevalues} + 1 \]

\[ h \leftarrow 2 \times h \]

call procedure: picardint

\[ h \leftarrow h/2 \]

call procedure: modlngrngint

end for

\[ timestepcounter \leftarrow timestepcounter + 1 \]

\[ h \leftarrow 2 \times h \]

call procedure: picardint

\[ h \leftarrow h/2 \]

call procedure: fmm

call procedure: modlngrngint

call procedure: close_betacalc_outfiles
Algorithm 28 Module: PHAD Procedure: MODLNGRNGINT

Input: dLeafNodes (number of leaf nodes), source particle coordinates, box center coordinates, particles momenta (px, py, pz), prevneighlist (previous neighbour list)

Initialize variables

\textbf{for} node = 1 \rightarrow dLeafNodes \textbf{do} \\
\hspace{1em} neighbnumbsprev(node) \leftarrow qsort() \\
\hspace{1em} templist \leftarrow merg() \\
\hspace{1em} templist(1) \leftarrow sourcectprev(node) + neighbctprev(node) \hspace{1em} \triangleright \text{Assign source count and neighbor count of the previous node to the first element of the vector templist} \\
\hspace{1em} \textbf{for} i = 1 \rightarrow sourcectprev(node) \textbf{do} \\
\hspace{1.5em} prevneighlist(I) \leftarrow templist \\
\textbf{end for} \\
\textbf{end for} \\
csp \leftarrow \text{lightspeed} \quad \triangleright \text{Assign speed of light to csp} \\
M \leftarrow pmass \quad \triangleright \text{Assign proton mass to M} \\
momconst \leftarrow h * echarge^2 / (\epsilon_0 \epsilon_0 \epsilon_0 \epsilon_0 M \times (csp)^2 \times dzero^2) \\
\textbf{for} node = 1 \rightarrow dLeafNodes \textbf{do} \\
\hspace{1em} num_of selfsources \leftarrow sourcect(node) \\
\hspace{1em} \textbf{for} i = 1 \rightarrow num_of selfsources \textbf{do} \\
\hspace{2em} index \leftarrow sourcenumbs(node)(i) \hspace{1em} \triangleright \text{Retrieve the } I^{th} \text{ element of the vector sourcenumbs(node)} \\
\hspace{2em} relx(index) \leftarrow source_x(index) - leaf_cx(node) \\
\hspace{2em} rely(index) \leftarrow source_y(index) - leaf_cy(node) \\
\hspace{2em} relz(index) \leftarrow source_z(index) - leaf_cz(node) \\
\hspace{1em} \textbf{end for} \\
\textbf{end for} \\
\textbf{for} jj = 1 \rightarrow np \textbf{do} \\
\hspace{1em} \text{call procedure: get_loop_limits(jj, np, dLeafNodes, n1, n2)} \\
\hspace{1em} \textbf{for} node = n1 \rightarrow n2 \textbf{do} \\
\hspace{2em} num_of selfsources \leftarrow sourcect(node) \\
\hspace{2em} \text{call procedure: qsort to sort neighbnumbs(node) vector} \\
\hspace{2em} \text{call procedure: merge to merg sourcenumbs(node), neighbnumbs(node) vectors} \\
\hspace{2em} index \leftarrow sourcenumbs(node)(i) \hspace{1em} \triangleright \text{i}^{th} \text{ element of sourcenumbs(node) vector} \\
\hspace{2em} \textbf{for} i = 2 \rightarrow num_of selfsources \textbf{do} \\
\hspace{3em} points : 3 \times Nvector \leftarrow relx(index), rely(index), relz(index) \\
\hspace{1em} \textbf{end for} \\
efield(x, y, z) \leftarrow (\text{get the derivative of locexpnode(x,y,z)}) \hspace{1em} \triangleright \text{efield : Electric Field} \\
\text{call procedure: polval to evaluate the filed at the target}
Algorithm 28 Module: PHAD Procedure: MODLNGRNGINT (part-2)

for \( i = 1 \rightarrow \text{num\_of\_self\_sources} \) do
    comparelist \( \leftarrow \) call function: compare(templist, prevneighlist(index))
    if (num\_of\_self\_sources \( \geq 1 \)) then
        tempone \( \leftarrow \) source\(_n\)(index) * momconst
        source\_px(index) \( \leftarrow \) source\_px(index) + (tempone * result\_x(I)) / relg
        source\_py(index) \( \leftarrow \) source\_py(index) + (tempone * result\_y(I)) / relg
        source\_pz(index) \( \leftarrow \) source\_pz(index) + (tempone * result\_z(I))
    end if
end for

sumone \( \leftarrow 0 \), sumtwo \( \leftarrow 0 \), sumthree \( \leftarrow 0 \)
A = templist = \((a_1, a_2, \ldots, a_n)\)
B = prevneighlist(index) = \((b_1, b_2, \ldots, b_n)\)
for \( J \rightarrow (A \setminus B).size \) do
    index2 \( \leftarrow \) comparelist(j)
    tempone \( \leftarrow \) source\_x(index) - source\_x(index2)
    temptwo \( \leftarrow \) source\_y(index) - source\_y(index2)
    tempthree \( \leftarrow \) (source\_z(index) - source\_z(index2))
    tempfour \( \leftarrow \) tempone\(^2\) + temptwo\(^2\) + tempthree\(^2\)
    tempfour \( \leftarrow \) sqrt(tempfour)
    tempfour \( \leftarrow \) tempfour\(^3\) / source\_n(index2)
    sumone \( \leftarrow \) sumone + (tempone / tempfour) / relg
    sumtwo \( \leftarrow \) sumtwo + (temptwo / tempfour) / relg
    sumthree \( \leftarrow \) sumthree + (tempthree / tempfour)
end for

tempone \( \leftarrow \) source\_n(index) * momconst
checkpotential \( \leftarrow \) checkpotential - sumone
source\_px(index) \( \leftarrow \) source\_px(index) + tempone * sumone
source\_py(index) \( \leftarrow \) source\_py(index) + tempone * sumtwo
source\_pz(index) \( \leftarrow \) source\_pz(index) + tempone * sumthree
sumone \( \leftarrow 0 \), sumtwo \( \leftarrow 0 \), sumthree \( \leftarrow 0 \)
for \( J \rightarrow (B \setminus A).size \) do
    index2 \( \leftarrow \) comparelist(J)
    tempone \( \leftarrow \) source\_x(index) - source\_x(index2)
    temptwo \( \leftarrow \) source\_y(index) - source\_y(index2)
    tempthree \( \leftarrow \) (source\_z(index) - source\_z(index2))
    tempfour \( \leftarrow \) tempone\(^2\) + temptwo\(^2\) + tempthree\(^2\)
    tempfour \( \leftarrow \) sqrt(tempfour)
Algorithm 28 Module:PHAD Procedure: MODLNGRNGINT (part-3)

\[
\text{tempfour} \leftarrow \text{tempfour}^3 / \text{source}_n(\text{index2}) \\
\text{sumone} \leftarrow \text{sumone} + (\text{tempone} / \text{tempfour}) / \text{relg} \\
\text{sumtwo} \leftarrow \text{sumtwo} + (\text{temptwo} / \text{tempfour}) / \text{relg} \\
\text{sumthree} \leftarrow \text{sumthree} + (\text{tempthree} / \text{tempfour}) \\
\]

end for

\[
\text{tempone} \leftarrow \text{source}_n(\text{index}) \ast \text{momconst} \\
\text{checkpotential} \leftarrow \text{checkpotential} - \text{sumone} \\
\text{source}_p\text{x}(\text{index}) \leftarrow \text{source}_p\text{x}(\text{index}) + \text{tempone} \ast \text{sumone} \\
\text{source}_p\text{y}(\text{index}) \leftarrow \text{source}_p\text{y}(\text{index}) + \text{tempone} \ast \text{sumtwo} \\
\text{source}_p\text{z}(\text{index}) \leftarrow \text{source}_p\text{z}(\text{index}) + \text{tempone} \ast \text{sumthree} \\
\text{sumone} \leftarrow 0, \text{sumtwo} \leftarrow 0, \text{sumthree} \leftarrow 0
\]

if (timestepcounter + 1 > resolutioncounter) then

\[
\text{resc} \leftarrow (\text{resolutioncounter} + 1) \\
\text{write particle coordinates (x, y, z) and momenta (px, py, pz) to files in binary} \\
\text{> This information is used to calculate the emittance of the beam} \\
\text{resolutioncounter} \leftarrow (\text{resolutioncounter} + \text{outputresolution})
\]

end if

end for
end for
end for

End of procedure MODLNGRNGINT

Algorithm 29 Module:PHAD Procedure: LNGRNGINT

Input: dLeafNodes(number of leaf nodes), source particle coordinates, box center coordinates, particles momenta(px, py, pz)

Initialize variables

for node = 1 → dleafnodes do

\[
\text{sourcenumbsprev}(\text{node}) \leftarrow 0 \\
\text{neighbnumbsprev}(\text{node}) \leftarrow 0 \\
\text{num_of_self_sources} \leftarrow \text{sourcect}(\text{node}) \\
\text{sourcectprev}(\text{node}) \leftarrow \text{num_of_self_sources}
\]

for I → num_of_self_sources do

\[
\text{index} \leftarrow \text{sourcenumbs}(\text{node})(I) \\
\text{sourcenumbsprev}(\text{node}).\text{append}(\text{index})
\]
Algorithm 29 Module:PHAD Procedure: LNGRNGINT (part-2)

\[
\begin{align*}
\text{relx}(index) & \leftarrow \text{source}_x(index) - \text{leaf}_x(node) \\
\text{rely}(index) & \leftarrow \text{source}_y(index) - \text{leaf}_y(node) \\
\text{relz}(index) & \leftarrow \text{source}_z(index) - \text{leaf}_z(node)
\end{align*}
\]

end for

\[
\begin{align*}
\text{neighbctprev}(node) & \leftarrow \text{neighbct}(node) \\
\text{neighbnumbsprev}(node) & \leftarrow \text{neighbnumbs}(node)
\end{align*}
\]

end for

for node = 1 \to dleafnodes do

\[
\begin{align*}
\text{num_of_self_sources} & \leftarrow \text{sourcect}(node) \\
\text{index} & \leftarrow \text{source numbs}(node) \\
\text{points}(1) & \leftarrow \text{relx}(index) \\
\text{points}(2) & \leftarrow \text{rely}(index) \\
\text{points}(3) & \leftarrow \text{relz}(index)
\end{align*}
\]

for \(I = 2 \to \text{num_of_self_sources} \) do

\[
\begin{align*}
\text{index} & \leftarrow \text{source numbs}(node).\text{extract}(I) \\
\text{points}(1) & \leftarrow \text{points}(1).\text{append}(\text{relx}(index)) \\
\text{points}(2) & \leftarrow \text{points}(2).\text{append}(\text{rely}(index)) \\
\text{points}(3) & \leftarrow \text{points}(3).\text{append}(\text{relz}(index))
\end{align*}
\]

end for

\[
\begin{align*}
\text{efieldl}(1) & \leftarrow -\text{der}(1, \text{locexpnode}(node)) & \triangleright \text{procedure “der” is the derivative} \\
\text{efieldl}(2) & \leftarrow -\text{der}(2, \text{locexpnode}(node)) \\
\text{efieldl}(3) & \leftarrow -\text{der}(3, \text{locexpnode}(node)) \\
\text{callpolval}(1 \text{efieldl}3 \text{points}3 \text{result}3) & \triangleright \text{procedure “polval” is polynomial evaluation}
\end{align*}
\]

for \(I \to \text{num_of_self_sources} \) do

\[
\begin{align*}
\text{index} & \leftarrow \text{source numbs}(node).\text{extract}(I + 1) \\
\text{if} \ (\text{num_of_self_sources} > 1) \ \text{then} \\
\text{checkz} & \leftarrow \text{result}(3).\text{extract}(I)
\end{align*}
\]

else if \( (\text{num_of_self_sources} = 1) \) then

\[
\begin{align*}
\text{checkez} & \leftarrow \text{result}(3)
\end{align*}
\]

end if

\[
\begin{align*}
\text{if} \ (\text{num_of_self_sources} > 1) \ \text{then} \\
\text{index} & \leftarrow \text{source numbs}(node).\text{extract}(I + 1) \\
\text{tempone} & \leftarrow \text{source}_n(index) \ast \text{momconst} \\
\text{source}_px(index) & \leftarrow \text{source}_px(index) + (\text{tempone} \ast \text{result}(i).\text{extract}(I))/\text{relg} & \triangleright \text{“relg” is the relativistic gamma} \\
\text{source}_py(index) & \leftarrow \text{source}_py(index) + (\text{tempone} \ast \text{result}(2).\text{extract}(I))/\text{relg} \\
\text{source}_pz(index) & \leftarrow \text{source}_pz(index) + (\text{tempone} \ast \text{result}(i).\text{extract}(I))
\end{align*}
\]
Algorithm 29 Module: PHAD Procedure: LNGRNGINT (part-3)

else if (num_of_self_sources = 1) then
  index ← sourcenums(node).extract(I + 1)
  tempone ← source_n(index) * momconst
  source_px(index) ← source_px(index) + (tempone * result(i))/relg
  source_py(index) ← source_py(index) + (tempone * result(2).extract(I))/relg
  source_pz(index) ← source_pz(index) + (tempone * result(i).extract(I))
end if
end for
End of procedure LNGRNGINT

Algorithm 30 Module: PHAD Procedure: COMPARE

Input: mrg:Array of Integers, sz:Integer, mrg2:Array of Integers, sz2:Integer
mrg: list of particle numbers in the current iteration index, mrg2: list of particle numbers in the previous iteration index

Initialize variables
sumsizes ← (sz + sz2 + 2)
if (sz = 0) then
  twominusone ← mrg2
  Interchange 1st element with sz2(last) element in twominusone array
end if
if (sz2 = 0) then
  oneminustwo ← mrg
  Interchange 1st element with sz(last) element in oneminustwo array
end if
ii ← 2, jj ← 2, kk ← 0, ll ← 0, sumsizes ← sz + sz2 + 2
if (sz > 0) and (sz2 > 0) then
  oneminustwo ← 0, twominusone ← 0
  check ← mrg(2)
  while (ii + jj) < (sumsizes + 1) do
    if (check = mrg2(jj)) then
      if ii < (sz + 2) then
        ii ← ii + 1
    end if
  end while
Algorithm 30 Module: PHAD Procedure: COMPARE (part-2)

if $ii < (sz + 2)$ then
    check ← mrg(ii)
end if
if $ii = (sz + 2)$ then
    $jj ← jj + 1$
    While $jj < (sz^2 + 2)$ do
        $twominusone ← twominusone$ append mrg2(jj)
        $jj ← jj + 1$
        $ll ← ll + 1$
    end while
end if
if $jj < (sz^2 + 2)$ then
    $jj ← jj + 1$
    if $jj = (sz^2 + 2)$ then
        While $ii < (sz + 2)$ do
            $oneminustwo ← oneminustwo$ append mrg2(ii)
            $kk ← kk + 1$
            $ii ← ii + 1$
        end while
    end if
else
    if ($ii < sz + 2$) and (check|mrg2(jj)) then
        $oneminustwo ← oneminustwo$ append mrg2(ii)
        $ii ← ii + 1, kk ← kk + 1$
        if $ii < (sz + 2)$ then
            check ← mrg(ii)
        end if
    end if
    if $ii = (sz + 2)$ then
        While $jj < (sz^2 + 2)$ do
            $twominusone ← twominusone$ append mrg2(jj)
            $jj ← jj + 1$
            $ll ← ll + 1$
        end while
    end if
else
    else
Algorithm 30 Module:PHAD Procedure: COMPARE (part-3)

\[
\begin{align*}
twominusone & \leftarrow twominusone \text{ append mrg(jj)} \\
ll & \leftarrow (ll + 1) \\
jj & \leftarrow (jj + 1) \\
\text{if } jj = (sz + 2) & \text{ then} \\
\quad \text{while } ii < (sz + 2) & \text{ do} \\
\quad \quad \text{oneminustwo} & \leftarrow \text{oneminustwo} \text{ append mrg(ii)} \\
\quad \quad kk & \leftarrow kk + 1 \\
\quad \quad ii & \leftarrow ii + 1 \\
\quad \text{end while} \\
\text{end if} \\
\text{end if} \\
\text{end while} \\
\text{if } (kk > 0) & \text{ then} \\
\quad \text{Interchange } 1^{st} \text{ element with } kk \text{ element in oneminustwo array} \\
\text{end if} \\
\text{if } (ll > 0) & \text{ then} \\
\quad \text{Interchange } 1^{st} \text{ element with } ll \text{ element in twominusone array} \\
\text{end if} \\
\text{compare} & \leftarrow (oneminustwo \text{ append twominusone}) \\
\text{return compare} \\
\text{End of function COMPARE}
\end{align*}
\]