Title
Discrete Element Multiphysical Models for Additive Manufacturing in conjunction with a Domain Specific Language for Computational Mechanics

Permalink
https://escholarship.org/uc/item/34n2m9bp

Author
Driver, Daniel Scott

Publication Date
2015

Peer reviewed|Thesis/dissertation
Discrete Element Multiphysical Models for Additive Manufacturing in conjunction with a Domain Specific Language for Computational Mechanics

by

Daniel Scott Driver

A dissertation submitted in partial satisfaction of the requirements for the degree of Doctor of Philosophy in Engineering-Mechanical Engineering and the Designated Emphasis in Computational and Data Science and Engineering in the Graduate Division of the University of California, Berkeley

Committee in charge:
Professor Tarek Zohdi, Chair
Professor David Dornfeld
Professor Per-Olof Persson

Fall 2015
Discrete Element Multiphysical Models for Additive Manufacturing in conjunction with a Domain Specific Language for Computational Mechanics

Copyright 2015
by
Daniel Scott Driver
Abstract

Discrete Element Multiphysical Models for Additive Manufacturing in conjunction with a Domain Specific Language for Computational Mechanics

by

Daniel Scott Driver

Doctor of Philosophy in Engineering-Mechanical Engineering
and the Designated Emphasis in Computational and Data Science and Engineering

University of California, Berkeley

Professor Tarek Zohdi, Chair

In this dissertation, two main topics will be discussed. First, a novel approach to computational mechanics via a Domain Specific Language (DSL) with a syntax to facilitate new model development will be presented. Second, discrete element multiphysical models will be proposed to study powder based additive manufacturing processes.

The DSL presented is a tool for computational mechanics which is designed to allow an engineer to focus more on model development and investigation by providing a syntax which makes it easier to define new discretizations and test different constitutive models. In order to achieve this, a system for managing data and specifying models is created around the Python development environment. There are three powerful features of the DSL. First, the language provides a syntax for creating objects which describe the fundamental physical objects and numerical elements in the problem and facilitates the allocation, management and modification of the data allocated for these objects. Second, all constitutive models are input using symbolic notation allowing for the language of math, rather than raw code, to be used to describe interactions. Third, low level code is automatically generated for the models and their gradients (via symbolic differentiation) for use with linear and non-linear solvers and time-stepping methods, both explicit and implicit. Also, Python wrappers are automatically generated so the high performance implementations can be used in the Python environment without additional work by the user.

The discrete element method is a Lagrangian technique which uses interactions of spherical elements to model material behaviors. In the past, it has been most commonly used in the study of granular media such as sands and soils. Here, extensions to the method to model heat transfer, thermal expansion and interactions with non-spherical objects will be discussed for simulating additive manufacturing processes. The formulation, implemented with the DSL, is used to study powder packing in a manner that is physically consistent with Selective Laser Sintering/Melting machines. Also, the dynamics of powders subjected to a heat source are investigated. Additional multiphysics
behavior is incorporated through a temperature dependent bonding model to create an efficient simulation for studying Laser Metal Deposition. Finally, a method for using discrete elements to enhance Eulerian finite elements to efficiently prevent advection welding is proposed.
To my family
Contents

List of Figures v
List of Tables viii

1 Introduction 1

2 Domain Specific Language (DSL) 5
   2.1 Compilers, Programming Languages and Computer Architecture . . . . . . 6
      2.1.1 Computer Architecture ........................................ 6
      2.1.2 Programming Languages ....................................... 7
   2.2 Goals of this DSL .................................................. 8
   2.3 Implementation and Syntax ........................................ 12
      2.3.1 Work Flow ..................................................... 13
      2.3.1.1 Create Data Objects and symbols ............................ 13
      2.3.1.2 Models using SymExpression ................................. 16
      2.3.1.3 Data Allocation ............................................. 18
      2.3.1.4 Connectivity And Meshes .................................... 22
      2.3.1.5 Compile to Python Module ................................... 24
   2.4 Future Considerations ............................................. 26
      2.4.1 Floating Point Error ........................................... 26
      2.4.2 High Performance Computing (HPC) Considerations .......... 27

3 Simulations with the DSL 30
   3.1 Simulation Setup Details .......................................... 30
   3.2 General Runga Kutta Methods ..................................... 32
      3.2.1 Fully Implicit ................................................ 32
      3.2.2 DIRK and Explicit Methods .................................. 34
      3.2.3 Steady State Solutions ....................................... 34
   3.3 Newton’s method .................................................. 35
   3.4 Laplace Equation Examples ....................................... 36
      3.4.1 Finite Differences ............................................ 37
      3.4.2 Finite Elements .............................................. 38
List of Figures

1.1 Left: Diagram of an SLS/SLM machine showing the powder reservoir on the right and the wiper blade which sweeps powder into the work area. The fresh powder forms a thin layer on the part and the laser sweeps over melting desired areas to form the next layer of the part. Right: Schematic of the LMD process showing the spray of powder from the nozzle which is melted by a laser (not pictured) and deposited on the workpiece. 3

3.1 Left: Ideal Simulation Module could remesh every time step. Right: The actual flow used for simulations. Remesh and flow control only happen when data is output. In both a rounded rectangle is a subroutine that is written for a simulation and called by the Sim Control module. 31

3.2 A indexing of a unit cell for finite difference. $h$ is the discretization size and is the length between 1 and 2. 37

3.3 Solution of the Poisson equation using finite differences and visualized with matplotlib. The plot is colored by $u$ and height is mapped to $u$. 38

3.4 Left: Solution of the Laplace equation with $K(u) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ Right: Same solution with elements boundaries shown. 40

3.5 Solution of the Laplace equation with $K(u) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ in solid gray. Solution to $K(u) = \begin{bmatrix} u + 1 & 0 \\ 0 & u + 1 \end{bmatrix}$ in color map corresponding to the the height, $u$. Notice the gap between the two solutions. 41

5.1 Contact Geometry 62

5.2 Finding the closest point between a sphere and a point. 63

5.3 Finding the Closest point on a line segment. 63

5.4 Finding the Closest point on a Rectangle. 64

5.5 Velocity Components of two colliding rigid bodies demonstrated on two spheres. 69

6.1 A few points in time of the evolution of the system. 76
6.2 Newton Iterations per time step for the 2-stage Fully implicit Gauss Legendre RK method (4th order). ............................................................ 76
6.3 Plots of the convergence behavior for different RK methods to integrate the DEM dynamics problem. For the $\alpha = 0$ case the implicit methods are not include because the Newton Iterations did not converge. ................. 78
6.4 Picture of sifting of Spherical particles ................................................... 80
6.5 Left: Non-spherical powder particles constructed using spherical elements and penalty constraints. Right: Particle bound up on sifting mesh created by the lines. ............................................................ 80
6.6 Left: With the perpendicular drag term on resulting in no rotation. Right: Perpendicular drag term off restoring the expected rotational behavior. . 82
6.7 Figures shows the dependence of the angle of the block in Figure 6.6 at given time as a function of the parameter $\gamma_\perp$ demonstrating the locking behavior and the strength at which it acts to prevent rotation. ............... 83

7.1 Diagram of an SLS/SLM machine showing the powder reservoir on the right and the wiper blade which sweeps powder into the work area. The fresh powder forms a thin layer on the part and the laser sweeps over melting desired areas to form the next layer of the part. .............................. 85
7.2 Dimensions of the simulation with the (0,0,0) location marked at the bottom of the work basin. ............................................................... 86
7.3 Left to Right and Top To Bottom: Particles are swept from the reservoir on the right and deposited into the work area by a soft coater blade. Three layers are deposited. Particles are colored by diameter. .................... 90
7.4 The solid fraction in the basin for Drop Packing and Wipe Packing as a function of the particle-particle tangential drag parameter. Each 'x' represents a run seeded with a different initial positions and radius. Left: Simple analysis of total powder volume/basin volume. Right: Same but volume of powder sticking over top of basin subtracted. .......................... 91
7.5 A plot showing the relative number density of elements between drop packing and wipe packing. Compared are the results for elements above and below 30$\mu$m, the work area basin drop, for a range of $\gamma_\perp$. This shows that smaller elements preferentially occur in the work basin regardless of drag parameter. .......................................................... 91
7.6 The heat source representing the laser which is represented by the purple hexagon has passed over the powder bed heating the particles. ........... 92
7.7 Left: With thermal expansion coefficient near that of Stainless Steel no particle movement is observed. Right: With a thermal expansion 10X that on the left images yields particles seen to pop up and out of the bed. .... 93
7.8 Left: Image showing the helix initial position of the elements for the deposition of a ring of material. Right: Image showing stacked lines to create the layered deposition of a wall. .............................................. 98
7.9 Figures showing representative stages in evolution of the deposition of the ring of material. The lines between elements are bonds that have formed. 99

7.10 The final state of the simulation shown in the bottom right image of Figure 7.9 without the discrete elements to show the bonding structure. 100

7.11 Left: Wall is being created by layered deposition of metal onto the top:
Right: The final wall as it falls over. The random nature of the particle size and deposition location left a defect near the base causing it to tip. 100

7.12 Snapshot of the creation of a pillar by metal deposition. The pillar in this case is much larger in radius than the stream of particles due to the relatively long time to bond compared to the flow rate. 101

7.13 Left: A single track of laser melting showing the expected denudation zone beside the melt track. Right: ALE3D simulation of a single melt tracking show the failure to reproduce this denudation zone. In the red circle, advection welded particles are not pulled in by the melt that touches them. Areas are colored by temperature. Image provided by Andy Anderson and Saad Khairallah. 102

7.14 Flow chart of a time step with the DEM enhancement for ALE simulation, in this case ALE3D. 104

7.15 Left: A enhanced ALE simulation of a solid particle hitting a solid substrate demonstrate the sticking as a result of advection welding. Right: The same simulation but enhanced with a Lagrangian discrete element. 105

A.1 Velocity Components of two colliding rigid bodies demonstrated on two spheres. 120
List of Tables

3.1 Error by newton Iteration for Linear Laplace Equations showing that solution is found to machine precision after the first iteration................. 41
3.2 Measures of convergence after each newton iteration. It is easy to see the quadratic convergence in the last 3 iterations as the exponent doubles in each iteration. After iteration 6, tolerance $(\mathbf{F}(\nabla \mathbf{F})^{-1}\mathbf{F} < 1 \times 10^{-25})$ is achieved.......................................................... 42
6.1 Convergence Rate Study. DNC stands for “Did Not Converge” in the newton iterations.......................................................... 77
7.1 Positions and directors for the 4 static rectangles which make up the divider and right hand side of the work basin......................... 86
7.2 Positions and directors 3 moving rectangles in the simulation................ 87
7.3 Positions and normals for the 6 infinite walls during the drop packing portion of the simulation.................................................. 87
7.4 Positions and normals for the 6 infinite walls during the wiping portion of the simulation.................................................. 87
B.1 The Butcher tableau key. $a$ is an $s \times s$ matrix and $c$ and $b$ are length $s$ vectors used in Runga-Kutta integration described in section 3.2........ 122
B.2 The Butcher array forward Euler.................................................. 122
B.3 The Butcher array for Explicit Midpoint........................................ 122
B.4 The Butcher array for RK3.................................................. 123
B.5 The Butcher array for RK38th.................................................. 123
B.6 The Butcher array for RK4.................................................. 123
B.7 The Butcher array for Dormand-Prince method[85]........................ 124
B.8 The Butcher array backward Euler.................................................. 124
B.9 The Butcher array backward Euler.................................................. 124
B.10 The Butcher array for Gauss Legendre 4.................................................. 125
Acknowledgments

This work was primarily funded by KAUST. This work was also partially funded by the Siemens corporation. Many thanks go to Ramesh Subramanian, Marco Brunelli, Nicolas Vortmeyer and Vinod Philip for their support. This work was also funded in part by the Army Research Laboratory through the Army High Performance Computing Research Center (cooperative agreement W911NF-07-2-0027). Many thank go to Raju Namburu for his support. This work was also funded in part by Development Program at LLNL under project tracking code 13-SI-002. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC. (Release: LLNL-PRES-676205 and LLNL-POST-675562) Many thanks go to Dan White, Andy Anderson and Saad Khairallah.

Also many thanks to my labmates with special mention of Matt Kury and Alejandro Queiruga for their invaluable contributions to my intellectual development.
Chapter 1

Introduction

Mechanical engineering is an extensive and interdisciplinary field of which a core interest is the topic of manufacturing. Manufacturing may be loosely defined as the production from raw materials of an item with a desired form or function. Deceptively simple in this definition is the concept of an item. In engineering, an item is distinguished from another item by a set of measurable properties of the item referred to as specifications. Examples of specifications are the length between two points on the item, the roughness of a surface or how much weight the item should hold when mounted in a particular fashion without breaking. Manufacturing plays an important role in all stages of the creation of an item because it is the step in the process that actually takes an idea and makes it into a reality. While an item is designed and specifications chosen, considerations to the limitations of current manufacturing technology must be considered. After an item is produced, if it fails to function as expected it must be determined if the item as specified is fundamentally incapable of fulfilling the desired function or if incorrect manufacturing led to an item that did not meet the specifications as requested.

The science of engineering for manufacturing can be described as the ability to systematically analyze and predict the result of the production process and determine if and how an item can be made to match the specifications. During a manufacturing process, materials may change phases (melt, evaporate, solidify), be cut, or bonded together, or be subjected to extreme rates of deformation and temperatures in the pursuit of creating a part. Engineering scientists have studied the underlying physics and behavior of materials in a wide range of scenarios so that they may more accurately understand and predict the properties of a manufactured product and how it will behave when used. Experimentalists and theorist work to develop mathematical models which are universal in their predictive ability. Some such laws includes Newton’s second law

$$\frac{dP}{dt} = F$$

or Fourier heat conduction model to find the change of temperature in a body over time,

$$\rho c \frac{du}{dt} = \nabla \cdot (K \nabla u).$$
By solving appropriate mathematical equations which describe the physical world, predictions about the result of a manufacturing process can be obtained. Solving the complicated set of equations that model the physics of a particular system such as a manufacturing process usually cannot be done analytically. Numerical methods are a means of finding approximate solutions of mathematical models such as those exemplified in equations 1.1 and 1.2. Many numerical methods have been invented to find approximate solutions to the mathematical models such as Finite Elements, Discrete Elements, Smooth Particle, Lattice Boltzmann, Molecular Dynamics, Moving Least Squares and Finite Differences to name just a few along with the plethora of variations on these general categories which improve the methods for certain problems. Computers are particularly capable to performing the computations required to evaluate the equations in these numerical methods.

Computational mechanics is the development and use of mathematical models, solved with computers, to find solutions to engineering problems. Computational mechanics has been applied very successfully to a wide range of engineering problems such as optimal geometries for tires [20, 24, 103, 32, 110], Bio-mechanics [44, 79], material design [113, 114] and heat transfer [16, 8] and much much more. Of particular interest to the author are applications of computational mechanics to the solution of problems pertaining to manufacturing. Computational mechanics has been used previously to study various topics in manufacturing including casting [108, 38, 42, 58], printing [84, 83, 17], cutting operations [57, 72, 39] and forming [51, 102, 88]. The increasing power of computers and the invention of better numerical methods have allowed for larger systems and more physics to be solved simultaneously leading to increasingly accurate analysis and predictions of the outcome of manufacturing processes and the behavior of the resulting product.

The triple bottom line[2] is a term coined to represent drivers of innovation which include economic, social and environmental concerns. These compel companies to develop products that are cheaper, more environmentally friendly and more to the liking of the purchasing public. To compete against other companies, new products are constantly being invented which often push the limits of current manufacturing understanding. In order to fulfill these more demanding specifications, new manufacturing methods and methods of analysis are invented. Additive manufacturing (AM) is a burgeoning area of manufacturing which promises to allow for the creation of parts beyond the reach of current manufacturing methods. In AM, a part is created by assembling small units of raw material to build up a final part, in contrasted to more traditional subtractive manufacturing methods where material is removed from a large section of raw material to form the final part. Of particular interest are the processes known as Selective Laser Sintering (SLS), Selective Laser Melting (SLM) and Laser Metal Deposition (LMD) which deposit melted powder in layers to build up parts.

In an SLS or SLM machine, represented in Figure 1.1, a part is built up layer by layer by using a laser to melt or sinter metal powder particles together. Initially, all the metal power is in the reservoir basin and none in the work area. To deposit a thin layer in the work area, first, a piston under the reservoir will rise a small amount exposing a layer of powder and the work area piston moves down(usually about 30-60µm) to receive the
powder. Next, a wiper blade moves across the machine and moves the exposed powder in the reservoir over to the work area a spreads a uniform layer filling the void that is created by the work area piston moving down. To ensure complete filling, more powder is used than expected and excess is pushed off the edge an into a collection basin. A laser then sweeps across the new layer of powder fusing it adjacent powder particles and the layer underneath it to form a layer of the part. The process repeats to build up the part layer by layer. SLS and SLM are distinguished by the temperature the powders achieve. Sintering is when just enough heat is applied to melt the outer layer to allow for bonding but does not change the powder shape significantly. With SLM, the powder is fully melted allowing the material to flow and fill the volume around it. Laser Metal Deposition is also a powder based method but powder is only applied at the sites where material will be in the next layer. This is done by a nozzle which sprays powder towards the work piece. A laser either melts the powder en route or heats the substrate enough to melt powder on impact, which then adheres to the surface, cools and solidifies to form the next layer of the part. The nozzle and laser move in a pattern to form the different geometries for each layer.

The physics of these processes span many regimes including granular dynamics of the unmelted powder, phase change as the particles melt, solidification and evaporation, gas/fluid dynamics of the ablated/melted materials and heat transfer. Also, a laser is often used to heat the powder adding further physical regimes to the problem. The disparate range of the multiphysics in the problem present a number of issues for the simulating these additive manufacturing techniques. Methods such as finite element method have shown good results good for studying the meso-scale physics of these process in detail,
they are too computationally expensive, taking on the order of 100,000 core hours [59],
to be used in the design and manufacture of an individual part.

This thesis addresses the goals of engineering and computational mechanics in two
ways. First, the Domain Specific Language (DSL) with syntax tailored to the for com-
putational mechanics will be presented. The DSL introduces a novel coding paradigm
which significantly reduces the difficulty of developing and testing new numerical meth-
ods and constitutive models. The paradigm describes a high level syntax which naturally
maps to computational mechanics concepts and is meant to be interacted with easily in a
scripting language such as Python but results in low level compiled functions that at the
machine level are executed. This produces a unique combination of usability and high
performance. The language is specifically designed to automatically produce a lineariza-
tion \( \nabla y \mathbf{F}(y, p, c) \) of complex functions for use in steady state solutions or implicit time
stepping without any additional description of the physical system. The language is also
designed to produce code in low level languages such as C or Fortran, so that existing
powerful libraries for these languages can be utilized by the DSL compiler. The DSL
system is a step towards an implementation-agnostic language for describing engineering
problems with rigorous and general definitions of physical concepts.

Second, this powerful tool will be applied to develop multiphysics discrete element
models for powder based additive manufacturing processes which are computationally ef-
cient, robust and easily extended to other multiphysics problems. The discrete element
methods (DEM) is generally attributed to Cundall and Strack [33] which is originally in-
tended to model the interaction of granular material where each spherical element directly
mapped to one grain in the physical system. The methods has been widely applied to
simulations of soil [95, 98, 69, 54], sieving [36, 86, 30] and mixing [6, 13, 14, 80] among
other applications. Here, discrete element framework is extended to model the powders in
the additive manufacturing systems to represent phase change, bonding and heat trans-
fer while retaining the original character of the method that the elements are spherical
and all properties of the element are homogeneous. The framework provides a reduced
order system which still provides a rich set of multiphysics which can be modeled while
remaining computationally efficient.

This dissertation will be presented in the following manner. Chapter 2 will describe
the DSL language and Chapter 3 will present additional information about the use of the
system for general simulations. Chapter 4 will present a general mathematical descrip-
tion of the multiphysics discrete elements and Chapter 5 will present specific constitutive
models used for the interaction between elements. Finally, in Chapter 6 numerical char-
acteristics of the discrete element model will be investigated and in Chapter 7, the models
will be applied to problems relating to powder based additive manufacturing methods.
Chapter 2

Domain Specific Language (DSL)

The goal of the DSL is to allow the user to easily describe a physical system, both linear and especially nonlinear, and avoid the boiler-plate activities of taking derivatives and writing C or Fortran Code directly. At same time, the goal is to do this without significant loss of efficiency in terms of computation time with some loss being acceptable as a trade off for a significant reduction in development time. The general idea is that the user provides a description of the problem in a high level language and a high performance routine is produced, in this case by writing C-code to be compiled and then called back in the high level language. The paradigm makes it easy to combine the best features of both systems, the high performance of a compiled language with the multitude of very convenient features and libraries available in an interpreted language. The idea of low level, high performance code made easily accessible via bindings to Python [70, 74] was first introduced to me by my labmate Dr. Alejandro F. Queiruga. Further inspiration came in the form of FEniCS [65, 66, 67] which is a powerful finite element package that will produce low level compiled code with Python bindings based on weak forms described in Python. FEniCS however is built around the finite element formulation with predefined elements solving a continuum problem on a mesh. The high level features that make it easy to set up and solve a finite element problem also make it difficult to easily describe different methods within the FEniCS framework. The DSL presented in this dissertation is targeted towards method development by providing a syntax that is designed to make the creation of new elements, constitutive behaviors and interaction behaviors while still relieving the developer of the duties of linearization and low level programming. The trade off for the increased generality is that the user must provide more description for the meaning of data and their interactions than is required for FEniCS.
2.1 Compilers, Programming Languages and Computer Architecture

In one interpretation, the DSL in this dissertation can be thought of as a library which adds to the functionality of existing languages, but this is a very limiting interpretation of its functionality. It is much better thought of as a Domain Specific Language: a syntax defined for a specific purpose and is not necessarily a complete programming language. Describing it as a language emphasizes the fact that the meaning of the syntax can be understand independent from its implementation opening up many more possibilities. In this section, a brief overview of the very basics of computer architecture and the meaning of a programming language will be presented which will a establish a common set of terminology to understand the description of the DSL in subsequent sections as a language and not just a library.

2.1.1 Computer Architecture

A modern computer in its most simplistic representation consists of a Central Processing Unit (CPU) and memory. In a Von Neumann architecture, the memory is thought of as having two sections, instruction memory and data memory [97, 73, 101]. The data memory as its name suggest stores data to be manipulated. The CPU is a device that accepts data and an instruction. The instruction indicates which piece of data should be modified and how it should be modified. The instruction memory stores stores a preset list which defines many operations of finding and modifying data.

The analogy I like best for a modern computer is baking. Here the person is the CPU, the recipes is the instruction set, and the ingredients are the data. The person has many operations it can preform such as stirring, adding, put in oven, take out of oven but no knowledge in what order to do those operations. A recipe stores what the person should do and to what ingredients. For example, a recipe tells the person to perform the list of operations in order:

1. add flower
2. add water
3. stir
4. add egg
5. stir
6. put in oven
7. wait 45 minutes
By changing the recipe and the ingredients a different baked good is produced. This is the same for the CPU but now the ingredients are numbers and the operations are algebraic such as add, subtract, multiply and divide etc. To get the computer to do anything, the recipe, referred to as machine code, has to be created. Machine code is set of binary numbers which represent the operations such as add and subtract and what piece of data on which the operation is to be performed.

2.1.2 Programming Languages

In the early days of computers, people wrote the machine code by hand. Very quickly however, as computer programs became larger and more complex and different hardware was made, this became untenable. To combat this, as time progressed more human friendly ways of producing the machine code for the computer were invented starting with systems such as Assembly and moving to C and Fortran. With these “programming” languages a human friendly syntax is defined that allows a person to more naturally describe what they want the computer to do. A “compiler” is then responsible for reading a text file which contains C or Fortran syntax and turning it into machine code, a recipe, for the computer. The machine code can then be put in the computer’s instruction memory and executed, the result being that data is manipulated and a useful calculation is performed.

It is important to distinguish between the programming language as a standard and the compiler. A programming language is merely a document which defines a syntax and expected behavior. The language in itself is not capable of doing anything, this is a key point. For the language to be useful a document containing a program described with the languages’ “code”, needs to be read and converted to machine code the computer can understand. This is a compiler. The compiler could consist of a smart person that reads a file with code, for example C-code, and manually translates that into machine code but it is much more common that another program is created to run on the computer which is capable of reading the text file and automatically generating the machine code. As long has both systems read the syntax and produce the expected behavior as defined by the language, they are both valid implementations of a compiler for the language standard. The task of turning the high level concepts the person describes into the most efficient/fastest machine code is very difficult. As such, many compilers, such as gcc[4], Intel compilers[3] and clang[1], have been created for trying to turn C-code into the most efficient machine code. All should produce machine code that leads to the same answer at the end of the computation, though some may take longer to get to that correct answer.

While C and Fortran were once the best systems for describing a program to a computer, as time progressed new technologies where invented to make the process easier for certain tasks. These new languages such as Python were designed to make it even easier for a human to interact with their computer. Python, which is used in this dissertation, is
a language that defines short syntax for very complex operations and is very fluid in that it allows data to function as any type at any moment, which it automatically manages. The extra ease tends to lead to slower computation as a language like Python must also be constantly checking other aspects such as data type and selecting correct operations instead of just blindly crunching numbers.

2.2 Goals of this DSL

Despite the advent of more human friendly languages, “low level” languages such as C and Fortran are still used particularly in applications where performance is important. Computational mechanics is a field where simulating a system of interest requires so many operations that performance considerations must be addressed.

Because low level languages must be used in computational engineering, the modern advancements in productivity and ease have not been accessible to the field. The DSL is an attempt to solve this by defining a syntax specifically designed for computational engineering; a language designed for specific task is referred to as Domain Specific Languages. At this point, it is again important to be reminded of the distinction between a language and the compiler. The syntax for this DSL defines standalone concepts for data objects, equations and variables. In theory, this syntax can compiled into machine code by any program which parses the syntax correctly and produces the correct results.

In practice, the syntax is designed to be implemented and used with a scripting language. In this case the author has created one implementation of the DSL compiler in Python. The reason for this is that the DSL language defines no “main” program. The “main” program in a language is the top level item that is called that is responsible for starting and stopping a program or simulation. The language is designed only to setup up vector valued functions of the form \( F(y) \colon \mathbb{R}^n \to \mathbb{R}^m \) and compile machine code for both of \( F(y) \) and \( \nabla_y F(y) \colon \mathbb{R}^n \to \mathbb{R}^n \times \mathbb{R}^m \), which can in turn be called in the original scripting language for ease of use. Here \( y \) is a vector which represents all the degrees of freedom in a system. The inputs to a generic function are more typically denoted \( x \), but this notation is used in an attempt to avoid confusion with the notation for position.

So what about \( F(y) \) and the automatic production of its gradient warrants the dedication of its own syntax and language? The function is a basic item in all computational operations seen in numerical methods for computational engineering and the gradient often is used when solving nonlinear problems, both steady state and dynamic. In computational engineering, the function contains all the information about the discretized system and is called, for example, in every stage of a Runga-Kutta time stepping method. The gradient is used with implicit methods and solving static problems both for linear and non-linear problems.

Using the dynamics of a point mass as a first example to demonstrates this, the coupled
differential equation for dynamics are
\[
\frac{dv}{dt} = \frac{\text{Force}(x, v)}{M}, \\
\frac{dx}{dt} = v,
\]
and are rewritten by substituting
\[
y = \left( \begin{array}{c} x \\ v \end{array} \right), \\
F(y) = \left( \begin{array}{c} \frac{\text{Force}(x, v)}{M} \\ v \end{array} \right)
\]
yielding the compact form
\[
\frac{dy}{dt} = F(y).
\] (2.1)

If is desirable to integrate in time, one might use a simple scheme like forward Euler. This is a first order method finds an approximate solution for \( y \) at a time \( h \) in the future denote \( y^{n+1} \) based off the current state of \( y \) denoted \( y^n \), which is accomplished via the update formula

\[
y^{n+1} = y^n + h F(y)|_{y=y^n}.
\]

This simple integration scheme merely requires an evaluation of a function of the form \( F(y)|\mathbb{R}^n \rightarrow \mathbb{R}^n \) multiplied by \( h \) and added to the current state.

Another method for integrating equation 2.1 is the implicit midpoint method. Again the goal is to find \( y \) at a time \( h \) in the future denote \( y^{n+1} \) based off the current state of \( y \) denoted \( y^n \). The update in this formula is

\[
y^{n+1} = y^n + h \left( \frac{1}{2} (F(y)|_{y=y^n} + F(y)|_{y=y^{n+1}}) \right)
\]

where this method requires two evaluations of \( F \) but one of them is at time \( h \). Again, the \( F \) takes center stage but \( y^{n+1} \) is not known so it would seem that it is impossible to evaluate the right hand side. One way to find \( y^{n+1} \) is iteratively via Newton’s method which solves problems of the form \( F'(y^{n+1}) = 0 \), finding the zeros of a function. This is done with the update scheme

\[
y^{n+1,i+1} = y^{n+1,i} - \left( \nabla y^{n+1} F'(y^{n+1})|_{y^{n+1}=y^{n,i}} \right)^{-1} F'(y^{n+1})|_{y^{n+1}=y^{n,i}}
\] (2.2)

where \( y^{n+1,0} \) is an initial guess and the iteration is run until it converges (more details of this will be discussed in section 3.2.1). This is done by first rephrasing the problem as a root finding problem by writing

\( F' \) is another function and does not indicate a derivative. The prime only indicates it is different from function \( F \) which is reserved for the function that represents the problem physics.
\[ F'(y^{n+1}) = y^{n+1} - \frac{h}{2}(F(y)|_{y=y^n} - F(y)|_{y=y^{n+1}}) = 0 \] \hspace{1cm} (2.3)

and then plugging in 2.3 into 2.2 to get

\[
y^{n+1,i+1} = y^{n+1,i} - \left( \nabla y^{n+1} \left[ y^{n+1} - \frac{h}{2}(F(y)|_{y=y^n} - F(y)|_{y=y^{n+1}}) \right] \right)_{y^{n+1}=y^{n+1,i}}^{-1} F' (y^{n+1})|_{y^{n+1}=y^{n+1,i}}
\]
\[
y^{n+1,i} = y^{n+1,i} - \left( I - \frac{h}{2} \nabla y F(y)|_{y=y^{n+1,i}} \right)^{-1} F' (y^{n+1})|_{y^{n+1}=y^{n+1,i}}
\]
\[
y^{n+1,i} = y^{n+1,i} - \left( I - \frac{h}{2} \nabla y F(y)|_{y=y^{n+1,i}} \right)^{-1} \left[ y^{n} + \frac{h}{2} (F(y)|_{y=y^n} + F(y)|_{y=y^{n+1}}) \right]_{y^{n+1}=y^{n+1,i}}.
\]

From this it starts to become clear that for time integration \( F(y) \) and \( \nabla y F(y) \) are needed.

Finite elements can also be phrased in the same way. For example, if the problem to be solved were the steady state heat equation, the strong form would be written as

\[ \nabla_x \cdot [K(u(u))\nabla_x u(u)] = 0 \]

where \( u \) is a vector of nodal temperature values and scalar function \( u \) is the temperature calculated as the sum of shape functions \( \phi_j, u(u) = \sum u_j \phi_j(x) \), and \( K \) is the conductivity matrix that may be a function of \( u \). Using the Galerkin method by multiplying a test function composed of the shape functions used in the interpolation, \( v = v^T \phi \) where \( v \) are arbitrary coefficients and \( \phi \) is a vector functions \( \phi_j \), and weakening the system gives the result

\[
\int_\Omega \nabla_x \cdot [K(u(u))\nabla_x u(u)] \, v dx = 0 \forall v
\]

\[
\int_\Omega \nabla_x \cdot [K(u(u))\nabla_x u(u)] - K(u(u))\nabla_x u(u) \cdot \nabla_x v dx = 0 \forall v
\]

\[
\int_\Omega K(u(u))\nabla_x u(u) \cdot \nabla_x v dx - \int_{\partial \Omega} K(u(u))\nabla_x u(u) \cdot n v dS = 0 \forall v
\]

\[
\int_\Omega [K(u(u))\nabla_x u(u)]^T (\nabla_x \phi)^T v dx - \int_{\partial \Omega} q \cdot n (v^T \phi) dS = 0 \forall v
\]

\[
\int_\Omega v^T \nabla_x \phi [K(u(u))\nabla_x u(u)] dx - \int_{\partial \Omega} q \cdot n (v^T \phi) dS = 0 \forall v
\]

\[
v^T \left[ \int_\Omega \nabla_x \phi [K(u(u))\nabla_x u(u)] dx - \int_{\partial \Omega} q \cdot n \phi dS \right] = 0 \forall v.
\]

Then by using the assumption that the coefficients, \( v \), are arbitrary

\[
\int_\Omega \nabla_x \phi [K(u(u))\nabla_x u(u)] dx - \int_{\partial \Omega} q \cdot n \phi dS = 0 \hspace{1cm} (2.4)
\]
leading to an $F$ of the form

$$F(u) = \int_{\Omega} \nabla_x \phi [K(u) \nabla_x u(u)] \, dx - \int_{\partial \Omega} q \cdot n \phi dS.$$ \hfill (2.5)

If $v = 0$ everywhere on all at Dirichlet boundaries the

$$F(\tilde{u}) = \int_{\Omega} \nabla_x \phi [K(u(\tilde{u})) \nabla_x u(\tilde{u})] \, dx - \int_{\partial \Omega} q \cdot n \phi dS$$ \hfill (2.6)

where $\Gamma_N$ is the subset of the boundary with Neumann boundaries and $\tilde{u}$ is a shorter vector, restricted to the subset of $u$ not determined by the Dirichlet boundaries. In the special case where all boundaries are Dirichlet this becomes

$$F(\tilde{u}) = \int_{\Omega} \nabla_x \phi [K(u(\tilde{u})) \nabla_x u(\tilde{u})] \, dx.$$ \hfill (2.7)

The global stiffness matrix for a given $u$ is found by taking the gradient $K_{global} = \nabla_u F(u)$. If $F$ is linear then the $K_{global}$ will be independent of $u$ and a single matrix solve can be used to find the solution. If the system is non-linear an iterative method can be used to find the solution for $u$. In this case, Newton’s method would look like

$$u^{i+1} = u^i - (\nabla_u F(u)|_{u=u^{i+1}})^{-1} F(u)|_{u=u^i}.$$

From these examples, hopefully it has been demonstrated that having an easy way to make high performance implementations of $F(y)$ and $\nabla_y F(y)$ would be beneficial for large swaths of computational engineering from both ODE and PDE’s (which become ODE’s after they are discretized), explicit and implicit time stepping and steady state solutions. The DSL is exactly designed to produce functions of the form $F(y)$ and from that $\nabla_y F(y)$ automatically. The automatic production of the gradient serves a few purposes. First there is not chance of coding errors or upkeep of the gradient if $F$ changes. Second, getting the gradient correct for a variety of physical system, element types, and material laws can be time consuming and potentially prohibit a method’s developer from taking full advantage of the available numerical methods, rather opting for perhaps less effective explicit method merely to save development time. Third, it opens up the possibility of searching through model variations and solution techniques because the program can automatically manage large section of the code production.

Once the $F(y)$ and $\nabla_y F(y)$ are made, their use in a full simulation with details which include flow control, data input and output, boundary conditions, and time stepping schemes needs to be addressed. This will be discussed in detail in section 3.1. In the next section, a description of the current syntax and implementation of the DSL will be presented.
2.3 Implementation and Syntax

The DSL which defines a natural syntax for computational engineering to setup up vector valued functions of the form $F(y):\mathbb{R}^n \to \mathbb{R}^m$. The compiler created by the author is implemented in and used with Python. The compiler works by creating a high performance implementation of $F(x)$ and $\nabla_x F(y)$ in the C language. Python wrappers of the C functions are then automatically compiled into a shared library (*.so in Linux) which can be imported into Python. In this way, the entire setup and simulation can be performed in Python, taking advantage of its many useful data constructs and libraries for manipulating data during setup, while still achieve the performance associated with C when the simulation is run.

The DSL accomplishes this task by writing two types of C-files. First there is one file which is just the assembly routine for $F$ and/or $\nabla_y F(y)$ (options are provided so that only necessary functions are written). The assembly routine is first responsible for looping over the connectivity data and collecting data to pass to the model subroutines. Second, the model subroutines are functions written into separate files which are responsible for calculating the sub-parts of the function $F$ (similar to elemental contributions in finite elements), such as the constitutive model for Hertzian contact, or the integration over a finite element. These are standalone functions where calculate their contribution to the $F$ and/or $\nabla_y F(y)$. The assembly routine then takes the return of the model functions and adds them to the appropriate spots in larger $F$ and/or $\nabla_y F(y)$. The DSL syntax in Python defines what data will be passed to the functions and the place it is supposed to be put in $F$ and/or $\nabla_y F(y)$. Based on the syntax all indexing is automatically handled and written into the assembly routine. The Scipy routine Weave is then used to automatically generate wrappers for the assembler function. Weave is also provided with the header files (also automatically created) and C-files for the model functions which it then can use to call the gcc compiler to generate the Python callable shared library which can be called to execute $F$ and/or $\nabla_y F(y)$.

It should be noted at this time that the DSL does not exactly make function of the form $F(y)$ but rather $F(y, p, c)$ where $y$ is the same as before and stores degrees of freedom in the problem which will need to be solved for. For know this is an 1-D array of doubles. $p$ stand for Parameters, this is additional data that is not solved for but might change between or during a run but is not hardcoded into the function and also is a 1-D array of doubles. Finally, $c$ is a 1-D array of integers which stores all of the connectivity data used during assembly. By separating the data into these three types makes it easy to manage the simulation in the Python layer without recompiling if desired. For example, one might make a parameter that is the speed of wiper blade or laser is moving is the simulation. One might want to study the optimal time to start moving the object. At run time, the parameter can start with a 0 velocity and then be given a non-zero velocity at a quarter, a half and three quarters through the simulation. By leaving the velocity in the parameters set these three cases can be examined without recompilations. In this way scripting and parameters studies can be managed in the much more convenient Python
layer. Also, separating the connectivity out allows the connectivity to be modified at runtime allowing for remeshing to occur at runtime without recompilation if needed.

Once a function and its gradient are created, they can be used to solve for static and dynamics systems both linear and non-linear. A separate library is maintained which contains implementations of solvers and integrators. The DSL is kept separate from this as the creation of the function and the gradient is a sufficiently general and useful operation that it is advantageous to keep it delineated both mentally and in practice from the solvers. One major reason for this is that there are many powerful libraries in existence such as PETSc \([12, 10]\) in C and Scipy in Python which already provide implementations for solvers and integrators which take a function and gradient (or pseudo-gradient) as input. The function production is kept separate with this in mind. Another reason is that the DSL is designed to be used for the parts of the simulation setup which change, i.e. physics models and assembly. The solvers and integrators, at least for the time being, do not change in implementation and thus do not need a system for writing them in a general sense. People do write auto-tuners and code generation for making efficient solvers, such as ATLAS \([104, 105]\), but that is not within the scope of this dissertation.

To parse the the DSL syntax Python is used. All syntax is designed to be valid Python code so parsing is actually just a matter of running it through a standard Python interpreter. When the syntax is run through the compiler, Python classes are created and their methods called which in turn implement the code generation. At the moment the DSL is still very much in development but it has reached a point that all of the work in this dissertation, except the last section 7.4. With the understanding that the syntax is still evolving, presented is the current state and where known problems exist they will be noted along with the suggested future improvement.

2.3.1 Work Flow

2.3.1.1 Create Data Objects and symbols

The first step in using the DSL is to create “Thing”s with the DataMaker Module. A “Thing” is an object which has a name and a description of the data the is associated to it. A Thing might represent, for example, a discrete element and have data for a position, velocity, radius and temperature, or a quadrilateral finite element with connectivity data or a bond with connectivity data and bond data liken an equilibrium length. A Thing object is created with the syntax “ThingObj=Thing(name)” where name is the name which will be associated to it in the data arrays. Data is added with the ‘AddData’ method using ThingObj.AddData(DataName,size,type). DataName is a string which is name of the data and will have ‘size’ number of doubles associated to it. Type can be ‘State’, ‘Parameter’ or ‘Connectivity’, corresponding to the data’s position in \(\mathbf{y}, \mathbf{p}\) or \(\mathbf{c}\) respectively. AddData automatically adds DataName to the namespace of ThingObj so ThingObj.DataName is now valid syntax. ThingObj.DataName.AddSubGroup(ComponentName,size) can be used to assign names to the data in DataName. The data array \(\mathbf{y}\) when allocated will
have in its namespace \( y.name.DataName.ComponentName \) which will return the data associated to the ComponentName.

**Algorithm 2.1** Example of making a particle “Thing” with the DSL. The Particle object is created with the name 'Particle' and has 7 doubles associated to it the state array \( y \) and 1 double in the Parameter array \( p \). AddData makes the data and AddSubGroup adds a name space and segmentation of the data.

SpatialDOF=3  
Particle=danplusplus.DataMaker.Thing('Particle')  
Particle.AddData('pos', SpatialDOF, 'State')  
Particle.pos.AddSubGroup('x',1)  
Particle.pos.AddSubGroup('y',1)  
Particle.pos.AddSubGroup('z',1)  

Particle.AddData('vel', SpatialDOF, 'State')  
Particle.vel.AddSubGroup('x',1)  
Particle.vel.AddSubGroup('y',1)  
Particle.vel.AddSubGroup('z',1)

Particle.AddData('Temp', 1, 'State')  
Particle.AddData('radT0', 1, 'Parameter')

Algorithm 2.1 shows an example of the creation of a discrete element in 3 dimensions with position, velocity and temperature degrees of freedom. First, the Particle 'Thing' is created with the DataMaker module and function Thing with the name 'Particle'. Next 3 doubles are added to \( y \), indicated by the keyword 'State' input, with the name 'pos'. One feature here is that as soon as Data is added using the AddData method, the 'Thing' object has 'pos' added to its namespace so that section of data can be easily and naturally referenced. The 'pos' data is given further meaning with AddSubGroup functions which name the 3 pieces of data in 'pos' to be 'x', 'y' and 'z', the components of the position. This is repeated for the velocity, 'vel', and the temperature, 'Temp'. Finally, the radius at reference temperature \( T_0 \) is created with the AddData function this time with the keyword 'Parameter' indicating it should be allocated into \( p \).

It is also convenient to make the matching SymPy[91] symbols at this time to go with this Thing object, the symbols will be used later to express functional forms of the models. Algorithm 2.2 shows the SymPy symbol creation to match the data described in the Particle “Thing” in Algorithm 2.1. SymPy symbols are made to match the position and velocity components along the the symbols for the temperature and the radius parameter. Components are grouped into symbolic 1-D Matrices(aka vectors) to match position and velocity data designations in Thing. Two sets of symbols are made for expressing particle-particle interactions.
Algorithm 2.2  Sympy symbol creation to match the data described in the Particle “Thing” in Algorithm 2.1. Sympy symbols are made to match the position and velocity components along the the symbols for the temperature and the radius parameter. Components are grouped into symbolic 1-D Matrices(aka vectors) to match position and velocity data designations in Thing. Two sets of symbols are made for expressing particle-particle interactions. The ‘real’ flag is set to ‘True’ indicating these values should be real numbers for symbolic manipulation.

\[
x_{1}, y_{1}, z_{1} = \text{sympy.symbols('x1 y1 z1', real=True)}
\]

\[
vx_{1}, vy_{1}, vz_{1} = \text{sympy.symbols('vx1 vy1 vz1', real=True)}
\]

\[
r_{1} = \text{sympy.ImmutableMatrix([x1, y1, z1])}
\]

\[
v_{1} = \text{sympy ImmutableMatrix([vx1, vy1, vz1])}
\]

\[
x_{2}, y_{2}, z_{2} = \text{sympy.symbols('x2 y2 z2', real=True)}
\]

\[
vx_{2}, vy_{2}, vz_{2} = \text{sympy.symbols('vx2 vy2 vz2', real=True)}
\]

\[
r_{2} = \text{sympy ImmutableMatrix([x2, y2, z2])}
\]

\[
v_{2} = \text{sympy ImmutableMatrix([vx2, vy2, vz2])}
\]

Finally, an example of a Thing to represent a bond is shown in Algorithm 2.3. Here the data object stores connectivity data and will result in 2 integers associated to it in the c array, indicated by the keyword ‘Connectivity’. This will store which two particles are bonded. The Bond also has a BondStatus Parameter which could be used to store a damage parameter for the bond.

Algorithm 2.3 Another example showing and example of how one might make a bond data object. The bond has two connectivity data to store which other “Things” are connected and state used to store information about the bond. Also included in this example is the creation of the symbol to go with the BondStatus Data.

\[
\text{PPPairBond}=\text{danplusplus.DataMaker.Thing('PPPairBond')}
\]

\[
\text{PPPairBond.AddData('PairBond', 2, 'Connectivity')}
\]

\[
\text{PPPairBond.AddData('BondStatus', 1, 'State')}
\]

\[
\text{PPPairBondStatus=\text{sympy.symbols('PPPairBondStatus', real=True)}}
\]

Symbols, like those in Algorithms 2.2 and 2.3, will be used to define the functional form of F symbolically in the next section and the Thing objects themselves will be used in the allocation of data arrays y, p, c, to segment those arrays, and associate the segments with namespaces in Python. An obvious improvement here is the addition of a method
to the Thing object that produces a set of symbols to match the object. This would do away with the annoying step of having to manually create all the symbols. Later, it will be seen that there is another step required to associate symbols to data. If the symbols were created by the Thing object the association between symbol and data could also be done automatically further saving the user time and reducing the potential for errors.

2.3.1.2 Models using SymExpression

In this step, the functional form of the models are created by operating on the SymPy symbols. By using the SymPy symbols, all the benefits of the SymPy library can be used which defines matrix multiplication, cross products, norms and most importantly derivatives, including an implementation of the Jacobian operator. The operations make it very easy to setup up functions but also makes it possible to automatically get derivatives of the functions via symbolic differentiation. The final form of the functional expression is stored in another object called SymExpression("symbolic expression").

A SymExpression is create by the SymObj=SymExpression(f,variables,parameters). f is a vector symbolic expression which represents a physical model usually corresponding to an element contribution, \( F_{elem} \), to \( F \). The input ‘variables’ and ‘parameters’ are each a list of symbols that appear in f. The symbols in ‘variable’ are symbols corresponding to degrees of freedom with data in \( y \). This list is which symbols will used to find the Jacobian of f to contribute \( \nabla_y F(y) \). The ‘parameters’ input has the rest of the symbols and will not be include in the gradient calculation. The list of symbols are used in the assembly step to determine which data needs to be accessed in the \( y \) and \( p \) arrays and passed into the model function. In the implementation, SymExpression has a method MakeCFile which is called during parsing which is writes a C-file which implements sub-part of the function \( F \) in C code.

The designation of the variables and parameters may seem at first redundant to the State and Parameter designation of the data objects. This definitely true if the problem is being solved monolithically (all \( y \) are solved for simultaneously) but if the state is solved by partitioning some \( y \) will be held constant and effectively will be parameters and would not want them in the gradient. The partitioning technique is commonly used and allows different solution techniques for different parts of the physics e.g. electromagnetic, dynamics, thermal. With the current implementation this approach can be done by making multiple \( F \)s, one for each partition. Data will be state data for some partitions of \( F \), but in parameter array \( p \) in others. The variables and parameters lists for SymExpression must reflect this. At this time the author does not have a solution that would supply and easy and general syntax to express partitioning. One solution might be to avoid creating two separate arrays for \( y \) and \( p \) so that the distinction is less important making it easier to switch its role in different partitions. This creates difficulties when solving though and is left future work.

An example of the creation of a symbolic expression and its inclusion in a SymExpression is shown in Algorithm 2.4 with the generated model function if Algorithm 2.5 and
gradient in Algorithm 2.6. The code in Algorithm 2.4 implements the right hand side of the ODE for motion of a single element without rotational degrees of freedom below

\[ \frac{dx}{dt} = \mathbf{v} \]
\[ \frac{d\mathbf{v}}{dt} = \frac{F_{\text{force}}(\mathbf{x}, \mathbf{v})}{M} \]  \hspace{1cm} (2.8)

where \( F_{\text{force}} = F_{\text{grav}} + F_{\text{PowerDrag}} = -Mg - \gamma \| \mathbf{v} \|^{\alpha-1} \mathbf{v} \). The result is that the functional form

\[ f_{\text{UpdateGrav}} = \left( \frac{\mathbf{v}}{F_{\text{force}}(\mathbf{x}, \mathbf{v})} \right) = \begin{pmatrix} \frac{v_x}{M} \\ \frac{v_y}{M} \\ \frac{v_z}{M} \\ -\gamma \| \mathbf{v} \|^{\alpha-1} \frac{v_x}{M} \\ -\gamma \| \mathbf{v} \|^{\alpha-1} \frac{v_y}{M} \\ -\gamma \| \mathbf{v} \|^{\alpha-1} \frac{v_z}{M - g} \end{pmatrix} \]

The assignment to the desired ODE will be done in a later step. \( \alpha, \gamma, g, M \) are model parameters set by the user which are not solved for and thus are in the parameters list. The variables that will be solved for are the velocity, \( \mathbf{v} \), which the gradient will be taken with respect to. The variables, parameters and symbolic model expression are then input to make a SymExpression to be assigned later to systems of equations. SymPy mostly handles code generation for the symbolic systems, however, it does it based on the names given to the symbols in the previous step. A uniform function structure is used with three pointers, two inputs \( X \) and \( \text{Par} \) and one output \( F \) or \( \nabla F \). The DSL is responsible for automatically indexing \( X \) and \( \text{Par} \) and assigning them to the correct symbol names used by the SymPy generated functions. The order of data in the input arrays is dictated by the order of the symbols in the list passed to SymExpression as input. When used with the assembly routines built into the DSL, the inputs \( X \) and \( \text{Par} \) will be generated for input into the model function.

An interesting aspect of Algorithm 2.4 out here is the use of the function PowerDrag. Because Python is entirely dynamically typed and the operators used in it are overloaded already for symbols and for doubles, it is possible to use this exact same function for operating on symbols, doubles or a combination. This means the exact same code can be reused for both purposes virtually eliminating the possibility of bugs that might occur when modifying a code but failing to propagate the change throughout the entirety of a code. This feature is very useful for post processing results to recover components of the force for plotting or analysis.

SymExpressions can also be used independently of the assembler and implements the methods MakeCFile and MakePythonModule. MakeCFile will write the corresponding C code for the element model denoted, \( F_{\text{elem}} \) and gradient \( \nabla F_{\text{elem}} \). MakePythonModule will write the C-code and a Python wrappers and make a shared library callable in Python for \( F_{\text{elem}} \) and \( \nabla F_{\text{elem}} \). This can be useful for testing models or writing new elements for
Algorithm 2.4 The DSL implementation of model for a particle under gravity with drag.

```python
def PowerDrag(GammaEff, alpha, speed, v):
    """force is in the opposite direction of v""
    return -GammaEff*speed**(alpha-1)*v

F_grav=DD.M1*sympy.ImmutableMatrix([0, 0, -DD.g])
speed=v.norm()
fdrag=PowerDrag(DD.Gamma,DD.Alpha, speed,DD.v1)
f=F_grav+fdrag
fUpdateGrav= symlink. ImmutableMatrix([DD.v1, f/DD.M1])

variables=sympy.ImmutableMatrix([[DD.v1]])
parameters=sympy.ImmutableMatrix([[DD.g],[DD.Gamma],[DD.Alpha],[DD.M1]])
PosUpdateAndGravDrag=danplusplus.SymDiff.SymExpression(fUpdateGrav, variables, parameters)
```

codes that already have assembler that allow for user input of new elements and gradient. An example of such a case is the finite element software FEAP [94].

The current problem with the use of Sympy for all setup of symbolic expression is that when operating on functions it automatically substitutes sub-expressions leading to correct but very long functional forms. As is particularly clear in the generated code in Algorithm 2.5 and 2.6. For example, the expression for the norm of the vector \((x, y, z)\) is \(\sqrt{x^2 + y^2 + z^2}\) and appears multiple times in the final expression. Because the same expression is repeatedly differentiated, the result is that it takes a long time to form the gradient. It would be better if sub-expressions were kept separate as independent nodes in the expression tree and then during differentiation the chain rule could be applied. When the code is written, it may also be beneficial for compute time to define new variables for the sub-expressions instead of plugging them in and having the code repeated many times. Whether keeping sub-expressions separate in the code would actually improve run time is unclear. gcc already has flag -cse for common subexpressions[4] and the compiler will automatically create subexpressions to improve run time so doing it explicitly at code generation may not actually improve compiled run time. More future work would need to be done to determine this though it is expected that at least code generation time would be reduced by keeping symbolically subexpressions separate.

2.3.1.3 Data Allocation

The data arrays are created using the DataAllocator Object created by DataAllocObj=DataAllocator(name) where name is a string. At this time the name input is
Algorithm 2.5 Resulting code generated for the function $F$ described by the DSL syntax in Algorithm 2.4.

```c
int PosUpdateGravDrag_f_func(double * X,
                               double * Par,
                               double * K){

double vx1=X[0];
double vy1=X[1];
double vz1=X[2];
double g=Par[0];
double Gamma=Par[1];
double Alpha=Par[2];
double M1=Par[3];
    F[0]=1.0*vx1;
    F[1]=1.0*vy1;
    F[2]=1.0*vz1;
    F[3]=1.0-Gamma*vx1*pow(pow(vx1, 2) + pow(vy1, 2) + pow(vz1, 2), Alpha - 1.0L/2.0L)/M1;
    F[4]=1.0-Gamma*vy1*pow(pow(vx1, 2) + pow(vy1, 2) + pow(vz1, 2), Alpha - 1.0L/2.0L)/M1;
    F[5]=1.0-Gamma*vz1*pow(pow(vx1, 2) + pow(vy1, 2) + pow(vz1, 2), Alpha - 1.0L/2.0L) - M1*g)/M1;
return 0;
}
```
Algorithm 2.6 Resulting code generated for the function $\nabla F$ described by the DSL syntax in Algorithm 2.4.

```c
int PosUpdateGravDrag_gradf(double * X,
                             double * Par,
                             double * K){
    double vx1=X[0]; double vy1=X[1]; double vz1=X[2];
    double g=Par[0];
    double Gamma=Par[1];
    double Alpha=Par[2];
    double M1 =Par[3];
    K[9]=−2.0*Gamma*pow(vx1, 2)*((1.0L/2.0L)*Alpha − 1.0L/2.0L)
    *pow(pow(vx1, 2)+ pow(vy1, 2) + pow(vz1, 2), (1.0L/2.0L)*Alpha
    − 1.0L/2.0L)/(M1*(pow(vx1, 2)+ pow(vy1, 2)+ pow(vz1, 2)))
    − 1.0*Gamma*pow(pow(vx1, 2) + pow(vy1, 2) + pow(vz1, 2),
    (1.0L/2.0L)*Alpha − 1.0L/2.0L)/M1;
    K[10]=−2.0*Gamma*vx1*vy1*(((1.0L/2.0L)*Alpha − 1.0L/2.0L)
    *pow(pow(vx1, 2)+ pow(vy1, 2) + pow(vz1, 2), (1.0L/2.0L)*Alpha
    − 1.0L/2.0L)/(M1*(pow(vx1, 2)+ pow(vy1, 2)+ pow(vz1, 2)))
    − 1.0*Gamma*vx1*vy1*(((1.0L/2.0L)*Alpha − 1.0L/2.0L)
    *pow(pow(vx1, 2)+ pow(vy1, 2) + pow(vz1, 2), (1.0L/2.0L)*Alpha
    − 1.0L/2.0L)/(M1*(pow(vx1, 2) + pow(vy1, 2) + pow(vz1, 2))));
    /* K[12−16] are similar to K[9−11] so
    are excluded here for brevity but are generated */
    K[17]=1.0*(-2*Gamma*pow(vz1, 2)*((1.0L/2.0L)*Alpha −1.0L/2.0L)
    *pow(pow(vx1, 2)+ pow(vy1, 2) + pow(vz1, 2), (1.0L/2.0L)*Alpha
    − 1.0L/2.0L)/(pow(vx1, 2)+ pow(vy1, 2)+ pow(vz1, 2))
    − Gamma*pow(pow(vx1, 2) + pow(vy1, 2) + pow(vz1, 2),
    (1.0L/2.0L)*Alpha − 1.0L/2.0L)/M1;
    return 0;
}
```
not used and may be removed in future versions. The DataAlloctor Object has two methods AddThings and Allocate. AddThings as the name suggests adds Thing Objects to be allocated into the data arrays. A Thing Object is added by DataAllocObj.AddThings(ThingObj,Number) where ThingObj is a Thing Object as created in section 2.3.1.1 and Number is the number of these that will be in the simulation. For example, if a ThingObject has 3 pieces of 'State’ data and 7 are added the result will be the addition of 21 doubles to the \( y \) array. Once all the Thing Objects have been added to the Allocator Object, the second method Allocate is used with the syntax \( y,p,c=\text{DataAllocObj.Allocate()} \) where \( y,p,c \) are variables that could be named anything but correspond to the \( y,p,c \) as previously defined. An example of this is shown in Algorithm 2.7. The use of Thing objects to set up data prototypes along with the syntax for allocation makes it very easy to track allocated data.

**Algorithm 2.7** Allocation of arrays by using Data objects from section 2.3.1.1. DataDescription is another file which would contain work done in section 2.3.1.1 and is imported to used in the example.

```python
import DataDescription as DD

DataAlloc=danplusplus.DataMaker.DataAllocator('DataAlloc')
DataAlloc.AddThings(DD.Particle,N_Particles)
DataAlloc.AddThings(DD.Rectangle,N_Rects)
DataAlloc.AddThings(DD.PPPairBond,N_PPPairBond)
DataAlloc.AddThings(DD.PRPairBond,N_PRPairBond)
DataAlloc.AddThings(DD.SimParams,1)

State,Parameters,Connectivity=DataAlloc.Allocate()
```

A big feature of the language is that the naming used in the creation of ThingObjects is translated into the namespace of the allocated arrays. For example, once the Particle Object setup in the example in Algorithm 2.1 is allocated, it can be dynamically interacted with by its name with syntax \( State.Particle.pos.x.Array \) which will return the portion of the \( y \) state array. In practice this returns a NumPy array and can be sliced, indexed and modified with any NumPy command [99]. So if one wanted to set the first 10 particles to have x position 1, one could use \( State.Particle.pos.x.Array[0:10]=1.0 \). Each namespace layer accessed by the “dot” syntax is capable of returning its respective array which is shaped for convenience. So, \( State.Particle.pos.Array \) would return the \( N_{\text{particle}} \times 3 \) array where each row corresponds to a Particle or \( State.Particle.Array \) would give the \( N_{\text{particle}} \times 7 \) array again with each row corresponding to a particle. The namespaces and use of the NumPy array slicing makes it very easy to set up initial conditions because all of the powerful NumPy functions like linspace or meshgrid can be used.

The syntax, with the exception of the unused and unnecessary name string, is very
clean and powerful and will likely remain unchanged in future work. It makes it very easy to setup initial conditions and perform post processing, find specific data within a discretization. The ease of use in the Python layer does not impact the performance of the C-Function because because only a single pointer to a 1-D array is passed which is indexed at compile time. All high level operations occur in Python and no complex data structures in the high performance layer are needed thus avoiding any potential overhead potentially incurred by them. Currently, the array storing $F$ does not have namespaces in the future this may be useful feature to add.

2.3.1.4 Connectivity And Meshes

The last major step is to setup the connectivity structure and assign models(aka SymExpressions) to it. The information in this syntax will be used to write the assembler. As an overview, the mesh objects that will be created are turned into for-loops in C and in each loop the models associated to it will be called and results added to $F$ and $\nabla F$. The connectivity objects will be used to automatically set up the indexing so the correct data is passed to each of the called model subroutines and results correctly placed in the global $F$ and $\nabla F$.

First, connectivity objects must be made. The connectivity stores which items are going to be looped over. For the moment this is limited to simple for-loops but in the future connectivity types which match more advanced iterable data structures such as bins and octrees may be included. The 3 currently implemented Connectivities are

1. AllDataTypeConnectivity(DataSegment) - Creates a loop over each piece of Data, the for-loop iterator itself is used to index Data

2. StaticConnectivity(ConnectivitySegment) - Creates a loop over each integer Connectivity Data, the for-loop iterator is used to index the Connectivity which is in turn used to index Data

3. VarLenConnectivity(ConnectivitySegment, LengthDataVariable) - Same as StaticConnectivity but LengthDataVariable sets the end condition for the loop at run time. Useful when remeshing or building Verlet lists.

The connectivity objects are then used to build interactions hierarchically using the “Mesh” object of which at the moment there is only one called NvN. A mesh object of this type is created by NvNMeshObj=NvN(ConnList,IteratorNamerObj,FullLoopFlag). The ConnList is a comma separated list in square brackets containing Connectivity Objects (e.g. [Conn1,Conn2,Conn3...,ConnN]) which matches the Python list syntax. The length of the list is the number of nested for loop which will be written. For example, if there are 2 connectivities in the list, 2 for loops will be written. The loops allow interaction between two connectivity structures. An example of this is if there were two solids with FEM connectivities, each would be stored in a StaticConnectibity Object, and the calculation of their possible collision would be done with an NvN object. The FullLoopFlag
when 'True' causes the inner loop to start at 0 and when 'False' starts with the outer-loop-index+1. The FullLoopFlag='False' is used when calculating interactions between the same connectivities and double counting needs to be avoided such as with particle-particle contact. Finally, the IteratorNamerObject is just a helper object created with IteratorNamerObj=IteratorNamer() which is responsible for making unique names for the for loop iterators. This input should be removed and the functionality included in the NvN implementation but has not be done yet. An example of connectivity setup is shown in Algorithm 2.8. In this example the AllDataTypeConnectivity connectivity is created for a set of particles and a set of rectangles. A mesh just for the particles is created used to apply forces that only rely on the state of one particle, also a particle-particle mesh and particle-rectangle mesh is created contact between them. Notice the particle-particle mesh is set 'False' to avoid double counting.

**Algorithm 2.8** Example of connectivity and meshes setup for particle code with Particle-Particle and Particle-Rectangle contact meshes.

```python
# connectivities
PConn=danplusplus.assembler.AllDataTypeConnectivity(State.Particle)
RConn=danplusplus.assembler.AllDataTypeConnectivity(Parameters.Rect)

# Make an iterator namer
IterNamer=danplusplus.assembler.IteratorNamer()

# set up meshes
PMesh=danplusplus.assembler.NvN([PConn], IterNamer, True)
PPMesh=danplusplus.assembler.NvN([PConn, PConn], IterNamer, False)
PRMesh=danplusplus.assembler.NvN([PConn, RConn], IterNamer, True)
```

The Mesh object also has one method 'AddPhysicsEquation' which is used to added a SymExpression model to be calculated inside the for-loop and added to $\mathbf{F}$ and thus also $\nabla \mathbf{F}$. Adding a model is accomplished with the syntax

```
NvNMeshObj.AddPhysicsEquation(SymExpression,
    DataSymbolMapping,
    Name,
    CodeTypes=['f_func', 'gradf', 'f_gradf'])
```

The SymExpression input is an object created as described in section 2.3.1.2. Name is a string which is what the function will be called C-Code. CodeType is a list which indicates which types of subroutines should be made 'f_func' makes just a function which returns $\mathbf{F}$, 'gradf' only $\nabla \mathbf{F}$ and f_gradf a function which returns both $\mathbf{F}$ and $\nabla \mathbf{F}$. Finally, the most complicated input is the DataSymbolMapping. This has the same syntax as a Python dictionary where the key is the SymPy symbol and the value is tuple of length
between 3 and 5. The 1st entry is a the data segment associated to the symbol which is designated using the namespaces as shown in section 2.3.1.3. The 2nd entry is a keyword setting the indexing behavior. The keyword “item” is used if the data is indexed from the information in the connectivity or “mesh” if indexed based off of the for-loop iterator. “item” is used for example to refer to particle data by the connectivity in a Bond in the example in Algorithm 2.3. However, “mesh” would be used to pick out the BondStatus Data because it is associated to the bond itself not the items that the connectivity is referencing. The 3rd entry in the tuple is an integer indicating which connectivity object in the list to reference. The 4th entry, if provided, is another tuple which sets which elements in the connectivity should be referenced. If it does not exist all elements are used. Finally, the 5th entry if provided indicates which components of the data segment in the 1st entry should be referenced to match the key if the key is a vector of symbols (1-D SymPy array). This is very little practical reason to use the 5th entry but is included for completeness. The 4th entry is useful when the indexes in a connectivity refer to different data types, common when building Verlet Lists for example between particles and FEM elements. A case when it might not be used is when the indexes in a connectivity are all the same type of data like indexes of nodes which will make up a finite element.

An example of applying a model to a mesh is shown in Algorithm 2.9. The model applied is the one created in section 2.3.1.2 Algorithm 2.4 which is imported from another file called ParticleBase and given the alias PBase. In the example, the dictionary mapping the symbols to the allocated data is shown. Included in this mapping is the association of the components of the model to data. This is what sets \( \frac{dx}{dt} \) equal to \( v_x \) and \( \frac{dv_x}{dt} \) equal to \( F_{force,x} \) in equation 2.8. When the C-code for the assembly function is made each particle will have the model applied to it. When integrated the result is the position based on velocity and the velocity based on the sum of the two force models. The generated code for the PMesh portion of the assembly is shown in Figure 2.10. The code is for 400 particles and demonstrates how the inputs are assembled to be passed into the model function PosUpdateGravDrag and the output \( F \) for the model is added to the global vector. The information in DataSymbolMapping is used to autogenerate all indexing for assembly.

### 2.3.1.5 Compile to Python Module

The last step is to compile. In this step behind the scenes, all C-Code is for the models, assembly routine, and wrappers for Python (created by scipy.weave) are written and passed through gcc to produce the shared library with \( F \) and \( \nabla F \) implementations callable in Python.

To do this, first make an Assembly Object with AssemObj=Assembly(State, Parameters, Connectivity) where State is \( y \), Parameters is \( p \) and Connectivity is \( c \) as allocated in section 2.3.1.3. The Assembly Object has three methods. The first is AddMesh with syntax AssemObj.AddMesh(MeshObj). This adds meshes as built in section 2.3.1.4. Without this addition, the mesh will not be added to the function so all loops and model calculations will be excluded during code generation. To be
Algorithm 2.9 Example of connectivity and meshes setup for particle code with Particle-Particle and Particle-Rectangle contact meshes.

```python
import ParticleBase as PBase
SymbolMapping={
    DD.g: (Parameters.SimParams.Gravity,"value",0),
    DD.Gamma: (Parameters.SimParams.Gamma,"value",0),
    DD.Alpha: (Parameters.SimParams.Alpha,"value",0),
    DD.M1: (Parameters.Particle.M,"item",0,(0,)),
    DD.v1: (State.Particle.vel,"item",0,(0,)),
}
PBase.PosUpdateAndGravDrag.F_expr [1]:
    (State.Particle.pos.y,"item",0),
PBase.PosUpdateAndGravDrag.F_expr [2]:
    (State.Particle.pos.z,"item",0),
PBase.PosUpdateAndGravDrag.F_expr [3]:
    (State.Particle.vel.x,"item",0),
PBase.PosUpdateAndGravDrag.F_expr [4]:
    (State.Particle.vel.y,"item",0),
PBase.PosUpdateAndGravDrag.F_expr [5]:
    (State.Particle.vel.z,"item",0)}

PMesh.AddPhysicsEquation (PBase.PosUpdateAndGravDrag,
    SymbolMapping,
    'PosUpdateGravDrag',
    CodeTypes=CodeTypes)
```

used in Python, the last step is to call MakePythonModule with the syntax SetupInputs,DppF,DppK=AssemObj.MakePythonModule(name, MatrixType=['Dense','COO'], CodeTypes=['f_func','gradf','f_gradf']). This function will call the other stand alone method MakeCFile which can be used to just write the C-Code. MakePythonModule also calls Weave to make Python wrappers and to be compiled into a module called nameDppAssembly where name is the name string in the input to the function. The keyword 'MatrixType' is used to set whether the gradient is return as a dense Numpy array or in Coordinate Matrix Format (index_i,index_j,value) for use with sparse matrix libraries. CodeTypes is the same as with SymExpression and again indicates which types of subroutines should be made 'f_func' makes just a function which returns \( F \), 'gradf' only \( \nabla F \) and f_gradf a function which returns both \( F \) and \( \nabla F \). In the imported Python module, the function with \( F \) is called Assembledf_func, \( \nabla F \) is Assembledgradf and both \( F \) and \( \nabla F \) returned in one function call is Assembledf_gradf. The function has three returns: SetupInputs is a dictionary of inputs given to Weave for compilation and is for reference and debugging, DppF is the array that that \( F \) will be returned to and DppK is the array that the gradient of \( F \) will be returned to when the functions are called.
Algorithm 2.10 C-code Generated to go with the setup in Figure 2.8 and 2.9.

```c
for (int a=0; a < 400; a++){
    Par[0]=SimulationParameter[1920];
    Par[1]=SimulationParameter[1921];
    Par[2]=SimulationParameter[1922];
    Par[3]=SimulationParameter[(a)*4+3];
    X[0]=Y[(a)*6+(0)*1+3];
    X[1]=Y[(a)*6+(1)*1+3];
    X[2]=Y[(a)*6+(2)*1+3];
    PosUpdateGravDrag_f_func(X, Par , F );
    DppF[((a)*6+0)*1+0]+=F[0];
    DppF[((a)*6+1)*1+0]+=F[1];
    DppF[((a)*6+2)*1+0]+=F[2];
    DppF[((a)*6+3)*1+0]+=F[3];
    DppF[((a)*6+4)*1+0]+=F[4];
    DppF[((a)*6+5)*1+0]+=F[5];
}
```

An example of this portion can be found in Algorithm 2.11. At this point the role of the DSL is done. The function and gradient can be used in conjunction with any solver or time integration method. The next chapter details of how $\mathbf{F}$ and $\nabla \mathbf{F}$ are used for a complete simulation in conjunction with Newton's Method and general Runga-Kutta integration schemes for time stepping a dynamic solution. First, in the next section a brief discussion of future considerations about numerical stability and high performance computing (HPC)/Parallel Computing will be discussed.

2.4 Future Considerations

2.4.1 Floating Point Error

When doing calculation in infinite precision or symbolically the results are exact. However, real computers do not store numbers with infinite precision but rather in floating precision. A concern when doing calculation in floating point precision is the limited precision might lead to a loss of information which results in incorrect answers [96, 35]. An example of this is when normalizing a vector whose length is very close to zero. Symbolically this is a well defined operation but the length may be so close to zero that the division by the length is effectively meaningless with the limited precision in the computer. In setting up equations for $\mathbf{F}$ symbolically, the dangers of floating point error must still be considered. While the symbolic representation is exact, after it is translated into C-code floating point numbers will be plugged into those expressions and
Algorithm 2.11 Example of the syntax for setting up an assembly and producing the Python module named DirectLaserSinterDppAssembly with only the \( F \) function made.

\[
\text{CodeTypes} = ['f_func ']
\]

#make the assembly object
Assem1=danplusplus.assembler.Assembly(State, Parameters, Connectivity)
Assem1.AddMesh(PMesh)
Assem1.AddMesh(PPMesh)
Assem1.AddMesh(PRMesh)

name='DirectLaserSinter' SetupInputs, DppF, DppK =
Assem1.MakePythonModule(name, MatrixType='COO',
CodeTypes=CodeTypes)

unacceptable error in the answer could result. For the unit vector, this may be avoided by always normalizing a vector guaranteed to be of sufficiently non-zero length. For example, relative velocity could be close to zero but if using discrete elements that are designed for minimal overlap the center-center vector is guaranteed to have length significantly greater than zero.

The problem may be less obvious and harder to avoid in other situations. For example, what if \( F \) requires a matrix inversion to occur. This is commonly seen in finite elements when mapping between the isoparametric coordinate system where integration is typically performed. The inversion could have a hidden operation that results floating precision error. When performing such operations numerically, pivoting can be used to reduce the error by arranging the order of operations to improve floating point accuracy. With the Symbolic representations however no knowledge of the relative sizes of numbers is known so the information cannot be accounted for via pivots. The ramifications of this issue are yet unknown and will require further study by the author to develop automatic mitigation techniques.

2.4.2 High Performance Computing (HPC) Considerations

Another major area of interest in computational engineering that must be addressed in parallel computing or sometimes called higher performance computing (HPC). The current implementation of the DSL compiler is not capable of generating parallel implementations of \( F \) and \( \nabla F \). While the author has not had time to include parallel implementations, it is of paramount importance to the field. One of the features of generating C-code as an intermediate step is that it allows for the relatively easy inclusion of parallelization.

Compilation is can be considered to be done in three steps: parsing, optimizing and lower level generation. Parsing is the step when the syntax is read and put into a format
that is easy to pass into an optimizer. The optimizer then attempts to make the most efficient low level instructions possible. Finally, a code generation backend makes usable instruction of the system. The separation of the two steps allows for multiple syntax to access a single optimizer. The current implementation is a little different in that the parser does not directly target an optimizer. First, it reads the syntax and stores them in Python objects. These objects then contain methods that tell it how to write its section of the C-code. The C-code then can be put into any ANSI compliant C compiler (gcc, clang, intel). The intermediate step to C has the advantage that the DSL syntax does not need to be encumbered by the full generality of computing and the author could easily advantage of the powerful optimizers already available for C code. (Technically the author could have targeted the optimizers used with C compilers such as LLVM [61] directly but this is much more difficult. It is is easier pretend the C-Compiler as a whole is the optimizer and the DSL code is parsed into C for this “optimizer”)

More importantly, implementation details can be hidden and optimized in the C-layer. Particularly with the use of high performance parallel libraries. Libraries such as PETSc [12, 11, 10] and Trilinos [46, 47, 48, 49] are large and well thought out libraries for scientific computing and already available to extend C to make parallel distributed operations, particularly for matrices, easier. The goals of the domain specific language is exactly to hide this level of operation from day-to-day considerations. However, by still going through the C layer, existing libraries can be used in the creating the best implementation of the DSL syntax. The current DSL compiler by the author does not compile into formats other than C but it should only require the addition to the parser to generate the appropriate. The parsing of the syntax will be done with the Python interpreter as before but code generation will change. For code generation, the compiler iterates through the Python data structures and asked each object to provide its respective implementation to be inserted into the larger program. For the moment, code generation returns a C implementation but by adding the ability in each block to return a PETSc implementation the overall code would be valid PETSc. The result is that the DSL code is left unchanged but the function implementation would now be in parallel, ideally without any additional information from the user. Likewise if portions were better addressed with Fortran, an implementation could added.

It would also be possible to add an entirely native implementation that bypasses the intermediate language C/PETSc/Fortran layer which directly targets an optimizer like LLVM. There is no plans to do this though because the approach of passing through another language to take advantage of the multitude of powerful libraries is much more useful. The author sees no situation in which writing the optimizer itself would be advantageous for a person who is interested in computational engineering. Vast work by the computer science community on optimizers would be nearly impossible surpass without dedicated research on the topic. To further improve performance, it would also be interesting to make use of autotuners for the implementation in the C-layer itself. Instead of manually coding the generators, autotuners might be used to automatically produce the best C-implementation without human intervention. Ultimately, the goal is to push
the field to automate steps of simulation which are not specifically human. By this, it is meant that the computer can not decide what problem the human user is interested in solving(...)at least not yet!) which is what the user should be responsible for expressing. The task of how to solve the problem should be done as automatically as possible taking advantage of all the cumulative knowledge of the field and the power of the computer to search for optimal approaches.

The DSL syntax and compiler implementation are all designed to access the best features at the different layers of already existing software. Python is particularly good at user friendly operations and easy data management. C is an established language for efficient coding with access to a plethora of powerful libraries. The syntax based around discretized forms allows for a uniform syntax that allows for all simulations to exist in a single framework. Finally, using syntax that is valid Python makes implementing the parser very easy and the compilation back to the Python callable functions means that the user can work exclusively in the Python environment, allowing the user to take full advantage of all the modern feature that are provided already in the Python environment. The incredibly large user base of Python also means that the many libraries for editing data and writing files and many more can also be utilized by the user for top level operations such as setting up the simulation, flow control, file IO and potentially solvers.
Chapter 3

Simulations with the DSL

3.1 Simulation Setup Details

The $F$ function contains all the information about the dynamics of as systems but in order to actually get to the solution requires more steps. Integration of the ODE to advance a time step and/or solving linear and non-linear systems must be performed. A general simulation flow should look like the image on the left in Figure 3.1. After the DSL compiler is run the data arrays are passed to the Initial Conditions Routine which sets up the actual values of the data at the start. Next the Update State module, using the $F$ and $\nabla F$ functions produced by the DSL, is called repeatedly to update the time. After each time step, ideally, simulation flow and remeshing would take place and output written if desired. Since many, many time steps are taken it is impractical to do this every time step. As a result, the flow chart on the right in Figure 3.1 is the process actually used henceforth. Here the initial conditions are set as before but remeshing and flow control routine are only executed periodically. In this way, many time steps are taken before the expensive remeshing and output occur. The upside is a much faster run time, the downside is that simulation can only be changed periodically so the mesh must remain robust enough or contain enough potential contacts so none are missed for the longer periods of time. This is the current setup and has been productive but other Sim Control modules that are more appropriate for a given problem could be used also.

The Sim Control routine is a general Python routine and each rounded rectangle represents another Python routine that is designed for a specific problem. This is easiest to see with a Write Output Block. This is a function which is written by the user that take the data in State,Parameter and Connectivity and outputs files for visualization and data analysis. For visualization, since the data between different simulations can be different, it is important to produce a specific routine that visualizes the information correctly. For visualizations, matplotlib [52] can be used to make plots at run time natively in Python but is quite slow. Generally, I more commonly used the XML VTK file format [81] and visualize in Paraview[5, 9], an easy to use open source visualization tool.

The Time Step, Flow Control and Remesh block is first responsible for changing the
time step if necessary. Second, Flow Control means the modification of any parameter data which dictates the processes behavior such as starting a laser moving across the domain or a wiperblade. Finally, remeshing is anything that effects the Connectivity data. In particle simulations, this could be updating the contact Verlet lists via Binning Algorithm as is done in the Laser Metal Deposition Simulation in section 7.3 or might be a call to Gmsh[41] or other meshing software and inserted into the Connectivity array.

Finally, the Update State is the heart of the simulation. This is where the degrees of freedom, \( y \), are updated via time integration to get the state at a time \( h \) in the future or solve a system of equations if solving for a steady state or quasi-static field. The update when integrating in time can be done with a Runga-Kutta scheme as described in the next section.
3.2 General Runge Kutta Methods

In a general Runge Kutta (RK) scheme for integrating an ODE, there is a matrix \( a \) which is \( s \times s \) where \( s \) is the number of stages. A vector \( c \) length \( s \) which stores the time of each stage and \( b \) is a vector of length \( s \) which stores the weights of the recombination of the stages. Various Butcher tableaus[18] providing \( a, b, c \) for different schemes can be found in Appendix B. The goal of time integration is to approximate the state at the next time step \( y^{n+1} \) time \( h \) in the future from current state \( y^n \). For a general RK scheme the update[62] is given by

\[
\begin{align*}
Y_1 & = y^n + h \sum_{j=1}^{s} a_{1j} F(Y_j, t_n + c_j h) \\
Y_2 & = y^n + h \sum_{j=1}^{s} a_{2j} F(Y_j, t_n + c_j h) \\
& \quad \vdots \\
Y_s & = y^n + h \sum_{j=1}^{s} a_{sj} F(Y_j, t_n + c_j h) \\
y^{n+1} & = y^n + h \sum_{j=1}^{s} b_j F(Y_j, t_n + c_j h)
\end{align*}
\]

or in short form

\[
\begin{align*}
Y_i & = y^n + h \sum_{j=1}^{s} a_{ij} F(Y_j, t_n + c_j h) \\
y^{n+1} & = y^n + h \sum_{j=1}^{s} b_j F(Y_j, t_n + c_j h)
\end{align*}
\]

3.2.1 Fully Implicit

A general \( a \) matrix could have non-zero entries anywhere, if there are entries above the diagonal then this is called a fully implicit scheme and all stages are possibly coupled to another and thus the \( Y_i \) must be solved for simultaneously in one large system of equations. The solving of \( Y_i \) can be rephrased as a zero finding problem of the form

\[
F_i'(Y) = Y_i - y^n - h \sum_{j=1}^{s} a_{ij} F(Y_j, t_n + c_j h) = 0
\]

where

\[
Y = \begin{pmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_s \end{pmatrix}, F'(Y) = \begin{pmatrix} F_1(Y) \\ F_2(Y) \\ \vdots \\ F_s(Y) \end{pmatrix}
\]
One way to solve this equation is with Newton’s method which is discussed in section 3.3. Newton’s Method requires an initial guess which can be anything, but one possibility here is

$$Y^1 = \begin{pmatrix} y^n \\ y^n \\ \vdots \\ y^n \end{pmatrix}$$

which is an intuitive choice because $y^n$ is the last know closest solution. To get the final solution, iterations are performed according to

$$Y^{k+1} = Y^k - (\nabla_y F'(Y)|_{Y^k})^{-1} F'(Y)|_{Y^k}$$

Writing this out for each stage

$$Y^{k+1}_i = Y^k_i - \sum_j \left( \nabla_y F(Y^k)|_{Y^k}_{ij} \right)^{-1} F'(Y)|_{Y^k_i}$$

The gradient of $\nabla_y F'$ is easy to evaluate because it is just a linear combination of $Y_i$s and evaluation of $F(Y)$. The derivative of such a sum is then the sum of the derivatives giving

$$(\nabla_y F'(Y))_{ij} = I\delta_{ij} - h \sum_{j=1}^s a_{ij} \nabla_y F(Y_j, t_n + c_j h)$$

which expressed as a matrix looks like

$$\nabla_y F'(Y) = \begin{bmatrix} I - ha_{11} \nabla_y F(Y_1, t_n + c_1 h) & -ha_{12} \nabla_y F(Y_2, t_n + c_2 h) & \cdots & -ha_{1s} \nabla_y F(Y_s, t_n + c_s h) \\
-ha_{21} \nabla_y F(Y_1, t_n + c_1 h) & I - ha_{22} \nabla_y F(Y_2, t_n + c_2 h) & \cdots & -ha_{2s} \nabla_y F(Y_s, t_n + c_s h) \\
\vdots & \vdots & \ddots & \vdots \\
-ha_{s1} \nabla_y F(Y_1, t_n + c_1 h) & -ha_{s2} \nabla_y F(Y_2, t_n + c_2 h) & \cdots & I - ha_{ss} \nabla_y F(Y_s, t_n + c_s h) \end{bmatrix}$$

To be explicit, the notation

$$\nabla_y F\left(Y_i^k\right)$$

is the gradient of $F$ with respect to $y$ evaluated at $Y_i^k$. After the $Y_i$ are solved for $y^{n+1}$ still needs to be found by

$$y^{n+1} = y^n + h \sum_{j=1}^s b_j F(Y_j, t_n + c_j h)$$

which requires another round of $F$ evaluations. The extra $F$ may be avoid if one decides that the last $F$ evaluation in the newton iteration is sufficient. This is not an unreasonable thing to do because often a stopping criteria for newton iterations involves checking that the update to $Y_i$ is near floating point precision as discussed in section 3.3. This would mean that

$$Y_i^{k+1} \approx Y_i^k$$

to near floating point precision so $F(Y_i^k, t_n + c_j h)$ should also be nearly acceptable if $F$ is well conditioned.
3.2.2 DIRK and Explicit Methods

If $a_{ij} = 0$ for $j > i$ this is a Diagonally Implicit Runga-Kutta (DIRK) method and the stage is uncoupled from later stages but the function evaluation still depends on itself for its evaluation and thus a solve step must be performed. For each stage,

$$F'(Y_i) = Y_i - y^n - h \sum_{j=1}^{i} a_{ij} F(Y_j, t_n + c_j h) = 0$$

for $Y_i$ is solved. The gradient to use with Newton's method for each stage would then be

$$\nabla Y_i F'(Y_i) = I - h a_{ii} \nabla y F(Y_j, t_n + c_j h)$$

Again, after the $Y_i$ are solved for $y^{n+1}$ also needs to be found as above. The fully implicit formulation collapses to this for DIRKs if each preceding stage is solved by itself then the $F'(Y_{j<i}) = 0$ which multiplies the $-h a_{i1} \nabla y F(Y_{j<i}, t_n + c_j h)$ terms in the full gradient matrix (eq. 3.2 or 3.3) and thus does not contribute. Also, $a_{j>i} = 0$ by definition and thus equation 3.4 is recovered.

If $a_{ij} = 0$ for $j \geq i$ then the integration is said to be an explicit method and means that to evaluate a stage, current and later stages are not needed so the function can be evaluated with the correct $Y_i$s at each stage. For this it is sufficient to only have an implementation of the function $F$ as no solving is required. For the first stage $a_{1j} = 0$ so $Y_1 = y^n$ and $y^n$ is assumed to be know as it is the state of the current time. Now $Y_2 = y^n + a_{21} F(Y_1, t_n + c_j h)$ because all other $a_{2j} = 0, j \neq 1$. For all subsequent stages it is the same that function evaluations only occur for know $Y_i$. For efficiency, after an $F$ is evaluated for a $Y_j$, its is added to each of the $Y_i$ it contributes to scaled by $a_{ij}$ and to the $y^{n+1}$ update scaled by $b_j$. This way $F$ is not evaluated redundantly.

Both of these methods are more memory efficient because full vector of length $sN$ and matrices($sN \times sN$) do not ever have to be formed only the vectors for $N$ and matrix of $N \times N$ (or DIRK) are needed. Also it should be noted that for the Explicit case the Fully Implicit formulation should not be used. While the correct answer will still be found it will be very inefficient. First for the Explicit case no solves are needed which is bad but worse it takes $s-1$ solves to get the solution. This is because the correct $Y_i$ needs to be solved for and then propagated to later stages before the true answer for each stage is found.

3.2.3 Steady State Solutions

For steady states solution for problems of the form $\frac{dy}{dt} = F(y)$, set $\frac{dx}{dt} = 0$ yielding $F(x) = 0$. Again, the zeros of the function are the solution. Newton’s method is again a possible option for solving such as system. An initial guess, $y^1$, close to the expected final answer should be used as the start for the iterations. The iterations here require
not large matrix assembly like in the fully implicit methods, simply the gradient of $F$ is sufficient. The iterations in this case are

$$y^{i+1} = y^i - \left( \nabla_y F \big|_{y^i} \right)^{-1} F \big|_{y^i}$$

Once tolerance is achieved, Newton’s method is stopped and an answer is found.

### 3.3 Newton’s method

Newton’s method is one method of solving for the zeros of a non-linear function. Given function $F(Y)$ the solutions of $F(Y) = 0$ can be found iteratively via the update formula

$$Y^{k+1} = Y^k - \left( \nabla_Y F \big|_{Y^k} \right)^{-1} F \big|_{Y^k}$$

The iterations are stopped when a tolerance is achieved. Common metrics used to decide when the tolerance are met are

- Residual $\| F(Y) \|_{Y^k} < tol$
- Displacement $\| \left( \nabla_Y F \right) F(Y) \|_{Y^k} < tol$
- Energy $F(Y)^T \left( \nabla_Y F \big|_{Y^k} \right)^{-1} F \big|_{Y^k} < tol$

Where $\| \|$ is a placeholder for a norm of choice such as the max norm or the $l_2$ norm. In finite dimensions, all norms around bounded by each other so convergence in one is guaranteed in another. In practice, different tolerances will have to be selected to achieve the same point of convergence. For a double precision, $F$ and $Y$, $tol = 1 \times 10^{-12}$ is usually acceptable for the Residual and Error measures. The Energy norm is effectively the multiplication of the other two errors so its tolerance needs to be squared where $tol = 1 \times 10^{-24}$ may be used. Actual tolerances have to be selected by the amount of floating point error caused by the function evaluation and gradient evaluation. The Energy norm is a great option because it measures both the Residual and the Displacement, meaning it measures both how close the residual $F(Y^k)$, is to zero but also how much the solution has been updated, $\left( \nabla_Y F \big|_{Y^k} \right)^{-1} F \big|_{Y^k}$, in the last step. It is possible to have a very small residual and still be very far from the solution if the derivative is flat. If it is, the updates will still be large as the iterations updates follow the function yet closer to the state that is the zero.

It should be noted that Newton’s method is not guaranteed to converge. Also for non-linear problems in general there is no guarantee that there is one solution. What is good about Newtons method though if the gradient is correct, it will converge quadratically (if the initial guess is sufficiently close to the solution). Quadratic convergence is observed when the error metric of choice drops by approximately square, e.g. iteration 1, error=$1 \times 10^{-3}$, iteration 2, error=$1 \times 10^{-6}$, iteration 3, error=$1 \times 10^{-12}$. This leads to extremely
quick convergence in terms of number of iterations. However, the computation time might be very long if the calculation for $\mathbf{F}$, $\nabla \mathbf{F}$ or the solve step is very costly in terms of computation time. In general, the calculation of the gradient and the solve are the most costly part.

For this reason, often explicit integration schemes are used to avoid the need to assemble a gradient and perform a solve. Explicit methods are used to solve dynamics problems and to solve static problems via dynamic relaxation, where dynamic system is allowed to evolve until it reaches a spot where $\frac{\partial y}{\partial t} = 0$ within tolerance. Another approach to reduce computation time are quasi-Newton methods. This is an approach where the exact gradient $\nabla_y \mathbf{F}(\mathbf{y})$ is not used instead an approximation that is computationally less intensive to evaluate and solve is used. In this case, quadratic converge might be lost and a smaller convergence region (translating into smaller time steps) may result. Even with the “wrong” gradient, if convergence occurs then the zero is still found just the same. By choosing a good pseudo-gradient and time step an optimal approach from a computation time stand point may be found, beating both Explicit stepping methods and methods depending on exact Newton’s method. The quasi-newton method can be substituted for newtons method in the zero finding step of the implicit schemes.

A wide variety of pseudo-gradients have been developed over time for solving non-linear systems efficiently. At the moment the DSL has not provisions for returning pseudo-gradients of any variety. Another approach is only evaluating the gradient once and using it for all iterations. In this way it is assumed the gradient will not change too much and computation time is saved by only requiring one evaluation and the factorization used for the solve can be reused multiple times.

Finally, non-gradient methods as used by Zohdi [111, 114, 112, 113] for solving for the zeros can be used. Here, iterative techniques that do rely on the gradient but only evaluations of $\mathbf{F}$ are used to find the zeros of an equation which give the solution to the implicit equation 3.1 or steady state problems. The most important take aways from this is that implicit Runge-Kutta schemes and static solves can be framed as zero finding problems. There are a myriad of advanced solving techniques that exist for robust zero-finding. Implementations of these methods are provided in many libraries such as Scipy in Python and PETSc along with its Python binding PETSc4py. For the zero-finding routines in the libraries all that must be provided is the $\mathbf{F}$ and/or $\nabla \mathbf{F}$ again demonstrating why so much time has been dedicated to these two functions.

### 3.4 Laplace Equation Examples

The PDE called the Laplace equation can be written as

$$\nabla \cdot (\mathbf{K}(u) \nabla u(x, y)) = 0$$

In these examples, the system is solved on a domain, $\Omega$, which is a square of dimensions $X[0,1]Y[0,1]$. The edges have a Dirichlet Boundary condition with a value of $u = 0$ and
the mid-point \( u(x = .5, y = .5) = 1 \). Dirichlet domain is referred to as \( \Omega_D \). The system can be written as, \( F(u) = 0 \) where

\[
F(u) = \begin{cases} 
\nabla \cdot (K(u) \nabla u(x, y)) & \Omega \setminus \Omega_D \\
u - u_b & \Omega_D
\end{cases} \tag{3.5}
\]

### 3.4.1 Finite Differences

Using the strong form above, the linear isotropic PDE is solved with

\[
K(u) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.
\]

Simplifying the strong form by substituting the above expression for \( K \) gives

\[
F(u) = \begin{cases} 
\nabla^2 u(x, y) & \Omega \setminus \Omega_D \\
u - u_b & \Omega_D
\end{cases}
\]

which the \( \nabla^2 \) is discretized on a uniform grid using the second order centered stencil. Assuming a unit cell for the finite difference with local indexes as shown in Figure 3.2, for nodes not on the boundary condition \( F \) at the location is given by

\[
F_{elem} = \frac{-4u_1 + u_2 + u_3 + u_4 + u_5}{h^2} \tag{3.6}
\]

\[
\nabla F_{elem} = \frac{1}{h^2} \begin{pmatrix} -4 & 1 & 1 & 1 \end{pmatrix} \tag{3.7}
\]

The parts pertaining to this \( F_{elem} \) are coded in the DSL is shown in Algorithm 3.1 with the resulting generated C-code in Algorithm 3.2 which can be seen to match equations 3.6 and 3.7.
Figure 3.3: Solution of the Poisson equation using finite differences and visualized with matplotlib. The plot is colored by $u$ and height is mapped to $u$.

On $\Omega_D$ the requirement that $u - u_b$ is enforced exactly by setting the initial guess of $u = u_b$ on $\Omega_D$ then choosing the update formula to be

$$F_{\text{elem}} = u_1$$

$$\nabla F_{\text{elem}} = 1$$

This is a tricky way of getting the stiffness matrix to have a one in it so that preserves the $u_b$ put in the initial guess during the solve. Since the problem is linear, the solution is found with a single solve via $u = (\nabla F)^{-1} u_0$ where

$$u_0 = \begin{cases} u_b & \Omega_D \\ 0 & \text{otherwise} \end{cases}$$

and $u$ are the nodal values of the solution. The results are shown plotted using matplotlib [52] in Figure 3.3.

### 3.4.2 Finite Elements

In the finite element solution, the requirement that $\nabla \cdot (K(u)\nabla u(x, y)) = 0|\Omega \setminus \Omega_D$ is applied weakly using Galerkin finite elements which are derived in equation 2.5 are
reproduced below

\[ F_{weak}(u) = \int_{\Omega} \nabla_x \phi [K(u)] \nabla_x u(u)] \, dx - \int_{\partial \Omega} q \cdot n \phi \, dS \]

since all boundaries are Dirichlet

\[ F_{weak}(u) = \int_{\Omega} \nabla_x \phi [K(u)] \nabla_x u(u)] \, dx. \]

Where \( \phi_i \) in \( \phi \) are shape functions associated to node \( i \) for linear quadrilateral elements. The Dirichlet boundary conditions, the constraint \( u - u_b = 0|\Omega_D \), is applied strongly point-wise with linear penalty method

\[ F_{pen}(u) = K_s(u - u_b)|_{\Omega_D} \]

giving

\[
F(u) = F_{weak}(u) + F_{pen}(u) \\
= \int_{\Omega} \nabla_x \phi [K(u)] \nabla_x u(u)] \, dx + K_s(u - u_b)|_{\Omega_D} = 0 \quad (3.8)
\]

where \( u_b \) are the boundary values described above and \( \Omega_D \) are the nodes on the edges and nodes at \( x = .5, y = .5 \). Nodally, the penalty is given by

\[ F_{j,pen}(u) = K_s(u_j - u_{b,j})|_{\Omega_D} \]

for \( j \) where the nodes are on the Dirichlet boundary.

The weak form part of the problem can be set up over an element as

\[ F_{i,elem}(\tilde{u}) = \int_{\Omega_i} \nabla_x \tilde{\phi} [K(\tilde{u})] \nabla_x \tilde{u}(\tilde{u})] \, dx \]

where

\[
\tilde{\phi} = \begin{pmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \\ \phi_4 \end{pmatrix}, \quad \tilde{u} = \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{pmatrix}, \quad \tilde{u}(\tilde{u}) = \tilde{u} \cdot \tilde{\phi}
\]

are the \( \phi_i \) are the four shape functions and \( u_i \) are nodal values associated with quadrilateral element \( i \), which covers \( \Omega_i \) of the original domain \( \Omega \). This is usually integrated over in an isoparametric domain defined by \( x(z) = \hat{x} \cdot \hat{\phi} \) where \( \hat{x} \) are the nodal positions of the element and \( \hat{\phi} = \phi(z) \) is a vector of shape functions in the isoparametric domain. Substituting results in

\[
F_{i,elem}(u) = \int_{\Omega} \nabla_z \tilde{\phi} \nabla_x z [K(\tilde{u})] \nabla_x z] \, dz \cdot (\nabla_x z) \, dz \\
= \int_{\Omega} \nabla_z \tilde{\phi} [\nabla_x z]^{-1} [K(\tilde{u})] \nabla_x z] \, dz \cdot (\nabla_x z) \, dz
\]
where \( \hat{u} = u(z) \) over \( \Omega_i \) and \( \square \) represents the isoparametric domain to which the element maps. The elemental and nodal contributions are then assembled into the global \( \mathbf{F} \) as

\[
\mathbf{F} = \sum_{i=\text{elements}} \mathbf{F}_{i,\text{elem}}(u) + \sum_{j=\text{BoundNodes}} \mathbf{F}_{j,\text{pen}}(u).
\]

This will not converge to the solution of equation 3.5 exactly unless \( K_s = \infty \); here \( K_s = 1 \times 10^5 \).

The discretized \( \mathbf{F} \) function in equation 3.8 is now a zero finding problem for which Newton’s method is used as in section 3.3 to find a solution. Below a linear and non-linear problem are solved. The mesh is produced using Gmsh [41] and visualizations with Paraview [5, 9] using VTK XML file format [81].

### 3.4.2.1 Linear Constitutive Model

First with the same \( \mathbf{K} \) as with the finite difference problem

\[
\mathbf{K}(u) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}
\]

the problem is again linear. By solving equation 3.8 the solution is found and plotted in Figure 3.4. Newton’s method is started with \( \mathbf{u} = \mathbf{0} \) and the solution is found exactly in one iteration (one matrix inversion). A second iteration must be done when using Newton’s method for the error metric to confirm the system has been solved which is demonstrated in Table 3.1. This shows convergence metrics for the Newton iteration and that the error is at machine precision after the second iteration proving that the first iteration solved the problem exactly. This demonstrates that the problem is in fact linear and the linearization is correctly generated by the DSL.
<table>
<thead>
<tr>
<th>Iteration</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F(u)$</td>
<td>100000.0</td>
<td>8.6293194812e-12</td>
</tr>
<tr>
<td>$(\nabla F)^{-1} F$</td>
<td>0.999988666483</td>
<td>3.89186183145e-16</td>
</tr>
<tr>
<td>$F (\nabla F)^{-1} F$</td>
<td>99998.8666483</td>
<td>3.35841191203e-27</td>
</tr>
</tbody>
</table>

Table 3.1: Error by newton Iteration for Linear Laplace Equations showing that solution is found to machine precision after the first iteration.

Figure 3.5: Solution of the Laplace equation with $K(u) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ in solid gray. Solution to $K(u) = \begin{bmatrix} u + 1 & 0 \\ 0 & u + 1 \end{bmatrix}$ in color map corresponding to the the height, $u$. Notice the gap between the two solutions.

3.4.2.2 Non-Linear Constitutive Law

The process is repeated for the constitutive model

$$K(u) = \begin{bmatrix} u + 1 & 0 \\ 0 & u + 1 \end{bmatrix}$$

which leads to a non-linear system. The solution is found with Newton’s method and shown Figure 3.5 plotted with the linear solution for comparison where the vertical height is the temperature for illustrative purposes. Table 3.2 demonstrates the quadratic convergence rate of Newton Iterations to again show linearization has been performed correctly and function generated correctly by the DSL. With Newton iterations, it is possible for convergence to still occur when the linearization of the system of equations is incorrect but usually at a linear rate.
Table 3.2: Measures of convergence after each newton iteration. It is easy to see the quadratic convergence in the last 3 iterations as the exponent doubles in each iteration. After iteration 6, tolerance \( \mathbf{F}(\nabla \mathbf{F})^{-1} \mathbf{F} < 1 \times 10^{-25} \) is achieved.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathbf{F}(u) )</td>
<td>100000.0</td>
<td>0.861</td>
<td>0.00872</td>
<td>5.18e-06</td>
<td>2.03e-12</td>
<td>3.68e-12</td>
</tr>
<tr>
<td>( (\nabla \mathbf{F})^{-1} ) ( \mathbf{F} )</td>
<td>0.99</td>
<td>0.0826</td>
<td>0.00250</td>
<td>2.28e-06</td>
<td>1.87e-12</td>
<td>1.36e-16</td>
</tr>
<tr>
<td>( \mathbf{F}(\nabla \mathbf{F})^{-1} \mathbf{F} )</td>
<td>99998.86</td>
<td>0.0712</td>
<td>2.18e-05</td>
<td>1.18e-11</td>
<td>3.80e-24</td>
<td>5.01e-28</td>
</tr>
</tbody>
</table>
Algorithm 3.1 The DSL code to setup the stencil for the Laplacian. “Connectivity.LaplacianStencil” is allocated so a row has 5 pieces of data.

```python
#symbols
u1, u2, u3, u4, u5 = sympy.symbols('u1 u2 u3 u4 u5', real=True)
h = sympy.symbols('h', real=True)
StencilVec = sympy.Matrix([u1, u2, u3, u4, u5])

#model
f = sympy.Matrix([[(-4*u1+u2+u3+u4+uyp1)/h**2]])
variables = sympy.Matrix([[StencilVec]])
parameters = sympy.Matrix([[h]])
LaplacianStencil = danplusplus.SymDiff.
    SymExpression(f, variables, parameters)

#set up meshes
LConn = danplusplus.assembler.
    StaticConnectivity(Connectivity.LaplacianStencil)
IterNamer = danplusplus.assembler.IteratorNamer()
LMesh = danplusplus.assembler.NvN([LConn], IterNamer, True)

#assign model
Mapping = {h: (Parameters.h, 'const', 0),
            StencilVec: (State.u, 'item', 0),
            f: (State.u, 'item*', 0, (1,))}
LMesh.AddPhysicsEquation(LaplacianStencil, Mapping,
                        'LaplacianSten',
                        CodeTypes=['f_gradf'])

#mapping add to mesh
Assem1 = danplusplus.assembler.Assembly(State, Parameters, Connectivity)
Assem1.AddMesh(LMesh)
```
Algorithm 3.2 f_gradf function C function for the Laplacian stencil generated for the model in Figure 3.1.

```c
int LaplacianSten_f_gradf (double * X, double * Par, double * F, double * K) {
    double u1=X[0];
    double u2=X[1];
    double u3=X[2];
    double u4=X[3];
    double u5=X[4];
    double h=Par[0];

    F[0]=1.0*(-4*u1 + u2 + u3 + u4 + u5)/pow(h, 2);
    K[0]=1.0/pow(h, 2);
    K[1]=1.0/pow(h, 2);
    K[2]=1.0/pow(h, 2);
    K[3]=1.0/pow(h, 2);
    K[4]=-4.0/pow(h, 2);
    return 0;
}
```
Chapter 4

Multiphysics Discrete Elements

The discrete element method was popularized by Cundall and Strack[33] as means to simulate the dynamics of granular systems. In this historic formulation, the discrete element has a one to one correspondence with a grain within the system. Each discrete element is kinematically a rigid sphere with homogeneous properties such as density, elastic modulus, Poisson ratio etc. The elements are often referred to as soft though because the constitutive models allow for overlap between elements in contact. Presented here will be a description of the discrete element with its underlying assumptions and differential equations which govern the evolution of its properties in time. The element will keep the assumptions is a kinematically rigid sphere with homogeneous properties but extend the model to account for heat transfer between elements and thermal expansion. Additional multiphysics behavior will also be incorporated through new constitutive models for bonding presented in following chapters.

4.1 The Discrete Element as an Element

The macroscale objects that are of interest to the field of engineering are made up of very large numbers of atoms. In order to simplify the physical description of macroscale objects the assumption is often made that they are actually continuous in nature, made up of an infinite number of material points. This is the basis of the field of continuum mechanics. On this continuum, physical laws and material models are imposed which allow engineers to model the behavior of the macroscale object. However, because computers are finite, all the infinite points of the continuum in the model cannot be solved for. As a solution, the infinite set of points in the continuum are grouped together to form a finite set of elements. These elements represent an assumption that all the points contained in it will linked together according to a functional form described by the state of only a few points. Breaking the continuum down to be represented by elements is know as discretization.

While the language of continuum mechanics is generally used to describe bodies which
do not have voids or cracks, the idea of an infinite number of points existing and grouped together in elements can still be used to understand the system of discrete elements which are inherently discontinuous objects. For this, I propose the definition that a discrete system be any whose set of items are countable in that they can be mapped to the integers. In contrast, let an non-discrete system be one that the fundamental units are uncountable specifically in the sense that they can be mapped to the real numbers.

Starting with a non-discrete system with an uncountable number of points let each point be assigned a unique identifier which is denoted $Z$ with $n_{dim} \in \mathbb{Z}$ real numbers where $n_{dim}$ corresponds to the spatial dimension of the problem. The system and its unique identifiers $Z$ can be mapped to $\mathbb{R}^{n_{dim}}$. Note: $Z$ does not have any interpretation as a position or any other physical value it is just an identifier, also no assumptions are made about the continuity of $Z$ or its derivatives. The points exist without any reference to physical systems or quantities.

Next, a mapping is introduced $D(Z, t)|\mathbb{R}^{n_{dim}} \times \mathbb{R} \to \mathbb{R}^{n_{data}}$ where $D$ is data, or state space, that is associated to the point with the unique identifier $Z$ at a given time $t$. $n_{data} \in \mathbb{Z}$ is the number of unique real number data associated to the point. No assumptions about the continuity or the invertibility of the mapping are made. A discrete object is created by designating a subset from the uncountable set of points (subsets can contain countable or uncountable number of points) which will be denoted $Z_i$ where $i \in \mathbb{Z}$ is a unique integer identifier as per the definition of a discrete set. All that is required of these sets are that the $Z_i$ are exclusive in that their union is the empty set and only a discrete number of them exist. They do not need to include all of $Z$.

As of now the components of $D$ has no meaning and no way of finding it is just an abstract concept connecting points to data. Physical and engineering meaning comes from naming components of $D$ and applying constraints. It should be noted that, naming an element of data $D$, does not effect the mathematical definition just its interpretation. With dynamics, for each point there are $2n_{dim}$ components per point, a position and velocity for each spatial dimension. $n_{dim}$ of $D$ will be assigned to a spatial position, $x \in \mathbb{R}^{n_{dim}}$ and another $n_{dim}$ values are assigned to the spatial velocity, $v \in \mathbb{R}^{n_{dim}}$. Again, this naming matches physical concepts but has no actually bearing on the mathematical character of the problem. Note:In continuum mechanics the portion of the mapping $D$ that handle position for a body is often called the configuration mapping and denoted as $\chi$, where $x = \chi(X, t)$ where $X$ is some position at a reference time. $\chi$ in continuum mechanics makes assumptions about continuity and invertibility that do not apply with discrete elements. Thus this more general $D$ is proposed.

An element is a restriction on the functional form of $D$ for a discrete object which dictates the functional form for all the infinite number of points contained in the set based on a discrete set of control points. There are many types of restrictions that are common in numerics, below are a few including the discrete element.
4.1.1 Discrete Elements and Rigid Bodies

To recover the notion of a discrete element as a kinematically rigid sphere with homogeneous properties a few constraints are added to the possible functional forms that $D$ can take. First, the rigid body is ensured by adding constraint that every point must remain the same distance away from every other point in the set at all times. Mathematically this is expressed as:

$$\|x^k - x^j\|_2 = \|y^k - y^j\|_2 = 0 \forall x^k, x^j, y^k, y^j \in Z_i \forall t_0, t_1 \quad (4.1)$$

where $\|\cdot\|_2$ is the euclidean norm in $\mathbb{R}^{n_{dim}}$. This does not have, at the moment, a notion of geometry. A geometry is set by choosing a mapping at a given time called it $X = D(Z_i, t_0)$ is a reference position at $t_0$. With the rigid body constraint this determines the geometry for all time. Also, the mapping determines continuity. By this definition it would be perfectly acceptable to define a rigid body that is has parts that are completely unconnected. If continuity is desired, $X$ that is continuous over set $Z_i$ should be chosen. Here, a spherical initial condition will be imposed.

The Discrete element also assumes that the other data in the mapping $D$, such as temperature or charge distribution, are constant across $Z_i$. So $D_k(Z_i, t) = C_{i,k}(t)$ where $D_k$ is a component of the mapping and $C_{i,k}$ only depends on time, the discrete object and the component but not $Z_i$. The rigid body constraint can also be view as a constant property by recognizing that the rigid body constraint is synonyms with choosing a mapping that within an element $i$, $x = Q_i(t)X + c_i(t)$ where $Q$ is a rotation and $c$ is a translation are components of $D$ that are constant across the element $i$.

Finally, it is generally held that two material points can not occupy the same point in space. This requirement is not enforced globally in the definition of the discrete element. Here with the spherical starting geometry, the additional requirement that that the $x$ portion of $D(Z_i)$ be invertible only within the set is required. This guarantees that that the material points within the set $Z_i$ do not exist at the same point in space. Because the discrete elements are “soft” in that the constitutive models are allow for overlap of the elements, it is not required that $D$ as whole be invertible which would prevent the overlap/soft behavior.

4.1.2 Pseudo-Rigid Bodies

Another popular element in granular dynamics which has been studied extensively is the pseudo rigid body [21, 22, 28, 27, 25, 26, 56, 76, 90, 89]. A pseudo rigid body is defined by a homogeneous deformation gradient. Where the deformation gradient $F = \frac{dX}{dx}$ where
\( \mathbf{x} \) is the section of the mapping for spatial position at a given time and \( \mathbf{X} = \mathbf{D}(\mathbf{Z}_i, t_0) \) is a reference position at \( t_0 \) as is standard in notation for a continuum formulation. This assumption makes it so that all the infinite points in the element must move in a way that \( \mathbf{F} \) is homogeneous. As Casey pointed out[21, 22] as per Antman and Marlow[7] it is interesting to view homogeneous \( \mathbf{F} \) as a material constraint and thus there is a corresponding reactive stress that appears which enforces it. Casey also points out that the magnitude of the reactive stress is a measure of how close the unconstrained solution with infinite points is to the constrained system with the restrictions imposed by the element.

4.1.3 Nodal Finite Elements

Likewise with Finite Elements, an element corresponds discrete sets \( \mathbf{D}(\mathbf{Z}_i) \) but the mapping \( \mathbf{D} \) in that the positions of \( \mathbf{x} \) within the set are restricted to the interpolation based on a sum of shape function. Specifically within each element the mapping is required to take the form

\[
\mathbf{D}_j(\mathbf{Z}_i) = \sum_{k=1}^{N_{\text{nodes}}} a_{jk} \phi_{jk} \quad \text{with no implied sum over } j \text{ and } \phi_j \text{ need not be the same for different components, } j \text{ or elements } i \text{ though they often are. Traditionally, this constraint of state of the infinite points is enforced strongly by substituting the mapping into the governing equations. It would also be possible to enforce this using a local material constraint as was performed in pseudo-rigid bodies. Here again a reactive stress would exist that enforces the interpolation constraint a notion that was recognized by Antman and Marlow[7]. The reaction force is how much extra stress had to be added so that the body obeys the constraint, in this case the interpolation function established by the sum of the shape functions. The relative size of the reaction force indicates how much the constraint is effecting the solution compared to the unconstrained problem.}

With regard to the requirement that material points not occupy the same spatial point it should be noted that in the nodal finite element formulation this is not required globally or within an element. In other words \( \mathbf{D}(\mathbf{Z}_i) \) is not required to be invertible. The loss of invertability is called element inversion and can lead to incorrect solutions. Element inversion is avoided using heuristic metrics which are used to gauge whether an element is close to inversion. If it is, the domain is rediscretized by making new sets of \( \mathbf{Z}_i \) to which the element shape functions are applied to constrain the motion of its infinite points such that the mapping is no longer likely to lose to become uninvertible.

4.2 Dynamics and Rigid Body

The rigid body constraint for the discrete element using constant distances as is valid but in practice it is more efficient to use an equivalent definition that a rigid body can be parameterized by a rotation and a translation. In this way only \( 2n_{\text{dim}} \) degrees of freedom are need to fully describe position of every point point in set \( \mathbf{Z}_i \) associated to the discrete element. \( n_{\text{dim}} \) are the center of mass position and the other \( n_{\text{dim}} \) represent the angular
orientation of the points around the center of mass. Another $2n_{\text{dim}}$ are also needed to store the center of mass velocity and angular velocity to fully describe the dynamical state of the rigid body.

In Newtonian dynamics, the momentum of a rigid body is determined by the ODE

$$\frac{dp_i}{dt} = F_{i,\text{tot}}$$  \hfill (4.2)

where $p_i \in \mathbb{R}^{n_{\text{dim}}}$ is the momentum of the $i^{th}$ element and $F_{i,\text{tot}} \in \mathbb{R}^{n_{\text{dim}}}$ is the net force on the $i^{th}$ element. $n_{\text{dim}}$ is the spatial dimension of the system e.g. in 3-Dimensions, $n_{\text{dim}} = 3$. It is also observed in Newtonian mechanics that the force between two points is equal and opposite, so $F_{ij} = -F_{ji}$. Taking the momentum to be classical and non-relativistic, $p_i = m_i v_i$, where $m_i \in \mathbb{R}$ is the mass of the $i^{th}$ particle and $v_i \in \mathbb{R}^{n_{\text{dim}}}$ is the velocity of the $i^{th}$ particle. Plugging in the momentum into equation 4.2 and applying the chain rule gives

$$m_i \frac{dv_i}{dt} + \frac{dm_i}{dt} v_i = F_{i,\text{tot}}$$  \hfill (4.3)

and if the mass does not change, as is usually assumed, this becomes the familiar

$$m_i \frac{dv_i}{dt} = F_{i,\text{tot}}.$$  \hfill (4.4)

Integrating this gives the velocity of the $i^{th}$ particle. Finally, the definition of the velocity is the time derivative of position

$$\frac{dr_i}{dt} = v_i.$$  \hfill (4.5)

$r_i$ is the position of the rigid body ($r$ is used to distinguish the discrete position from the continuous position, $x$). Integrating equations 4.4 and 4.5 gives both the position and velocity of the particle.

The rotational state of the particle is found using

$$\frac{dL_i}{dt} = \tau_{i,\text{tot}}$$  \hfill (4.6)

where $\tau_{i,\text{tot}} \in \mathbb{R}^{n_{\text{dim}}}$ is the torque and $L_i \in \mathbb{R}^{n_{\text{dim}}}$ is the angular momentum. Assuming a classical non-relativistic angular momentum, $L_i = J_i \omega_i$ where $\omega_i \in \mathbb{R}^{n_{\text{dim}}}$ is the angular velocity and $J_i \in \mathbb{R}^{n_{\text{dim}} \times n_{\text{dim}}}$ is the moment of inertia tensor. Plugging the angular momentum into 4.6 and applying the chain rule results in

$$\frac{dL_i}{dt} = \frac{dJ_i \omega_i}{dt} = J_i \frac{d\omega_i}{dt} + dJ_i \frac{dt}{dt} \omega_i = \tau_{i,\text{tot}}.$$  \hfill (4.7)

However, the components of $J_i$ matrix representation are constantly changing as the body rotates in the fixed coordinate system making the above equation difficult to use. Expressing the problem in the coordinates of the moving reference frame of the body
denoted by the prime, where the momentum of inertia tensor representation is constant for a rigid body which does not change shape or mass, $\frac{dJ}{dt} = 0$, 

\[
\left( \frac{d\mathbf{L}_i}{dt} \right)_{\text{spatial}} = \left( \frac{dJ_i\omega_i}{dt} \right)_{\text{spatial}} = \left( \frac{dJ_i\omega_i'}{dt} \right)_{\text{body}} + \omega'_i \times J'_i\omega'_i = \frac{J'_i d\omega'_i}{dt} + \frac{dJ'_i}{dt} \omega'_i + \omega'_i \times J'_i\omega'_i = \frac{J'_i d\omega'_i}{dt} + \omega'_i \times J'_i\omega'_i
\]

thus the differential equation which governs the angular velocity in the body frame is

\[
J'_i \frac{d\omega'_i}{dt} = \tau'_{i,tot} - \omega'_i \times J'_i\omega'_i
\]

For spherical elements, the moment of inertia is diagonal so $\omega'_i \times J'_i\omega'_i = 0$, yielding

\[
J'_i \frac{d\omega'_i}{dt} = \tau'_{i,tot}
\]

and dropping the primed notation as the form is the same in either frame again yields

\[
J_i \frac{d\omega_i}{dt} = \tau_{i,tot}.
\]

This same result can be more quickly derived by noting that for a spherical element the momentum of inertia in the lab frame is also constant due to symmetry so $\frac{dJ}{dt} = 0$ thus

\[
\frac{d\mathbf{L}_i}{dt} = J_i \frac{d\omega_i}{dt} = J_i \frac{d\omega_i}{dt} + \frac{dJ_i}{dt} \omega_i = \tau_{i,tot}
\]

but the full version is included to alleviate potential confusion with often cursorily described formulations for angular velocity. Also for a spherical body with constant density, $J_i = \frac{2}{5} Mr^2 I$ where $I$ is the $\mathbb{R}^{n_{dim}} \times \mathbb{R}^{n_{dim}}$ identity matrix. So only the scaler $J_i$ is needed

\[
J_i \frac{d\omega_i}{dt} = \tau_{i,tot}
\]

The rotational position can be expressed in a number of ways such as with Euler angles[93] or a quaternion description[15] and found via integration of the angular velocity with the appropriate equations. Here, the element of interest is spherical which are symmetric so the angular position contributes no additional information, as such will not be discussed here.
For spherical rigid bodies, the set of ODEs to solve to get the state of the \(i^{th}\) particle is

\[
\frac{dr_i}{dt} = v_i \\
\frac{dv_i}{dt} = \frac{1}{m_i} F_{i,tot} \\
\frac{d\omega_i}{dt} = \frac{1}{J_i} \tau_{i,tot}
\]

or

\[
\frac{dr_i}{dt} = v_i \\
\frac{dv_i}{dt} = \frac{1}{m_i} F_{i,tot} \\
\frac{d\omega_i}{dt} = \frac{1}{J_i} \tau_{i,tot}
\] (4.8)

The two spatial ODEs (top two) are obviously coupled through the velocity. However, in general the system is fully coupled as \(F_{i,tot}(r_{1\rightarrow N}, v_{1\rightarrow N}, \omega_{1\rightarrow N})\) and \(\tau_{i,tot}(r_{1\rightarrow N}, v_{1\rightarrow N}, \omega_{1\rightarrow N})\) noting that the forms of these functions must be appropriately selected as to obey objectivity (invariance to shift in frame by rotation or translation).

For an \(N\)-body system of \(N\) elements there is a set of ODEs for each and each has the potential to be coupled to the other. Using the variables without subscripts to denote the complete state of the global system

\[
\begin{align*}
\mathbf{r} &= \begin{pmatrix} r_1 \\ r_2 \\ \vdots \\ r_N \end{pmatrix}, \\
\mathbf{v} &= \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_N \end{pmatrix}, \\
\mathbf{\omega} &= \begin{pmatrix} \omega_1 \\ \omega_2 \\ \vdots \\ \omega_N \end{pmatrix}
\end{align*}
\] (4.9)

\[
\mathbf{J} = \begin{pmatrix} J_1 I & 0 & 0 & 0 \\ 0 & J_1 I & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & J_1 I \end{pmatrix}, \\
\mathbf{M} = \begin{pmatrix} m_1 I & 0 & 0 & 0 \\ 0 & m_2 I & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & m_N I \end{pmatrix}
\]

where \(I\) is the \(\mathbb{R}^{n_{dim}} \times \mathbb{R}^{n_{dim}}\) identity matrix. The full systems dynamics is determined by

\[
\begin{align*}
\frac{d\mathbf{r}}{dt} &= \mathbf{v} \\
\mathbf{M} \frac{d\mathbf{v}}{dt} &= \mathbf{F}_{tot} \\
\mathbf{J} \frac{d\mathbf{\omega}}{dt} &= \mathbf{\tau}_{tot}
\end{align*}
\] (4.10) (4.11) (4.12)
where are the inverses of the diagonal are easily evaluated as the inverse of the diagonal components. Again a full coupling may occur as in general $F_{tot}(r, v, \omega)$ and $\tau_{tot}(r, v, \omega)$ again noting that the form of such relations should be formulated to preserve objectivity.\cite{93}

4.2.1 Argument for Neglecting Rotational Degrees of Freedom

For the simulations in this dissertation, the rotational degrees of freedom will be neglected ($\omega = 0$). The reason for this is two fold one physical and one computational. Physically, the systems of interest here are additive manufacturing processes namely SLS/SLM and LMD. The goal of these simulations is to model the packing and deposition processes and the resulting state at the end of the process. A couple arguments can be made for why the reduced order system should not significantly effect the results of a deposition simulation. First, the particles in the system, once deposited, are generally in a densely packed system of other particles. The result is that on average the torque applied by one particle is nullified by the interaction with another. Also, the viscous drag models used for friction scale with velocity and at steady state when deposition is finished, the velocity is zero and thus no torques in the model exist so the rotational degrees of freedom play no role.

A major interest in using discrete elements is for their computational efficiency in exchange for a reduced order solution space. Including the rotational degrees of freedom would only add to the computational complexity of the model. The goal is to find models which are helpful in understanding engineering processes and as such it is acceptable to explore reduced order models with simplistic physics as a means of finding engineering insight in a computationally efficient manner.

Without the rotational DOF the system is reduced to

\[
\frac{dr}{dt} = v \quad (4.16)
\]

\[
\frac{dv}{dt} = M^{-1}F_{tot} \quad (4.17)
\]
or for a single element

\[
\frac{d\mathbf{r}_i}{dt} = \mathbf{v}_i \\
\frac{d\mathbf{v}_i}{dt} = \frac{1}{m_i} \mathbf{F}_{i,tot}
\] (4.18)

with \( \mathbf{F}_{i,tot}(\mathbf{r}, \mathbf{v}) \) with \( \mathbf{r}, \mathbf{v} \) defined as in equation 4.9. Some consequences of this reduction will be explored in later sections.

### 4.3 Thermal Formulation

#### 4.3.1 Energy Balance

For any system with conserved energy, \( \frac{d}{dt} (KE + PE) = W_{tot} \) where \( KE \) is the kinetic energy, \( PE \) is the potential energy and \( W_{tot} \) is the total work rate. For the particle system like discrete elements

\[
\frac{d}{dt} \left( \sum_i \left[ KE_i + PE_i + \sum_j PE_{ij} \right] \right) = W_{tot}
\]

The kinetic energy of the rigid body is

\[
KE_i = \frac{1}{2} M v_i^2 + \frac{1}{2} I_i \omega_i^2
\]

The potential energy is

\[
PE_i = PE_{i,int} + PE_{i,grav} = \int_{\Omega_i} \rho E dV + m_i g h_i
\]

where \( E \) is the internal energy capacity. The rigid body has no internal mechanical potential because as rigid it does not deform thus the stress power term is 0. And \( PE_{ij} \) is the energy stored in the element-element interaction models. The total work rate of the system is given by

\[
W_{tot} = \sum_i \sum_j W_{loss,ij} + \sum_i Q_{i,source}
\]

were \( W_{loss,ij} \) is the loss in energy in the system due the dissipative inter-element forces. \( Q_{i,source} \) are body terms that heat the element such as internal reactive terms. The heat transfer between element \( Q_{ij} \) is conservative so and \( \sum_i \sum_j Q_{ij} = 0 \), adding zero does not change anything so

\[
W_{tot} = \sum_i \sum_j W_{loss,ij} + \sum_i Q_{i,source} + \sum_i \sum_j Q_{ij}
\]
Here it is assumed that the dissipative energy, $W_{\text{loss},ij}$, is lost to the environment as heat and does not effect the internal potential energy. Thus the mechanical and thermal energy can be decouple to get

$$\frac{d}{dt} \left( \sum_i KE_i + P E_{i,\text{grav}} + \sum_i \sum_j P E_{ij} \right) = \sum_i \sum_j W_{\text{loss},ij}$$

$$\frac{d}{dt} \left( \sum_i P E_{i,\text{int}} \right) = \sum_i Q_{i,\text{source}} + \sum_i \sum_j Q_{ij}$$

Rewriting the bottom equation as

$$\sum_i \left[ \frac{d}{dt} (P E_{i,\text{int}}) - Q_{i,\text{source}} - \sum_j Q_{ij} \right] = 0$$

because this is arbitrary it yields for a single element, $i$, that

$$\frac{d}{dt} (P E_{i,\text{int}}) = Q_{i,\text{source}} + \sum_j Q_{ij}$$

denoting $Q_{i,tot} = Q_{i,\text{source}} + \sum_j Q_{ij}$ and substituting in $P E_{i,\text{int}}$ gives

$$\frac{d}{dt} \left( \int_{\Omega_i} \rho E dV \right) = Q_{i,tot}$$

Since the body is rigid, the derivative moves through the integral, and with density constant this becomes

$$\int_{\Omega_i} \rho \frac{dE}{dt} dV = Q_{i,tot}.$$ 

Assuming the model for internal energy capacity $E = C_i \Theta$ where $C_i$ is a constant heat capacity for element, $i$, and $\Theta$ is the temperature.

$$\int_{\Omega_i} \rho C_i \frac{d\Theta}{dt} dV = Q_{i,tot}.$$ 

Finally since for a discrete element, it is assumed the temperature is constant, $\Theta = \Theta_i$, over the body so the integrand is independent of the term inside giving

$$\rho C_i V_i \frac{d\Theta_i}{dt} = Q_{i,tot}$$

and with $\rho V_i = m_i$ this becomes the final governing ODE

$$C_i m_i \frac{d\Theta_i}{dt} = Q_{i,tot} \quad (4.19)$$

$$\frac{d\Theta_i}{dt} = \frac{1}{C_i m_i} Q_{i,tot} \quad (4.20)$$
where \( Q_{i,tot} \) [Watts] is the power applied to the body by flux through a surface eg. conduction or heat sources eg. joule heating or laser. \( C_i \) is the specific heat capacity with units \([J/kg/K]\).

For the global system

\[
\frac{d\Theta}{dt} = H^{-1} Q_{tot}
\]

\[
\frac{d\Theta}{dt} = Q_{tot}
\]

where

\[
\Theta = \begin{pmatrix}
\Theta_1 \\
\Theta_2 \\
\vdots \\
\Theta_N
\end{pmatrix},
Q = \begin{pmatrix}
Q_{1,tot} \\
Q_{2,tot} \\
\vdots \\
Q_{N,tot}
\end{pmatrix},
H = \begin{pmatrix}
m_1 C_1 & 0 & 0 & 0 \\
0 & m_2 C_2 & 0 & 0 \\
0 & 0 & \ddots & 0 \\
0 & 0 & 0 & m_N C_N
\end{pmatrix}
\]

### 4.3.2 Thermal Expansion

Thermal expansion is potentially significant effect in additive manufacturing due to the large range in operating temperatures. The rigid body definition is enriched with an equation of state namely linear thermal expansion. For this, constraint on the element motion is modified to become

\[
x^k = D(Z^k_i, t_0)
\]
\[
x^j = D(Z^j_i, t_0)
\]
\[
y^k = D(Z^k_i, t_1)
\]
\[
y^j = D(Z^j_i, t_1)
\]
\[
\Theta_{i,0} = \Theta_i(t_0)
\]
\[
\Theta_i = \Theta_i(t_1)
\]

\[
(1 + \alpha_{therm}(\Theta_i - \Theta_{i,0})) \left\| x^k - x^j \right\|_2 - \left\| y^k - y^j \right\|_2 = 0 \forall x^k y^k y^j \in Z_i \forall t_0, t_1
\]

For a sphere with a constant temperature, linear radial expansion as below

\[
r_i(T) = r_{\Theta_0,i}(1 + \alpha_{therm}(\Theta_i - \Theta_0)) \quad (4.21)
\]

satisfies this constraint that \((1+\alpha_{therm}(\Theta_i - \Theta_{i,0})) \left\| x^k - x^j \right\|_2 - \left\| y^k - y^j \right\|_2 = 0 \forall x^k y^k y^j \in Z_i \forall t_0, t_1\), which can be seen by looking at the spherical coordinate representation of the positions

\[
x_1 = r \cos \theta \sin \phi
\]
\[
x_2 = r \sin \theta \cos \phi
\]
\[
x_3 = r \cos \phi.
\]
Thus the thermally expanded positions are
\begin{align*}
y_1 &= (1 + \alpha_{\text{therm}}(\Theta_i - \Theta_{i,0})) r \cos \theta \sin \phi \\
y_2 &= (1 + \alpha_{\text{therm}}(\Theta_i - \Theta_{i,0})) r \sin \theta \cos \phi \\
y_3 &= (1 + \alpha_{\text{therm}}(\Theta_i - \Theta_{i,0})) r \cos \phi.
\end{align*}

From this it can be seen that in the euclidean norm \( \| \mathbf{y}^k - \mathbf{y}^j \|_2 \), the \((1 + \alpha_{\text{therm}}(\Theta_i - \Theta_{i,0}))\) can be factored out, so \( \| \mathbf{y}^k - \mathbf{y}^j \|_2 = (1 + \alpha_{\text{therm}}(\Theta_i - \Theta_{i,0})) \| \mathbf{x}^k - \mathbf{x}^j \|_2 \) and the constraint is fulfilled. Practically, this will effect the simulations be changing the radius of the elements.

### 4.4 Combined DEM Formulation

For a system of multiphysics discrete elements, the full system of constraints on the motion of the continuum is

\[
\frac{d\mathbf{r}_i}{dt} = \mathbf{v}_i \\
\frac{d\mathbf{v}_i}{dt} = \frac{1}{m_i} \mathbf{F}_{i,\text{tot}} \\
\frac{d\Theta_i}{dt} = \frac{1}{C_i m_i} \mathbf{Q}_{i,\text{tot}} \\
\mathbf{F}_{ij} = -\mathbf{F}_{ji} \\
(1 + \alpha_{\text{therm}}(\Theta_i - \Theta_{i,0})) \| \mathbf{x}^k - \mathbf{x}^j \|_2 - \| \mathbf{y}^k - \mathbf{y}^j \|_2 = 0 \forall \mathbf{x}^k \mathbf{x}^j \mathbf{y}^k \mathbf{y}^j \in \mathbf{Z}_i, \forall t_0, t_1 \\
\mathbf{D}(\mathbf{Z}_i) \text{ invertible for each } i
\]

where \( \mathbf{F}_{i,\text{tot}}(\mathbf{r}, \mathbf{v}, \Theta) \) and \( \mathbf{Q}_{i,\text{tot}}(\mathbf{r}, \mathbf{v}, \Theta) \) are potentially fully coupled.

### 4.5 Meshes

Usually discrete elements along with their counterparts smooth particle hydrodynamics and molecular dynamics are usually described as meshless methods [63, 77, 87]. However, the term “meshless” this is a bit of a misnomer. Meshes are merely a data structure that can be iterated over which serves the purpose of storing the connection between sections of the discretization and inform the simulation as to what pieces of information should be used during a particular calculation. For example, in finite elements, the mesh stores which nodes will go into calculation for an element. Meshes of this description definitely exist in discrete element in the form of which elements are contacting and what data is used for calculating the interactions between them. Below are a couple of mesh data structures that are useful in efficiently determining between which elements the contact and interaction models should be evaluated.
4.5.1 “N\textsuperscript{2}” Mesh

The simplest of the iterable mesh data structures is the “N\textsuperscript{2}” mesh. This is where each item in a set is check against every other item in a set. This is called the N\textsuperscript{2} check because for a set checked against itself, the cost scales as O(N\textsuperscript{2}) where N is the number of items in the set. With this, every discrete element attempts to interact with every other in a set. The constitutive law is then responsible for turning interaction on and off. A classic example of this is collision. When simulating granular material contact occurs if the discrete elements overlap, a force is applied as seen in the power law contact model in equation 5.5. If the elements do not overlap then the force law returns zero. In this way every particle interacts with every other but most return zero force interaction.

The N\textsuperscript{2} is often vilified as having poor scaling which is true but it is not always the worst option. First, for small number of items the difference is not particularly significant. Also for contact, in theory every element should have the potential to interact with every element which is exactly represented by this mesh structure. As previously noted, the scaling can be undesirable for large problems but small problems with the N\textsuperscript{2} mesh is a very good way to create benchmark solutions which guarantee a solution without any missed contacts. This can then be used to check other implementations with more efficient data structures. Finally, the N\textsuperscript{2} mesh may be acceptable if the cost to evaluate the constitutive model is much greater than the overlap calculation. The cost to evaluate this situation requires

\[
\text{Cost} = \alpha N^2 + \beta m N
\]

where \(\alpha\) is the cost to determine contact and \(\beta\) is the cost evaluate the constitutive law if a contact is found. For constitutive models with a cut off, \(m\) is typically a O(1) number of other elements that a particular element is in contact with. \(m\) in DEM is typically between 8 and 20 contacts per element. Often times, \(\beta \gg \alpha\) and so the O(N\textsuperscript{2}) can still be a small part of the cost. The two cost only become comparable for a particular \(N\) if

\[
\alpha N^2 \approx \beta m N
\]

\[
\beta \approx \frac{\alpha N}{m}
\]

so if \(\beta\) is on the order of \(\alpha N/m\) or larger than the overlap check may be negligible compared to the constitutive law calculation and the generally undesirable N\textsuperscript{2} scaling should not be worried about. However, \(\beta\) is generally constant so for large enough \(N\) the N\textsuperscript{2} will dominate for a larger system.

If for some reason \(\beta\), the cost of the constitutive law, scales with \(N\) also then this will change and it is possible the constitutive law will always be as costly as the contact check. One might imagine a situation where the constitutive law is a function that depends on all other particles in the system. This is true for long range forces a problem usually handled by setting cutoff distances as with the Lenard-Jones potential or using other approximation methods such as multipole method [29, 78]. In molecular dynamics, multi-
body interactions are common but usually chosen to only involve a few particles for cost reasons and because the limited interactions are shown to be physically appropriate.

In the DSL, the $N^2$ is created using the NvN mesh with two connectivities and is the origin of the name.

### 4.5.2 Verlet Lists

Another way to store which elements are likely to interact is the Verlet list[100]. This is essentially a list of connectivities one would see in a traditional finite element implementation. It is the same in that is a list of integers which indicates two items which should be involved in a computation. The difference as I see it with a traditional finite element is that with the finite element mesh it stores exactly which nodes are supposed to interact. With a contact Verlet list only some of the potential interactions are stored, namely those that are physically close together. Verlet lists are updated periodically as the particles move and new particles move into the vicinity of particle. This must be done frequently enough so that no contacts are missed. If the Verlet list had $N^2$ entries it would be the same as the finite element mesh as it would store exactly which elements should be checked. This however would negate the computational savings of the Verlet list is intended to deliver by limiting the number of required checks.

Here it should also be noted that re-meshing as done in finite elements actually encompasses two steps which are rarely delineated but matters in the context of discrete elements. The first step is more rightly described by the word re-discretization. In this step, a new set of elements are created which in turn determine the functional form which constrain the state that the continuum of points can exist. The next step is to create the mesh which indicates that the nodes are connected to enact these constraints over the element. By using the same nodes for the adjacent elements continuity is enforced.

The building of Verlet lists only amounts to the second step. The elements and the constraints on the state of the material points they enforce are unchanged. All that has been update are how the discretization interact with each other which is the second step. Re-discretization is not bad but it should be recognized it fundamentally changes the solution spaces. Rediscretizing is occasionally required because the constraint that is the element becomes unphysical such as when an element inverts. Rediscretizing allows for the tracking of the continuum in a way that obeys the physics.

Verlet list for discrete elements can be created using a number of algorithms, a $N^2$ search or binning are a couple of examples. The $N^2$ search is as above where close elements are found by looking over every element. While this will still scale with as $O(N^2)$ but does not need to be done every timestep and thus computation is save. For binning, described in the next section, an efficient data structure based on locality is created which can be quickly traverse to create a Verlet list. When the elements have moved enough, the Verlet list can be remade with the same(or different) algorithm to ensure no contacts are missed. Once a Verlet list is made it can be looped over in $O(N_{verlet})$ where $N_{verlet}$ are the number of connections in the list.
Another advantage of a Verlet list beyond speed is that data can associated to a connection itself since it is now actively tracked. This is useful for connections that have a history component like stick-slip model of friction or if the connection can created or destroyed (the connection data represents bond status in this case). When tracking data, it is required that when a new Verlet list is made, old Verlet list data must be associated and copied to entries in the new Verlet List.

With the DSL, Verlet list can be implemented using the Variable length connectivity which is then used as the one connectivity used in an NvN mesh. The length $N_{\text{verlet}}$ stores the length of the current list and is the second input to the Variable length connectivity constructor.

4.5.3 Binning and Linked List

To avoid the $N^2$ scaling, other schemes can be used to reduce the cost of building the Verlet list. One such scheme is binning[71]. Here, the physical space that the computation is occurring in is broken up into sections, most commonly cuboid. Each of the sections are referred to as bins. A particle is associated to a bin based on the position of its center. Knowing the bins are next to each other, it is seen that the particle could only possibly interact with elements in its bin or bins that are adjacent. Contact of this element must only be checked against particles (if there is one) in the 26 other bins surrounding it in 3D, or the 8 bins surrounding it in 2D. If the particle is thought of as being self interacting then contact with itself must also be checked, this is not generally the case for granular media but may be needed in the case of methods such as SPH. To determine the bin of each particle is $O(N)$ as each particle must be looped past and bin determined. Then assuming here are approximately constant number of elements in each bin, a Verlet list of $\approx mN$ length is create where $m$ is th number of particles times adjacent bins and $m \ll N$. The bin size is determined by the time between recreation of the Verlet list and how fast the elements are moving. The bin must be large enough that no possible contacts are missed between the two times the lists are built. It should also be noted that instead of building Verlet lists with the bins, this iterable data structure could be used directly within a simulation as mesh too. At the moment, the DSL has no native treatment of bins and thus a Verlet lists must made for input into the DSL.

To produce the bin data structure one could use library however here an implementation in Python has been written by the author based of the linked list data structure. In such a system, a linked list of particles is made for each bin. When a element is first found to be in the bin it is made to be the head of the linked list. When subsequent elements are found to be in the bin, it is made the new head and it points to the previous head of the linked list. When contact checks are performed between bins, the link list is merely traversed and each element in the list and its counterpart in a adjacent bin are placed in a Verlet list connection.

A good data structure for storing this link list data is one array which is shaped to have $n$ indexes, where $n$ is the spatial dimension, and one array which is the length of the
number of discrete elements, $N$. The $n$-index array is created to have a 1-1 correspondence with a bin and stores the head of the bin. It is initialized to -1 and 1D array is also initialized to -1. When the first particle in a bin is found the $n$-index array value is set to be that particle index, $I_{p1}$, of the particle. When next particle, index $I_{p2}$, is found to be in a bin, index $I_{p2}$ in the 1D array is set to be the value $I_{p1}$, and the $n$-index array is set to the value $I_{p2}$. If a third element is found, index $I_{p3}$, is found index $I_{p3}$ in the 1D array is set to value $I_{p2}$, and the $n$-index array value is set to $I_{p3}$. To traverse the list, the index in the $n$-index array points to the next position in the 1D array. This now stores index $I_{p2}$, contact is checked with this particle, and the next particle to check is found in the 1D array at index $I_{p2}$, which now stores $I_{p1}$. Contact is check with particle $I_{p1}$ and the list looks to position $I_{p1}$. Nothing has been stored here and a value of -1 is found, which indicates the end of the link list has been found the loop should be broken.
5.1 Sphere Contact With Line, Rectangle and Cuboid

For each constitutive interaction model in the next section, it is assumed that there is a scalar overlap, \( \delta \), and outward normal, \( \hat{n}_c \), of the body at the point of contact. The body the sphere is interacting with can either be a point, line, rectangle or cube. The overlap and outward normal of the rigid body in this section are determined by

\[
\delta = R_s + R_b - ||r_c||_2 \tag{5.1}
\]

\[
\hat{n}_c = \frac{r_c}{||r_c||_2} \tag{5.2}
\]

where \( R_s \in \mathbb{R} \geq 0 \) is the radius of the sphere and \( R_b \in \mathbb{R} \geq 0 \) is the effective radius of the body with which it is interacting. These are set as parameters by the individual running the model. The vector from the closest point on the geometric description of the body to the center of the sphere is \( r_c \in \mathbb{R}^{n_{\text{dim}}} \) where \( n_{\text{dim}} \in \mathbb{Z} \) is the spatial dimension of the problem. When \( \delta > 0 \) then the two objects are said to be in contact with one another, which is shown in Figure 5.1 for spheres. Note: If \( R_s = R_b = 0 \), it is guaranteed that \( \delta \leq 0 \) so points, by this definition are never in contact; for all problems of interest one or both radii will be non-zero. The form of \( n_c \) is chosen so that when \( R_s = 0 \) or \( R_b = 0 \), the normal vector is still well defined and consistent with \( \lim R_s \to 0 \) or \( \lim R_b \to 0 \). If \( R_b \neq 0 \) the effect is the geometry is thickened by radius \( R_b \) and “sharp” edges have rounded edges/corners or radius \( R_b \). Eg. a line with a radius will have semi-spherical end-caps.

For this to be useful, \( r_c \) must be found which can be done analytically for a few simple rigid bodies: points, line segments (embedded in 2-D or higher), plane segments (2-D rectangle in 2-D or higher) and cuboid (embedded in 3-D or higher). The following sections will show how to find \( r_c \) for these rigid bodies. Note: The title of the section is “sphere” contact but all contact formulations work for circles in 2-D as well.
5.1.1 Sphere-point overlap

First, and mostly easily is the $r_c$ for a sphere and a point. The $r_c$ is just the center to center vector

$$r_c = r_{ij} = r_j - r_i$$

where the index $i$ is used to denote the sphere of interest and $j$ is the index of the body that sphere with index $i$ is interacting with. This is shown in Figure 5.2.

5.1.2 Sphere-Line Segment

Likewise for a line segment, the point closest to the sphere can be determined. This is accomplish by finding the closest point on the infinite line via a projection and then checking if it is outside the line segment. If it does fall outside segment then the closest point is on the end of the line segment. The line segment is described by a point, $r_j$, a unit director $d_1$, and two scalars $l_{min}$ and $l_{max}$ which are the minimum and maximum extents of the line segment along director $d_1$. With this information $r_c$ is defined by

$$\|r_{ij}\| = (r_j - r_i) \cdot d_1$$

$$r_{ij} = [(r_j - r_i) \cdot d_1] d_1$$

$$r_{ij,lim} = min (max (r_j - r_i) \cdot d_1, l_{min}), l_{max} d_1$$

$$r_c = r_j + r_{ij,lim} - r_i$$

$$r_c = r_j + min (max (r_j - r_i) \cdot d_1, l_{min}), l_{max} d_1 - r_i$$

This concept is shown in Figure 5.3. Note that if $l_{min} < \|r_{ij}\| < l_{max}$ then

$$r_c = r_j + (r_j - r_i) \cdot d_1 - r_i$$

$$= r_j - r_i + [(r_j - r_i) \cdot d_1] d_1$$

$$r_c = r_{ij,\perp} = r_{ij} - r_{ij}\parallel$$

and the expected projection to the line is recovered. The min and max are responsible for handling the endcap condition.
Figure 5.2: Finding the closest point between a sphere and a point.

Figure 5.3: Finding the closest point on a line segment.
5.1.3 Sphere-Rectangle

For a rectangle, the closest point is again found by projecting onto the plane now defined by a point, \( r_j \), two orthogonal unit directors \( d_1 \) and \( d_2 \), and four scalars \( l_{k,\text{min}} \) and \( l_{k,\text{max}} \) for \( k = 1, 2 \) which are the minimum and maximum extents of the rectangle along director \( d_1 \) and \( d_2 \). With this information \( r_c \) is defined by

\[
\begin{align*}
  r_{ij} &= r_j - r_i \\
  l_{d1} &= \min(\max(r_{ij} \cdot d_1, l_{1,\text{min}}), l_{1,\text{max}}) \\
  l_{d2} &= \min(\max(r_{ij} \cdot d_2, l_{2,\text{min}}), l_{2,\text{max}}) \\
  r_{ij,\text{lim}} &= l_{d1}d_1 + l_{d2}d_2 \\
  r_c &= r_j + r_{ij,\text{lim}} - r_i \\
&= r_j + l_{d1}d_1 + l_{d2}d_2 - r_i
\end{align*}
\]

The procedure is to use a projection to find the closest point in the plane and then restrict the position back onto the rectangle shown in Figure 5.4. If an infinite wall is desired then \( l_{\text{min}} = -\infty \), \( l_{\text{max}} = \infty \) the result is that the min and max always yield \( r_{ij} \cdot d_{1,2} \), thus

\[
\begin{align*}
  r_c &= r_j + (r_{ij} \cdot d_2) d_1 + (r_{ij} \cdot d_2) d_2 - r_i \\
&= r_j + r_{ij\parallel} - r_i \\
&= r_{ij\perp}
\end{align*}
\]

which means that it would be easier to described the infinite wall by just \( r_j \) and the normal of the wall \( n_{\text{wall}} \). In this case, \( r_c = r_{ij\perp} = r_{ij} \cdot n_{\text{wall}} \) and \( n_c = \frac{r_c}{\|r_c\|} = n_{\text{wall}} \).
5.1.4 Sphere-Cuboid

The procedure extends to cuboid. Again, the closest point is found by the projecting on to a body's orthonormal basis described by an origin point, \( r_j \), three orthonormal unit directors \( d_k \) and 3 sets of scalars \( l_{k,\text{min}} \) and \( l_{k,\text{max}} \) which are the minimum and maximum extents of the rectangle along director \( d_k \) where \( k=1,2,3 \). Again with this information \( r_c \) is defined by projecting onto this basis and then limiting the distance shown below

\[
\begin{align*}
  r_{ij} &= r_j - r_i \\
  l_{d1} &= \min(\max(r_{ij} \cdot d_1, l_{1,\text{min}}), l_{1,\text{max}}) \\
  l_{d2} &= \min(\max(r_{ij} \cdot d_2, l_{2,\text{min}}), l_{2,\text{max}}) \\
  l_{d3} &= \min(\max(r_{ij} \cdot d_3, l_{3,\text{min}}), l_{3,\text{max}}) \\
  r_c &= r_j + l_{d1}d_1 + l_{d2}d_2 + l_{d2}d_3 - r_i.
\end{align*}
\]

5.1.5 Sphere-Hypercuboid

This can be extended into n-dimensions by

\[
\begin{align*}
  r_{ij} &= r_j - r_i \\
  l_{dk} &= \min(\max(r_{ij} \cdot d_k, l_{k,\text{min}}), l_{k,\text{max}}) \\
  r_c &= r_j + \sum_{k=1}^{n} l_{dk}d_k - r_i.
\end{align*}
\]

Here, it is noted that for the line, rectangle and cuboid \( n=1,2,3 \), respectively. For the point case \( n = 0 \), the director part is ignored and \( r_c = r_j - r_i \) is recovered. Also the lower dimension can also be recovered in higher dimension by \( l_{k,\text{min}} = l_{k,\text{max}} = 0 \). By setting \( l_{k,\text{min}} = l_{k,\text{max}} = 0 \), the dimension associated with the \( k \) collapse away.

5.2 Constitutive Models

Using the \( \delta \) and \( n_c \) from the procedures above the constitutive models for how the discrete elements will interact can to be defined. The constitutive behavior along with the element description in Chapter 4 determine the trajectory. Below are a description of common constitutive models and their generalizations into power laws. The parameters for these models can be selected by the user or as the result of an inverse problem. In the inverse problem, the parameters of the constitutive model are considered to be unknown but are solved for by comparing the predictions of the simulations with known solutions from other simulations or experiments. In this way, insight into the underlying system can be found. Also, once the parameters are found new simulations can be performed to make predictions about the overall system under different conditions.
5.2.1 Conservative Force Models

A popular choice for contact for meso-scale systems is Hertz’s contact model \[55\]

\[
F_{ij,\text{hertz}} = \begin{cases} 
\frac{4}{3} E^* \sqrt{R^*} \delta \frac{3}{2} n_c & \delta > 0 \\
0 & \delta \leq 0
\end{cases} \tag{5.3}
\]

where \(E^* = \frac{E_i E_j}{E_i (1-\nu_i^2) + E_j (1-\nu_j^2)}\) and \(R^* = \frac{R_i R_j}{R_i + R_j}\) are effective properties for contact between spheres. For half-plane sphere contact \(R^* = R_{\text{sphere}}\). In this model \(E_i, E_j \in \mathbb{R}\) are the Young’s modulus, and \(\nu_i, \nu_j \in \mathbb{R}\) are the Poisson ratio’s. It is noted that newton’s 3rd Law, \(F_{ji} = -F_{ij}\) is satisfied by the change in sign of \(n_c\).

Another possible model is a linear spring model. While Hertz’s contact model is an analytical expression which accounts for deformation of the body, the linear spring model is an approximate model which works well as a penalty term to enforce a non-overlap condition. The expression for the linear spring model is

\[
F_{ij,\text{LinSpring}} = \begin{cases} 
K \delta n_c & \delta > 0 \\
0 & \delta \leq 0
\end{cases} \tag{5.4}
\]

where \(K \in \mathbb{R}\) is a spring constant and model parameter. A power law model can be written to generalize these two concepts

\[
F_{ij,\text{PowerLaw}} = \begin{cases} 
K_{\text{eff}} \delta \alpha n_c & \delta > 0 \\
0 & \delta \leq 0
\end{cases} \tag{5.5}
\]

noting that the units of \(K_{\text{eff}}\) are \(\frac{N}{m^\alpha}\). The previous two models can be written in this form with Hertz contact being

\[
K_{\text{eff}} = \frac{4}{3} E^* \sqrt{R^*} \\
\alpha = \frac{3}{2}
\tag{5.6}
\]

and Linear spring model being

\[
K_{\text{eff}} = K \\
\alpha = 1
\]

Models for bonding behaviors between elements are also need for when material solidifies next to each other. This can also be achieved by power law model but applied when \(\delta < 0\) of the form

\[
F_{ij,\text{BondPowerLaw}} = \begin{cases} 
-K_{\text{eff}} |\delta|^{-\alpha} n_c & \delta < 0 \\
0 & \delta \geq 0
\end{cases}
\]
The absolute value and sign must be used as now $\delta$ is no longer positive. A simpler way to automatically account for the sign is to use the form

$$F_{ij,BondPowerLaw} = \begin{cases} K_{\text{eff}} |\delta|^{\alpha-1} n_c & \delta < 0 \\ 0 & \delta \geq 0 \end{cases}. \quad (5.7)$$

If a symmetric contact and bonding model is desired it can be written concisely using the form in equation 5.7 resulting in

$$F_{ij,\text{SymPowerLaw}} = K_{\text{eff}} |\delta|^{\alpha-1} n_c \quad (5.8)$$

which can be used to model both compressive and tensile forces associated with a bonded material.

Another common physical phenomena that is interesting to add to the model is that of near field force. This model can be added along with a contact model to represent cohesion or adhesion between elements when they are close together. This near-field force is usually limited in extent in terms of how far away it acts on an element unlike the previous bonding models. A generic near-field attractive interaction can be defined using a cubic polynomial to build the model for a near field force as a function of overlap measures $\delta$. Assuming the near-field interaction has no force and derivative at the point of contact $\delta = 0$ and at the cutoff distance $\delta = \delta_{NF}$ and is symmetric about $\delta_{NF}/2$ with $F_{NF,max}$ occurring at the midpoint $\delta_{NF}/2$, the function is completely defined. Using the results in appendix C.1.2, the cubic in this situation has coefficients $a = -2F_{NF,max}(\delta_{NF}/2)$, $b = 3F_{NF,max}(\delta_{NF}/2)^2$, the resulting near field model constructed based as

$$F(r)_{NF,cubic} = \begin{cases} -\left[-2F_{NF,max}(\delta_{NF}/2)^3 + 3F_{NF,max}(\delta_{NF}/2)^2\delta\right]n_c & \delta_{NF}/2 < \delta < 0 \\ -\left[-2F_{NF,max}(\delta_{NF}/2)^3 (\delta_{NF} - \delta)^3 + 3F_{NF,max}(\delta_{NF}/2)^2 (\delta_{NF} - \delta)^2\right]n_c & \delta_{NF} < \delta < \delta_{NF}/2 \end{cases}. \quad (5.9)$$

A linear near field model could also be written down of the form

$$F(\delta)_{NF,linear} = \begin{cases} \frac{F_{NF,max}}{\delta_{NF}} \delta n_c & \delta_{NF}/2 < \delta < 0 \\ -\frac{F_{NF,max}}{\delta_{NF}} (\delta - \delta_{NF}) n_c & \delta_{NF} < \delta < \delta_{NF}/2 \end{cases}. \quad (5.10)$$

which would be cheaper to calculate but is very unsmooth and may lead to the need for smaller time steps and loss of order of accuracy due to the jumps in the derivative. Note: As reminder $\delta_{NF}$ is a negative value.
Another popular model for element-element interaction is the Lenard-Jones potential[53] for long range interaction. The interaction potential is usually written as

\[ V_{i,j} = \epsilon \left[ \left( \frac{r_m}{r} \right)^{12} - 2 \left( \frac{r_m}{r} \right)^{6} \right] \]

where \( r_m \) is the equilibrium distance between the two elements, \( r_m = R_i + R_j \), and \( r \) is the center to center length in our notation, \( r = \| \mathbf{r}_e \| = R_i + R_j - \delta \). Also, \( \epsilon \) is a parameter which determines the depth of the potential and thus the strength of the interaction force. Substituting this in gives

\[ V_{ij, LJ} = \epsilon \left[ \left( \frac{R_i + R_j}{R_i + R_j - \delta} \right)^{12} - 2 \left( \frac{R_i + R_j}{R_i + R_j - \delta} \right)^{6} \right]. \]

Taking \( F_{ij, LJ} = -\frac{dV_{ij, LJ}}{dr} \) gives the magnitude of the force in the radial direction

\[ F_{ij, LJ} = \epsilon \left[ 12 \left( r_m \right)^{12} \left( \frac{1}{r} \right)^{13} - 2 \left( r_m \right)^{6} \left( \frac{1}{r} \right)^{7} \right] = \frac{12\epsilon}{r} \left[ \left( \frac{r_m}{r} \right)^{12} - \left( \frac{r_m}{r} \right)^{6} \right]. \]

Again substituting to get in terms of \( \delta \) and the Radii

\[ F_{ij, LJ} = \frac{12\epsilon}{R_i + R_j - \delta} \left[ \left( \frac{R_i + R_j}{R_i + R_j - \delta} \right)^{12} - \left( \frac{R_i + R_j}{R_i + R_j - \delta} \right)^{6} \right]. \]

This force will be applied in the \( \mathbf{n}_c \) direction giving

\[ F_{ij, LJ} = \frac{12\epsilon}{R_i + R_j - \delta} \left[ \left( \frac{R_i + R_j}{R_i + R_j - \delta} \right)^{12} - \left( \frac{R_i + R_j}{R_i + R_j - \delta} \right)^{6} \right] \mathbf{n}_c. \] (5.11)

The Lennard-Jones potential has infinite extent making it computationally intensive to evaluate for particles in an extended system. However, the potential quickly approaches zero with radial distance so the lost potential is negligible and thus is neglected. To do this, a cutoff distance is set for when the force stops being applied is chosen. Traditionally, this cutoff is taken to be at \( r_{cut} = 2.5 \times 2^{-\frac{1}{6}} r_m \). This truncation produces a discontinuity in the force which is unphysical and numerically intensive to treat.

To correct this, a common method of smoothing is to establish a new cut off at \( r_{cut} = 2 \times 2^{-\frac{1}{6}} r_m \) and use a cubic interpolating polynomial which matches the force and the derivative of the force at \( r_{cut} \) and use the original distance as the location of zero force, \( r_0 = 2.5, \sigma = 2.5 \times 2^{-\frac{1}{6}} r_m \). Explicitly for the interpolating polynomial \( P \) that connects \( r_{cut} \) to \( r_0 \), the boundary conditions are

\[ P(r_0) = 0 \] (5.12)
\[ P(r_{cut}) = F_{ij, LJ}(r_{cut}) \] (5.13)
\[ P'(r_0) = 0 \] (5.14)
\[ P'(r_{cut}) = F'_{ij, LJ}(r_{cut}). \] (5.15)
Figure 5.5: Velocity Components of two colliding rigid bodies demonstrated on two spheres.

Where

\[ F'_{ij,LJ}(r) = -\frac{\epsilon}{r^2} \left[ 156 \left( \frac{r_m}{r} \right)^{12} - 84 \left( \frac{r_m}{r} \right)^6 \right]. \]

A third order polynomial is chosen to satisfy these conditions because it has enough degrees of freedom to satisfy the end point requirements and is unique. Since one point has both a zero value and derivative it fulfills the requirements for the special case in Appendix C.1.2. The result is that the force magnitude

\[
F(r)_{ij,LJ\text{trunc}} = \begin{cases} 
F'_{ij,LJ}(r) & r < r_{cut} \\
3a(r - r_0)^3 + b(r - r_0)^2 & r_{cut} < r < r_0 \\
0 & r > r_0
\end{cases} 
\]

\[ a = \frac{F'_{ij,LJ}(r_{cut})(r_{cut} - r_0) - 2F'_{ij,LJ}(r_{cut})}{(r_{cut} - r_0)^3} \]

\[ b = \frac{3F'_{ij,LJ}(r_{cut}) - F'_{ij,LJ}(r_{cut})(r_{cut} - r_0)}{(r_{cut} - r_0)^2} \]

still in the \( n_c \) direction.

### 5.2.2 Dissipative Force Models

For a rigid body, velocity at the point of contact, denoted by \( v_c \), is given by

\[
v_{c,i} = v_i + \omega_i \times r_{i,c}
\]

where \( v_i \) is the center mass velocity and \( r_{i,c} \) is the vector from the center of rotation to the point of contact. In discrete elements because of the inherent permissiveness of overlap, the point is a bit ambiguous. One possible choice is to pick the point of contact to occur
along the line connecting the contact centers and on the plane defined by the intersection of the bodies. For the sphere in contact, velocity at the contact is given by

\[ \mathbf{v}_{c,i} = \mathbf{v}_i + \omega_i \times (\mathbf{r}_i - l_i \mathbf{n}_c) \]

and for the other body denoted by subscript \( j \),

\[ \mathbf{v}_{c,j} = \mathbf{v}_j + \omega_j \times (\mathbf{r}_c + l_j \mathbf{n}_c) \]

where \( l_i \) and \( l_j \) are the distance to the contact point as defined in Appendix A. If the body has rotational degrees of freedom \( \mathbf{r}_i - l_i \mathbf{n}_c \) and \( \mathbf{r}_c + l_j \mathbf{n}_c \) are used in the torque calculation for \( i \) and \( j \) respectively. If the body is assumed to have no rotational degrees of freedom as is the case here, then \( \omega_i = 0 \) then \( \mathbf{v}_{c,i} = \mathbf{v}_i \) and \( \mathbf{v}_{c,j} = \mathbf{v}_j \) and the point of contact is not needed for the velocity calculation or the calculation of torque. The relative velocities of the two objects at the contact point as shown in Figure 5.5 are used to define viscous drag forces. It is common [14, 75, 50, 33, 112] to add a viscous energy loss terms parallel to the radial direction and for motion perpendicular to the radial direction to establish a dash-pot constitutive model for contact.

First, a linear drag in the direction parallel body centers can be expressed in the form

\[
\mathbf{F}_{ij,\text{Drag}} = \begin{cases} 
\gamma_i \frac{d\delta}{dt} \mathbf{n}_c & \delta > 0 \\
0 & \delta \leq 0
\end{cases}
\] (5.17)

which can be used to model the loss of energy from deformation during contact. Here, \( \gamma_i \) is a scaling factor with units \([N/(m/s)]\) where \( \frac{d\delta}{dt} \) is just the relative velocity in the normal direction

\[
\frac{d\delta}{dt} = -(\mathbf{v}_{c,i} - \mathbf{v}_{c,j}) \cdot \mathbf{n}_c
\] (5.18)

The non-trivial portion of the constitutive equation can written as

\[
\gamma_i \frac{d\delta}{dt} \mathbf{n}_c = \gamma_i \left[ -(\mathbf{v}_{c,i} - \mathbf{v}_{c,j}) \cdot \mathbf{n}_c \right] \mathbf{n}_c
\]

\[
= -\gamma_i \left[ (\mathbf{v}_{c,i} \cdot \mathbf{n}_c) \mathbf{n}_c - (\mathbf{v}_{c,j} \cdot \mathbf{n}_c) \mathbf{n}_c \right]
\]

\[
= -\gamma_i \left[ \mathbf{v}_{c,i} \mathbf{n}_c - \mathbf{v}_{c,j} \mathbf{n}_c \right]
\] (5.19)

plugging this back in gives the slightly more intuitive form

\[
\mathbf{F}_{ij,\text{Drag}} = \begin{cases} 
-\gamma_i \left[ \mathbf{v}_{c,i} \mathbf{n}_c - \mathbf{v}_{c,j} \mathbf{n}_c \right] & \delta > 0 \\
0 & \delta \leq 0
\end{cases}
\] (5.20)

where if the elements are in contact then this is a force opposite their relative velocities along the direction of contact. Equation 5.17 is a little easier to extend to the power law version of

\[
\mathbf{F}_{ij,\text{PowerDrag}} = \begin{cases} 
\gamma_i |\mathbf{v}_{c,i} - \mathbf{v}_{c,j}|^{\alpha} \mathbf{n}_c & \delta > 0 \\
0 & \delta \leq 0
\end{cases}
\] (5.21)
where the absolute value is necessary because $\frac{d\delta}{dt}$ is not guaranteed to be positive and the sign of $\frac{d\delta}{dt}$ must be recovered so that the force is opposite the motion. Another form that I prefer to use is the one shown below

$$ F_{ij,\text{PowerDrag}} = \begin{cases} \gamma \frac{d\delta}{dt} \left| \frac{d\delta}{dt} \right|^{-1} n_c & \delta > 0 \\ 0 & \delta \leq 0 \end{cases} $$ (5.22)

because it is easier to code and does have the ambiguity of the sign function at $\frac{d\delta}{dt} = 0$.

To model the friction between the discrete elements, a viscous drag model is also used with respect to the tangential velocities, perpendicular to the normal, of the particles

$$ F_{ij,\text{Drag}} = \begin{cases} -\gamma \perpendicular \left\{ (v_{c,i} - (v_{c,i} \cdot n_c) n_c) - (v_{c,j} - (v_{c,j} \cdot n_c) n_c) \right\} & \delta > 0 \\ 0 & \delta \leq 0 \end{cases} $$ (5.23)

$$ v_{c,i,\perp} = v_{c,i} - (v_{c,i} \cdot n_c) n_c $$

$$ v_{c,j,\perp} = v_{c,j} - (v_{c,j} \cdot n_c) n_c $$

To extend this to a power law version, first rewrite equation 5.23 as

$$ F_{ij,\text{Drag}} = \begin{cases} -\gamma \perpendicular \left\{ (v_{c,i,\perp} - v_{c,j,\perp}) (I - n_c \otimes n_c) \right\} & \delta > 0 \\ 0 & \delta \leq 0 \end{cases} $$

$$ v_{c,i,\perp} = v_{c,i} - (v_{c,i} \cdot n_c) n_c $$

$$ v_{c,j,\perp} = v_{c,j} - (v_{c,j} \cdot n_c) n_c $$

Then to make a power law model

$$ F_{ij,\text{PowerDrag}} = \begin{cases} -\gamma \perpendicular \left( \left\| v_{c,i,\perp} - v_{c,j,\perp} \right\| \right)^{\alpha - 1} \frac{v_{c,i,\perp} - v_{c,j,\perp}}{\left\| v_{c,i,\perp} - v_{c,j,\perp} \right\|} & \delta > 0 \\ 0 & \delta \leq 0 \end{cases} $$ (5.24)

The sign function is no longer needed because the direction of motion is preserved in the difference of the velocity vectors. When the velocities are comparable the division by $\left\| v_{c,i,\perp} - v_{c,j,\perp} \right\|$ can be near zero and is ill conditioned in floating point precision. This can be avoided by simplifying to get the preferred form

$$ F_{ij,\text{PowerDrag}} = \begin{cases} -\gamma \perpendicular \left( \left\| v_{c,i,\perp} - v_{c,j,\perp} \right\| \right)^{\alpha - 1} \left[ v_{c,i,\perp} - v_{c,j,\perp} \right] & \delta > 0 \\ 0 & \delta \leq 0 \end{cases} $$ (5.24)

which eliminates this problem and will defined when $v_{c,i,\perp} = v_{c,j,\perp}$. Also, note that equation 5.22 can also be rewritten using equation 5.19 in this form as

$$ F_{ij,\text{PowerDrag}} = \begin{cases} -\gamma \left\| v_{c,i} - v_{c,j} \right\|^\alpha \left[ v_{c,i} - v_{c,j} \right] & \delta > 0 \\ 0 & \delta \leq 0 \end{cases} $$ (5.25)
The perpendicular drag (friction model) and the parallel drag do not depend on the sign of $\delta$ and so if one wants to apply them to a bond between material no modification is need to the functional form. All that is required is to apply the conditional so that it matches when the bond is active. For the version without the conditional the additional annotation $Sym$ will be used to indicate the model is symmetric about $\delta = 0$.

A final loss mechanism utilized in the simulation is a drag model with a background fluid. This model introduces a drag proportional to difference in velocity between the element and the background fluid field. Using the same form for the other power law drag models gives

$$F_{i,DualPower} = -\gamma_D \|v_i - v_{bf}\|^{\alpha_D-1} [v_i - v_{bf}]$$

(5.26)

where $v_{bf}$ is a background fluid velocity at location of the center of the element, $r_i$.

5.2.2.1 Objectivity

One should be careful and remember that velocity and relative velocity are not objective quantities. Using $+$ to denote the superposed rigid body frame with rotational position $Q(t)$ and translation $c(t)$ then

$$x^+ = Q(t)x + c(t)$$
$$v^+ = \dot{Q}(t)x + Q(t)v + \dot{c}(t)$$

Neither of which are objective. The difference is positions

$$x_1^+ - x_2^+ = Q(t)x_1 + c(t) - Q(t)x_2 - c(t)$$
$$= Q(t) (x_1 - x_2)$$

is objective so force laws such as Hertzian contact or the power law, equations 5.3 and 5.5 respectively, which use the difference in position are objective.

However, for relative velocity

$$v_1^+ = \dot{Q}(t)x_1 + Q(t)v_1 + \dot{c}(t)$$
$$v_2^+ = \dot{Q}(t)x_2 + Q(t)v_2 + \dot{c}(t)$$
$$v_1^+ - v_2^+ = \dot{Q}(t) (x_1 - x_2) + Q(t) (v_1 - v_2),$$

the term $\dot{Q}(t) (x_1 - x_2)$ makes the relative velocity not objective. However, in discrete elements the velocities used to calculate the drag forces both exist at the same spatial location so $x_1 = x_2$ the result being that $\dot{Q}(t) (x_1 - x_2) = 0$ if $\dot{Q}(t)$ is finite and thus

$$v_1^+ - v_2^+ = Q(t) (v_1 - v_2)$$

but only if the velocities are occurring at the exact same spot in space.
5.2.3 Thermal Constitutive Models

Two models for heat exchange are used in this dissertation. First is a heat transfer model between two impinging entities. The model is the simplest possible in that it depends linearly on the temperature difference between the two bodies. When two bodies are in contact, the rate of heat transfer between the bodies is modeled as

\[ Q_{ij,\text{HeatTransfer}} = \begin{cases} K_{ij} (\Theta_i - \Theta_j) & \delta > 0 \\ 0 & \delta \leq 0 \end{cases} \tag{5.27} \]

where \( K_{ij} \) is a model parameter that accounts for the cumulative heat transfer effects such as surface conduction and convective heat transfer between elements in close proximity. Here again the model does not depend on \( \delta \) and can used as a model for heat transfer with bond if the condition is removed. The version without the conditional will be denoted by the subscript \( \text{Sym} \).

The other model is a laser-like heat source which has a Gaussian distribution in profile and reduces in intensity exponentially into the body. Let \( r_s \) be the position of the source, \( d \) be the direction the source is pointed, \( r_i \) be the position of the element. Then

\[
\begin{align*}
    r_{is} &= r_i - r_s \\
    l_\parallel &= r_{is} \cdot d \\
    l_\perp &= \| r_{is} - l_\parallel \cdot d \|
\end{align*}
\]

Where \( l_\parallel \) is the distance way from the source parallel to its axis and \( l_\perp \) is the distance way from the main axis of the source. The intensity profile is given by

\[
I = I_o \exp \left( \frac{-l_\perp^2}{2\sigma^2} \right) \exp \left( \lambda l_\parallel \right)
\]

where \( \sigma \) is the standard deviation of the beam, \( I_o \) is the nominal intensity at \( r_s \), and \( \lambda \) is the parameter that controls the rate of drop off of the heat source into the body of material. It is assumed that the power entering a given element is approximately the apparent area of the particle times the intensity at the center of the element, \( Q_i = I(r_i)\pi R_i^2 \). The final functional form for the heat source is

\[
Q_{i,\text{source}} = \begin{cases} I_o \pi R_i^2 \exp \left( \frac{-l_\perp^2}{2\sigma^2} \right) \exp \left( \lambda l_\parallel \right) & l_\parallel > 0 \\ 0 & l_\parallel \leq 0 \end{cases} \tag{5.28}
\]
Chapter 6

DEM Model Characteristics

6.1 DEM Convergence Study

6.1.1 Model Details

A 2-D simulation is set up with 10 circular discrete elements which move according to rigid body dynamics without rotational degrees of freedom. Each element moves according to reduced order dynamics ODEs

\[
\frac{dx_i}{dt} = v_i \\
\frac{dv_i}{dt} = \frac{1}{M_i} F_{i,tot}
\]

where the subscript \( i \) indicates a particular discrete element. The elements contained in a 1x1 box from \([0,1] \times [0,1]\) made from 4 line segments as per section 5.1.2 with a 5th line segment across its center at \( y = .5 \) extending from \( x = .25 \) to \( x = .75 \). A power law contact force is applied between elements via a simple \( N^2 \) mesh. Likewise, a power law contact model is used for contacts with \( 5 \times N \) list of elements and walls combinations. The discrete elements are also subject to gravity in the negative y direction. Thus

\[
F_{i,tot} = \sum_{j=\text{elements} \neq i} F_{ij,PowerLaw,ss} + \sum_{j=\text{Lines}} F_{ij,PowerLaw,sl} + M_i \begin{pmatrix} 0 \\ -g \end{pmatrix}
\]

where \( F_{ij,PowerLaw} \) is as described in equation 5.5 where \( n_c \) and \( \delta \) are calculated with the sphere-sphere algorithm and sphere-line contact enumerated in section 5.1 and denoted by the subscripts ss and sl, respectively. For the power law contact forces \( K_{eff} = \left( \frac{1}{E_i} + \frac{1}{E_j} \right)^{-1} \) for both circle-circle and circle-line contacts. All \( E_i, E_j = 1 \times 10^5 \) regardless of circular element or line segment so \( K_{eff} = 5 \times 10^4 \). \( M_i = 1 \), \( rad = 0.05 \) for all circular elements and \( g = 9.8 \). The Particle-Particle power law contact models is expressed symbolically to make a SymExpression as in Algorithm 6.1. The example shows how easy it
Algorithm 6.1 Power Law Contact force with Sphere-Sphere Contact expressed in the
DSL.

```python
#express contact symbolically with sympy
r1=sympy.Matrix([[x1, y1]])
r2=sympy.Matrix([[x2, y2]])
rc=r2-r1
lengthrc=rc.norm()
nc=rc/lengthrc
delta=rad1+rad2-lengthr12
E=1/((1/E1)+(1/E2))
f2=sympy.Piecewise((E*(delta**alpha),(delta > 0.0)),(0.0,True))*nc
f1=-f2
f=sympy.Matrix([[f1/M1, f2/M2]])

#store expression in the danplusplus SymExpression Object
variables=sympy.Matrix([[r1], [r2]])
parameters=sympy.Matrix([[rad1], [E1], [M1], [rad2], [E2], [M2], [alpha]])
Hertzian=danplusplus.SymDiff.SymExpression(f, variables, parameters)
```

is to setup models as the description of the physics and math are performed at a very
high level by creating vectors and performing standard vector operations.

Because the DSL automatically generates the necessary gradients, fully implicit DEM
simulations are possible. Figure 6.1 shows a few frames of the system with $\alpha = 1$ evolving
using the 2-stage fully implicit Gauss Legendre RK method. Figure 6.2 shows the number
of Newton iterations needed for convergence for a series of timesteps. The system is non-
linear when two elements are resolving a contact because the contact force function has
a jump in the derivative from 0 when not in contact to a stiffness of $K_{eff}/M_i$ when they
are. If during a time step no contacts needs to be resolved, only two newton iterations
are needed because in this configuration the system is linear with respect to the variables.
In this case, one iteration is needed to solve the linear system and the second iteration
occurs to check that convergence has occurred.

### 6.1.2 Convergence Study

In order to better understand the consequences of the power law contact model, con-
vergence studies where run over a set of explicit and implicit Runge-Kutta schemes of
which the Butcher tableueaus can be found in appendix B. The system is run to final time
$T_f = 0.02$ for powers $\alpha = 0, 1, 1.5, 2.5$, matched for both the line and element contacts.
For the implicit methods, Newton’s method is used to find the solution and the system of
equations in each iteration is solved using scipy.sparse.linalg.spsolve function in Python.
Figure 6.1: A few points in time of the evolution of the system.

Figure 6.2: Newton Iterations per time step for the 2-stage Fully implicit Gauss Legendre RK method (4th order).
The convergence rate for Newton's method is $tol = 1 \times 10^{-24}$ with the energy tolerance. A “true” solution for comparison is created by running RK4 with $h = 1 \times 10^{-8}$. The error is measured using the max norm over the position degrees of freedom. Plots of the convergence behavior are shown in Figure 6.3 and measure order of accuracy are given in Table 6.1.

What is observed from the convergence study is that the maximum order of accuracy depends on $\alpha$. Particularly it seems to depend on the continuity of the constitutive model, the power law in this case, and its derivatives. The $n^{th}$ derivative with respect to $\delta$ of the power law model is

$$
\frac{d^n F_{ij,\text{PowerLaw}}}{d\delta^n} = \begin{cases}
\alpha(\alpha-1)\ldots(\alpha-n+1)K_{\text{eff}}\delta^{\alpha-n}\mathbf{n}_c & \delta > 0 \\
0 & \delta \leq 0
\end{cases}
$$

The model is $C^\infty$ for each sections of the piecewise functions but continuity can be lost at $\delta = 0$. Continuity is maintained up until $n \geq \alpha$ at this point the $\alpha - n$ is either 0 or negative. If $\alpha - n = 0$ then

$$
\frac{d^n F_{ij,\text{PowerLaw}}}{d\delta^n} = \begin{cases}
\alpha(\alpha-1)\ldots(\alpha-n+1)K_{\text{eff}}\mathbf{n}_c & \delta > 0 \\
0 & \delta \leq 0
\end{cases}
$$

and at $\delta = 0$ there is a jump in the function. If $\alpha - n < 0$ then at $\delta = 0$ there is a zero in the denominator so the function does not match the 0 which is the limit from the other direction. The result is that $\alpha = 0$ is not continuous with order of accuracy limit of $O(h^1)$, $\alpha = 1$ is $C^0$ with limit of $O(h^2)$, $\alpha = 1.5$ is $C^1$ with limit of $O(h^3)$ and $\alpha = 2.5$ is $C^2$ with limit of $O(h^4)$. Thus, $N_{\text{accuracy}} = N_{\text{continuity}} + 2$ or $C^{N_{\text{continuity}}}$ has a limiting order of accuracy of $O(h^{N_{\text{accuracy}}})$ with the special case if the function is not continuous the $N_{\text{accuracy}} = 1$. This result is true for both explicit and fully implicit methods.

This observation is of particular importance to discrete elements because often the physics of the system of interest determines which models are appropriate. For example,
Figure 6.3: Plots of the convergence behavior for different RK methods to integrate the DEM dynamics problem. For the $\alpha = 0$ case the implicit methods are not included because the Newton Iterations did not converge.
$\alpha = 1.5$ is often used because to match the Hertzian contact model for the collision of elastic spheres. If this model must be used, the order of accuracy is limited to 3rd order so one would likely want to chose a method at most 3rd order accurate to not waste computation. Likewise it is very common to use $\alpha = 1$ because it is simple and can be viewed as the first order approximation to any model. Combined with discontinuity introduced by contact this will limit the order of accuracy to 2nd order. Higher order methods seems to still have lower truncation error as seen in plots in Figure 6.3 where the lines have a same limited slope but the lines are lower indicating an overall lower error. The stability characteristic are not investigated here but the higher order methods may also still have superior stability properties.

A couple solutions are proposed. First is to use models which are smooth enough for the desired order of integration. Also regularizing by adding a third piecewise element to the function to smooth out the model’s function at the $\delta = 0$ discontinuity may also be a good solution in some situation. A third possible solution is to take time steps of the right size that they hit on the times when contact occurs. This way the numerical integration exists on either side of the discontinuity avoiding the problem, this is the same approach that is taken in finite elements when a domain is meshed with edges the coincide with the discontinuous material boundary so the material model is smooth over an element. Generally for a simulation of particle dynamics finding the time of contact (aka discontinuity) is not practical as the contact can happen time and in no predictable fashion. Adaptive time stepping may be able to improve this by resolving the contact discontinuity better but further work is needed to determine the exact implications.

### 6.2 Unphysical Bonding Models

In additive manufacturing, the creation of powders for use in the deposition processes is a topic of interest. Powders are produced via range of methods including milled, electrolytic, atomization and precipitation which in general produce a wide range of sizes and shapes based on the material, process and production parameters[19, 92, 107]. The morphology of the particles have the potential to effect the additive manufacturing process at all steps but one place where the morphology has a large impact is in sieving powders after production. Sieves are often used to segregate particles of different sizes and distributions which can improve packing and ultimately the final part density. The mesh used to separate particles in fine powders, however, are prone to clogging[64, 68].

To investigate clogging during sieving with non-spherical geometries, a discrete element model is formulated. A mesh is created using a set of lines with radius as described in section 5.1.2 at 90 degree angles to each other and rectangles to build a hopper to drop powder from and a basin to catch powder after it passed through the mesh. Finally, non-spherical bodies where constructed by bonding spherical elements together to approximately match non-spherical geometries. Figure 6.4 and 6.5 shows the sieving numerical experiment for with spherical and non-spherical entities.
Figure 6.4: Picture of sifting of Spherical particles

Figure 6.5: Left: Non-spherical powder particles constructed using spherical elements and penalty constraints. Right: Particle bound up on sifting mesh created by the lines.
In the process of investigating the sieving experiment, it was noticed that while the spherical elements were behaving as expected, the non-spherical bodies were not rotating as expect, either not rotating at all or sluggishly. For example on the right in Figure 6.5 a number of elements sitting vertically in the basin a position that should be relatively improbable. In extreme cases, bodies would hits edge on a line and instead of rotating around the line it would bounce off as if the force is being applied at the center of mass.

It was discovered that the reason for this is the inclusion of the perpendicular drag term in the bonds that connected the bodies. Specifically, the force sum on each element was as below

\[ F_{i,tot} = \sum_{j=\text{elements}\backslash \text{body}} F_{ij,PowerContact,ss} + F_{ij,PowerDrag\perp,ss} + F_{ij,PowerDrag\parallel,ss} + \sum_{j=\text{body}} F_{ij,\text{SymPowerContact}} + F_{ij,\text{SymPowerDrag\perp}} + F_{ij,\text{SymPowerDrag\parallel}} + \sum_{j=\text{lines}} F_{ij,\text{PowerContact,ls}} + F_{ij,\text{PowerDrag\perp,ls}} + F_{ij,\text{PowerDrag\parallel,ls}} + \sum_{j=\text{rectangles}} F_{ij,\text{PowerContact,rs}} + F_{ij,\text{PowerDrag\perp,rs}} + F_{ij,\text{PowerDrag\parallel,rs}}. \]

The offending term that cause the body to move only with translational degrees of freedom is \( F_{ij,\text{SymPowerDrag\perp}} \). Excluding this term results in the recovery of the rotation that is expected with the body. This is demonstrated in Figure 6.6 where the series on the left show lack of rotations as a result of the term and the series on the right shows the corrected behavior with the drag term removed. Figure 6.7 is a plot of the angle of the simulated block around the y axis in the xz plane at a given time as a function of the drag parameter \( \gamma_{\perp} \). This figure shows the dependence of the rotational locking on the \( \gamma_{\perp} \) and the intensity in which it acts to suppress rotation of the body.
Figure 6.6: Left: With the perpendicular drag term on resulting in no rotation. Right: Perpendicular drag term off restoring the expected rotational behavior.
Figure 6.7: Figures shows the dependence of the angle of the block in Figure 6.6 at given time as a function of the parameter $\gamma_\perp$ demonstrating the locking behavior and the strength at which it acts to prevent rotation.
Chapter 7

Models for Additive Manufacturing

7.1 Physically based Powder Packing for SLS/SLM

In an SLS or SLM machine, represented in Figure 7.1, a part is built up layer by layer by using a laser to melt or sinter metal powder particles together. Initially, all the metal power is in the reservoir basin and none in the work area. To deposit a thin layer in the work area, first a piston under the reservoir will rise a small amount exposing a layer of powder and the work area piston moves down (30-60µm) to receive the powder. Next, a wiper blade moves across the machine and moves the exposed powder in the reservoir over to the work area, spreading a uniform layer and filling the void that was created by the work area piston moving down. To ensure complete filling, more powder is used than expected and excess is pushed off the edge and into a collection basin. A laser then sweeps across the new layer of powder fusing it to adjacent powder particles and the layer underneath it to form a layer of the part. The process repeats to build up the part layer by layer.

The state of the powder when the laser begins to sinter/melt a new layer of the part together is important for understanding the final state of the part at completion. Before a simulation of the laser melting and sintering process can be performed an initial packing of powders must be obtained. Traditionally in the literature, this initial packing is based on various algorithms including random particle insertion with drop packing [59, 40, 60] where non-overlapping particles are seeded in the simulation and allowed to fall and settle into the work area. However, these method of packing do not match the procedures of wiping used in actual SLS or SLM machines currently on the market as described above.

The wiping procedure seen in operational machines has the potential to change the properties of the powder. Of particular interest are the solid fraction in the work area and the particle radius distribution. These two are thought to be related to final part density and smaller particles require less heat to melt then larger particles which could effect the desired laser intensity and speed. To better understand the effect of the wiping process, a discrete element simulation is used which better matches the actual machine by fully modeling the rising and falling action of the reservoir and the wiping of the powder into
the work area basin. The powder in the work area will be used to study effects on packing efficiency and particle distribution.

7.1.1 Model Description

For this simulation, each powder particle is modeled by a single spherical discrete element. The powder in the simulation starts with an initial diameter distribution which is Gaussian in nature centered at $D_{\text{mean}} = 27\mu m$ with full width half maximum (FWHM) of $FWHM = 1.17D_{\text{mean}} = 31.6\mu m$ which is a standard deviation of $\sigma = 13.4\mu m$. The powder used in these machines are often sifted so the there is a minimum and maximum diameter. Here, $D_{\text{min}} = 17\mu m$ is the minimum diameter in the distribution and $D_{\text{max}} = 42\mu m$ is maximum. To match this distribution in the simulation environment, diameters are selected based off the Gaussian distribution without the cutoff using the NumPy [99] package “random” and the routine “normal”(NumPy version 1.9.1). Elements with a diameter outside the cut off are assigned new diameters again using the “normal” routine. Elements with diameters outside the cutoff are again reselected. This process repeats until all diameters fall between the bounds.

The basins and wiper are created by using a rectangles described in subsection 5.1.3. All geometries are defined from a (0,0,0) point at the bottom of the work basin marked in Figure 7.2. There are 6 infinite walls and 7 finite rectangles which are used in the simulation. The 6 infinite walls form a box around the simulation and the rectangle form
Figure 7.2: Dimensions of the simulation with the (0,0,0) location marked at the bottom of the work basin.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>(0, 0, 0)</td>
<td>(leftmid, 0, midheight)</td>
<td>(leftmid, 0, 0)</td>
<td>(rightside, 0, 0)</td>
</tr>
<tr>
<td>d₁</td>
<td>(0, 1, 0)</td>
<td>(1, 0, 0)</td>
<td>(0, 1, 0)</td>
<td>(0, 1, 0)</td>
</tr>
<tr>
<td>d₂</td>
<td>(0, 0, 1)</td>
<td>(0, 1, 0)</td>
<td>(0, 0, 1)</td>
<td>(0, 0, 1)</td>
</tr>
</tbody>
</table>

Table 7.1: Positions and directors for the 4 static rectangles which make up the divider and right hand side of the work basin.

the bottom of the two basins, the 3 are used for the divider in the middle, one is the right hand side of the work basin and the last one is the wiper blade. The 4 static rectangles are located at the positions in table 7.1. The bottoms of the basin and the sweeper start at positions defined in table 7.2.

The first step in the simulation is drop packing the discrete elements into the reservoir. For this portion of the simulation, the 6 infinite walls are placed around the reservoir to contain the elements as they drop into place. The position and normal for the 6 infinite walls for the drop packing phase are provided in table 7.3.

Inside the cube created by the 5 walls (wall 3 in table 7.3 is below the basin bottom and is not active here) and the basin bottom, the initial positions of the particles are randomly seeded. No attempt is made to prevent overlap. At the start of the simulation, the particles in contact will repel each other due to the Hertzian contact interaction force and bounce around contained by the cube damped by viscous drag models at contact. An element is said to be in contact with a wall or rectangle according to subsection 5.1.3.
When a element is in contact with a infinite wall or rectangle is interacts again using the Hertz contact model with norm and drag terms detailed below. The elements fall under the force of gravity and come to rest on the bottom of the reservoir and pack the basin. The drop packing phase last for $t_{drop} = 0.06$ seconds.

Next, the wipe cycles occur which are repeated 3 times. For this, the infinite wall closest to the work reservoir is moved out of the way and used to create the basin in which to catch excess powder. The new positions of the infinite walls are shown in Table 7.4. First, at $t_{raiselower} = 0.02$ seconds, the reservoir basin raises at a rate of

$$v_{resv} = \left(0, 0, \frac{t_{raise}}{t_{raiselower}}\right)$$

starting from $z = 0$, and the work basin lowers at a rate of

$$v_{work} = \left(0, 0, -\frac{t_{lower}}{t_{raiselower}}\right)$$

starting from midheight. This exposes powder to be wiped over into the newly created space in the work area. Then, for the next $t_{wipe} = 0.1$ seconds,
the wiper blade moves with velocity $v_{\text{wipe}} = \left( \frac{t_{\text{sweep}}}{t_{\text{wipe}}}, 0, 0 \right)$. The result is the wiper blade rectangle moves across the domain pushing powder particles into the basin. At the end wipe, the wiper blade is returned to its initial position instantaneously. This is done a total of 3 times.

The spherical elements for the powder particle behave according to the dynamics formulation in Chapter 4 with only translational degrees of freedom. The elements interact with other particles and walls in the simulation via the constitutive models which contribute to the sum of forces below

\[
F_{i,\text{tot}} = \sum_{j=\text{elem} \neq i} F_{ij,\text{part-part}} + \sum_{j=\text{Rect}} F_{ij,\text{rect-part}} + \sum_{j=\text{infwalls}} F_{ij,\text{infwall-part}} + F_{i,\text{gravity}} + F_{i,\text{DragPower}}
\]

\[
F_{ij,\text{part-part}} = F_{ij,\text{PowerContact,ss}} + F_{ij,\text{PowerDrag\perp,ss}} + F_{ij,\text{PowerDrag\parallel,ss}}
\]

\[
F_{ij,\text{rect-part}} = F_{ij,\text{PowerContact,rs}} + F_{ij,\text{PowerDrag\perp,rs}} + F_{ij,\text{PowerDrag\parallel,rs}}
\]

\[
F_{i,\text{infwall-part}} = F_{ij,\text{PowerContact,rs}} + F_{ij,\text{PowerDrag\perp,rs}} + F_{ij,\text{PowerDrag\parallel,rs}}
\]

\[
F_{i,\text{gravity}} = M_i \begin{pmatrix} 0 \\ 0 \\ -g \end{pmatrix}
\]

where the subscript $ss$ indicates $\delta$ and $n_c$ are calculated using the Sphere-Sphere equation, $rs$ denotes the rectangle-sphere equation and $r_{\infty}s$ is the modification to the rectangle-sphere check for an infinite plane. The sum of the particle-particle forces include all elements that are not i. Doing this amounts to an $N^2$ mesh discussed in section 4.5.1 which causes the cost to calculate this term increases as $O(N^2)$ where $N$ is the number of particles in the simulation. A Verlet could be used to improve scaling but only a small number of elements ($N=400$) are used so it is not necessary in this case. Verlet lists for tracking the likely contact candidates are used in the LMD simulation in section 7.3.

The movement of the rectangles are also mediated with ODE’s, specifically the velocity ODE

\[
\frac{d\mathbf{r}_{\text{rect}}}{dt} = \mathbf{v}_{\text{rect}}
\]

where the velocity of the rectangle is parameter set as described above. The entire numerical experiment is written using the DSL with simulation flow performed as described in Chapter 3. For the actual computations, a unit system is chosen to avoid large differences in magnitude of the floating point numbers. The length unit used is $\text{LengthUnit} = 100\mu\text{m}$, the time unit is $\text{TimeUnit} = 0.01\text{seconds}$, and the mass unit is chosen so the an element of diameter $D_{\text{mean}}$ would have a magnitude of 1 which is found by $\text{MassUnit} = \frac{\rho 4\pi}{3} \left( \frac{D_{\text{mean}}}{2} \right)^3$. A full list of parameters can be found in appendix D.1.
7.1.2 Results

Figure 7.3 shows representative frames in a time series of the wipe packing simulation. A simulation is quite efficient taking only about 30 minutes on a single core on a personal laptop. The drop packing and subsequent wipe packing simulations were run over a range of particle-particle drag parameters for tangential drag (⊥ to center to center vector) and for each drag parameter, the simulation was run 12 times with different initial positions and radius, generated by the pseudo random number generator in NumPy from 12 different seeds. This is done to determine the sensitivity of the simulation to initial conditions.

The parameter study reveals some interesting trends in the solid fraction which is measured two ways, first by the total volume of spherical elements divided by the basin size and the second the same but with the volume of the element sticking over the top of the basin removed to reduce edge effects. The edge effects with the sides of the basin are not taken into account here. The results of the parameter study are seen in Figure 7.4. The observed behavior is that the packing density decreases approximately linearly with increasing drag until about 600 nN/m/s at which point the decrease remains linear by the slope appears to decrease (the solid fraction decreases less with increased drag parameter). The change is likely also dependent on blade speed and the drag parameter between the ball and the coater blade. The effect of these parameters will be studied in future work. It should also be noted that the initial random loose packing in the reservoirs is consistent with measurements of solid fraction of loose packed steel spheres in a cylinder of about 0.6 [82].

Second, the wiping leads to a change in the distribution of the particles. Figure 7.5 shows that this by dividing the percentage of particles above 30um after drop packing and after wipe packing; Likewise this is done for particles below 30um. From this it is seen that wiping reduces the number of particles with a diameter larger than the work basin drop depth of 30 um. This fits the intuition that the wiper blade would grab particles sticking out above the top and preferentially pull them out of the work area. Smaller particles move into fill the void and thus represent a larger portion of particles in the work area compared to in the reservoir (drop packed area). The effect seems to be mostly independent of the particle-particle tangential drag parameter. Likely, this effect is also dependent on blade speed and the tangential drag parameter between the particle, wiper blade and wiper blade rigidity which will be studied in future work.

7.2 Powder Ejection by Thermal Expansion

In speaking with scientists at LLNL, it was indicated that during the SLS process, powder particles sometimes pop out as the laser is passed over the bed to form the part. One hypothesis for the cause of this effect is that thermal expansion of the particle would be sufficient to cause powder to pop out of the basin. To test this, multiphysics discrete model was constructed to account for the thermal behavior and expansion of the powder as a heat source is run across the powder.
Figure 7.3: Left to Right and Top To Bottom: Particles are swept from the reservoir on the right and deposited into the work area by a soft coater blade. Three layers are deposited. Particles are colored by diameter.
Figure 7.4: The solid fraction in the basin for Drop Packing and Wipe Packing as a function of the particle-particle tangential drag parameter. Each ‘x’ represents a run seeded with a different initial positions and radius. Left: Simple analysis of total powder volume/basin volume. Right: Same but volume of powder sticking over top of basin subtracted.

Figure 7.5: A plot showing the relative number density of elements between drop packing and wipe packing. Compared are the results for elements above and below 30µm, the work area basin drop, for a range of $\gamma_{\perp}$. This shows that smaller elements preferentially occur in the work basin regardless of drag parameter.
The dynamics are formulated exactly as in the wiping simulation and implemented in the DSL. The wiping simulation is used to create an initial condition for the powder in the work area before heat source interacts with the discrete elements. The difference is now the radius of the elements is determined by an equation of state that is temperature dependent. The thermal expansion of the DEM element is modeled using a linear expansion state equation model

\[ R_i(T) = R_{T_0,i}(1 + \alpha_{\text{therm}}(\Theta_i - \Theta_0)) \]

where \( R_{T_0} \) is the Radius of the particle at \( T_0 \) and \( \alpha_{\text{therm}} \) is the thermal expansion coefficient. For 316L stainless steel \( \alpha_{\text{therm}} = 16 - 20 \times 10^{-6} \frac{1}{\text{K}} \) [31] depending on temperature.

The thermal behavior of the powder is described by the ODE in equation 4.19 reproduced below

\[ C_i M_i \frac{d\Theta_i}{dt} = Q_{i,\text{tot}}. \]

The heat in and out of the element is the sum

\[ Q_{i,\text{tot}} = \sum_{j=elem \neq i} Q_{ij,\text{HeatTransfer}} + Q_{i,\text{source}} \]

where \( Q_{ij,\text{contact}} \) is the rate of heat transfer between elements in contact according to equation 5.27 with \( K_{\text{cond}} = 1 \times 10^3 \text{SimUnits} \) (the rescaled units system used in the simulations). There is no heat transfer between the spherical elements and the walls. The heat source term, \( Q_{i,\text{source}} \) is modeled by a Gaussian distributed heat source described by equation 5.28, marked by the hexagon in Figure 7.6. The heat sources position, \( r_s \), is determined by the differential equation

Figure 7.6: The heat source representing the laser which is represented by the purple hexagon has passed over the powder bed heating the particles.
Figure 7.7: Left: With thermal expansion coefficient near that of Stainless Steel no particle movement is observed. Right: With a thermal expansion 10X that on the left images yields particles seen to pop up and out of the bed.

\[ \frac{dr_s}{dt} = v_s \]

where \( v_s \) is the parameter set by the user, in this case 2m/s. The heat sources passes over the bed, which is only 200um long, in 100\( \mu \)s so to resolve the time scale time steps of \( dt = 1 \times 10^{-9} \) seconds. Once the heat source has passed, the time step is returned to \( 1 \times 10^{-6} \) to evaluate the resulting dynamics which occurs on a much longer time scale.

A total power in the laser of \( P_{4\sigma} = 100 \) watts within the 4 sigma radial drop off mark of the laser, \( D_{4\sigma} = 54 \mu m \). Assuming that this is approximately the power out to infinity, \( P_\infty \approx P_{4\sigma} \) and using that the volume under a 2-D Gaussian, \( P_\infty = 2\pi \sigma^2 I_0 \), it is determined that so \( I_0 \approx \frac{P_{4\sigma}}{2\pi \sigma^2} \). The heat capacity is set to be \( C_i = 500 J/KgK \).

It is seen from this that for substantial motion to induced by the change in temperature, the coefficient must be about 10 times larger than the value of stainless steel as shown in Figure 7.7. The power of the laser and thus the temperature could be increased to achieve the same effect as increasing the thermal expansion coefficient but this arrangement is chosen to match the system and to keep the temperature of the elements around the melting point of stainless steel to best match the regime of interest. Thus it is concluded that the thermal expansion is unlikely to be the cause of this observed behavior.

### 7.3 Laser Metal Deposition

Laser metal deposition (LMD) is an additive process in which a part is built up by spraying powder at locations where material is needed in the next layer. The powder is melted by a laser in transit from the nozzle to the part where it adheres, cools, solidifies and bonds to the part. Like other additive manufacturing processes that use melted
material to build up parts layer by layer, LMD is susceptible to undesirable effects such as residuals stresses from the differential cooling and bad bonding from issues with wetting or incomplete melting[43, 34]. While annealing can sometimes be used to reduce residual stresses of a final part, the residual stress can lead to deformation of parts and even cracking during production that annealing can not correct. Having an computationally efficient manner to model the laser deposition process would allow designers to pick process parameters such as deposition rate and pattern to best address the problems. Here, a discrete element model is developed as a computationally efficient tool to study this process.

7.3.1 Model Formulation

Each powder particle in the system is modeled by a discrete element with a position, velocity and temperature as per Chapter 4. The element starts in a melted state and are given an initial position and velocity to match the desired deposition pattern. At the moment, the powder particles are assumed to be above their melting temperature and the laser heating step is not modeled. The powder is deposited onto a rectangular base which is kept at a fixed temperature. Heat can be transferred from the element to the base and can also bond to a position on the base if the element cools when in contact with the plate. Elements also interact with each other when in contact with each other by a force law and heat transfer. Finally, the particles can form permanent bonds if they cool when in contact with each other. The dynamics, thermal and bonding models are fully coupled and solved monolithically. The model is written with the DSL and simulation flow controlled as described in section 3.1.

The model for the dynamics and the thermal interactions are the same as in the previous sections using the same constitutive models and equation of state for the radius. This will be written down in detail later but for the moment lets focus what is new, the bonding model. To track the bonds between elements or the plate, a Verlet list is used which stores the index of the the two objects that are potentially bonded and a state variable which is a measure of the bond status. The bond itself is binary, turning on when the bond status is greater than one but the bond status is a continuous variable determined by an ODE allowing it to easily be added to the existing simulation framework based on the integration of ODEs. Specifically, the bond status is determined by the equation

\[
\frac{dBondStatus}{dt} = \begin{cases} 
0 & \Theta_i < \Theta_{\text{solid}} \& \Theta_j < \Theta_{\text{solid}} \\
K_{\text{bondstatus}} & \Theta_i > \Theta_{\text{solid}} \& \Theta_j < (\Theta_{\text{solid}} + \Theta_{\text{bond}}) \\
|\Theta_j > \Theta_{\text{solid}} \& \Theta_i < (\Theta_{\text{solid}} + \Theta_{\text{bond}}) 
\end{cases}
\]  

(7.1)

where \( K_{\text{bondstatus}} \) is a user set parameter which effects how easily bonds are formed. \( \Theta_i \) and \( \Theta_j \) are the temperatures of the two objects in question. \( \Theta_{\text{solid}} \) is the temperature at which the powder solidifies. Since there is no actually information in the model for temperature
gradient within a discrete element by assumption and there is no actual measure of when solidification at any one point occurs. Thus \( \Theta_{bond} \) is a fudge region that indicates when the temperatures are close enough to solidification that the element would in the process of solidifying and likely to form a bond with an adjacent element.

In previous models, \( N^2 \) application of the force models between element was sufficient, however, here a more sophisticated treatment is needed to track potential bonds without having to store all \( N^2 \) possible bonds. To do this, Verlet lists described in section 4.5.2 are used to track elements which are close enough to each other that they are likely to interact. This saves on computation and memory but also allows for a way to associate the BondStatus data to a pair of potentially bonded elements. The Verlet list is update periodically in the re-mesh block in Figure 3.1 which describe in the simulation description section. To build the new Verlet list, the binning algorithm in section 4.5.3 is used. Finally, it is necessary that BondStatus data from the old Verlet list be copied to the new list. For each element in the new Verlet list a match(having the same two elements) in the old Verlet list is looked for, if found the BondStatus from the old Verlet list data is copied to the new list. If a match is not found in the the old list, the BondStatus is initialized to 0. If an entry no longer exists in the new Verlet list BondStatus is thrown out and if the Bond is found again it will be re-initializated to 0. While the loss of data is not a perfect system, it is what is done at the moment to reduce computational cost.

By having the Verlet list, instead of looping over all elements in the system, now the particle-particle sum only goes over entries in the Verlet list as below

\[
F_{i,tot} = \sum_{j=\text{Verlet}_i} F_{ij,\text{part-part}} + F_{i,\text{rect-part}} + F_{i,\text{gravity}} + F_{i,\text{DragPower}}
\]

where \( \text{Verlet}_i \) indicates connections that have element \( i \) in it. In practice the list is not grouped by a particular element so the list is traversed and elements are input into the global \( F \) vector by indexing and adding them to the correct spots. There is only one base plate rectangle in this simulation so no sum is needed on the force with the plate.

The force models now have an additional conditional for when the BondStatus is greater than 1.

\[
F_{ij,\text{part-part}} = \begin{cases} 
F_{ij,\text{PowerContact,ss}} + F_{ij,\text{PowerDrag\perp,ss}} + F_{ij,\text{PowerDrag\parallel,ss}} & \text{BondStatus < 1} \\
F_{ij,\text{SymPowerContact,ss}} + F_{ij,\text{SymPowerDrag\parallel,ss}} & \text{BondStatus \geq 1}
\end{cases}
\]

\[
F_{i,\text{gravity}} = \begin{cases} 
M_i \begin{pmatrix} 0 \\ 0 \\ -g \end{pmatrix} & r_{i,z} < 0 \\
0 & r_{i,z} > 0
\end{cases}
\]

where the subscript \( ss \) indicates \( \delta \) and \( n_c \) are calculated using the Sphere-Sphere equation.

As a reminder, \( Sym \) indicates the versions of the model where conditional \( \delta < 0 \) has been
removed. Note that for the particle-particle bond, after the Bond is turned on, the \(\perp\) drag term is removed. If this is not done, an unphysical internal force is applied which prevents the bonded body from rotating as is shown in section 6.2. Also, gravity is only turned on below zero. This allows the use of the area above zero to be used to set up the particle inflow without being effected by gravity.

The force law with the plate, \(\mathbf{F}_{i,\text{rect-part}}\), is more complicated because more information is needed to create a bond with the plate. The reason for this is that the point of bonding, \(\mathbf{r}_{\text{bondpoint},i}\), must also be stored which is the location of element \(i\) after it becomes stuck. The bond point is then used in a simple linear spring force bonding model below

\[
\mathbf{F}_{i,\text{bond}} = K_{\text{bond}} (\mathbf{r}_{\text{bondpoint}} - \mathbf{r}_i)
\]

meant to act as a penalty to force it to stay at approximately its bond point. The BondState is also tracked using equation 7.1 where \(j\) now denotes the plate which has a constant temperature. \(\mathbf{r}_{\text{bondpoint},i}\) is tracked by giving it the same rate of change as the element \(i\) until bonding occurs at which point the bond point is set and no longer allowed to change. This is accomplished by the differential equation for the evolution of the bond point below

\[
\frac{d\mathbf{r}_{\text{bondpoint},i}}{dt} = \begin{cases} 
\mathbf{v}_i & \text{BondStatus} < 1 \\
0 & \text{BondStatus} \geq 1 
\end{cases}
\]

where \(\mathbf{v}_i\) is the velocity of the particle being considered for a bond with the plate. With this, the force is

\[
\mathbf{F}_{i,\text{rect-part}} = \begin{cases} 
\mathbf{F}_{ij,\text{PowerContact},rs} + \mathbf{F}_{ij,\text{PowerDrag}\perp,rs} + \mathbf{F}_{ij,\text{PowerDrag}\|,rs} & \text{BondStatus} < 1 \\
\mathbf{F}_{i,\text{bond}} + \mathbf{F}_{ij,\text{SymPowerDrag}\perp,rs} + \mathbf{F}_{ij,\text{SymPowerDrag}\|,rs} & \text{BondStatus} \geq 1 
\end{cases}
\]

where \(rs\) denotes the rectangle-sphere equation is used to find \(\delta\) and \(n_c\). The element will be in contact with the plate when it bonds as it must be in contact to cool to initiate bonding. The drag with the plate is used to damp unwanted vibrations from the spring force which acts as a penalty to bond the element to the plate.

Finally, the temperature governed by equation 4.19. The right hand side, the total heat rate in and out of the element has a terms to model the ability to loose heat to the plate and to adjacent elements, either during contact or if bonded. The result is that

\[
Q_{i,\text{tot}} = \sum_{j=\text{Verlet}_i} Q_{ij,pp} + \sum_{j=1} Q_{ij,\text{plate}}
\]

\[
Q_{ij,pp} = \begin{cases} 
Q_{ij,\text{HeatTransfer}} & \text{BondStatus} < 1 \\
Q_{ij,\text{SymHeatTransfer}} & \text{BondStatus} \geq 1 
\end{cases}
\]

\[
Q_{ij,\text{plate}} = \begin{cases} 
Q_{ij,\text{HeatTransfer}} & \text{BondStatus} < 1 \\
Q_{ij,\text{SymHeatTransfer}} & \text{BondStatus} \geq 1 
\end{cases}
\]
where for $Q_{ij,\text{plate}}$, the role of the other element $j$ is played by the plate and $\Theta_j$ is the temperature of the plate which is assumed to be constant. The radii of the elements are again determined by the equation of state

$$R_i(T) = R_{T_0,i}(1 + \alpha_{\text{therm}}(\Theta_i - \Theta_0))$$

where $\alpha_{\text{therm}} = 1 \times 10^{-5}/K$. Note that as the particle cools the radius decreases. This means that the equilibrium distance for the force calculation in the bond which depend on the radii will also shrink creating a tensile force in the bond.

Particles are initialized with a base radii, $R_{T_0,i}$, which are in chopped Gaussian distribution with mean diameter $D_{\text{mean}} = 27\mu m$ with full width half maximum (FWHM) of $FWHM = 1.17D_{\text{mean}} = 31.6\mu m$ which is a standard deviation of $\sigma = 13.4\mu m$ and minimum and maximum diameter, $D_{\text{min}} = 17\mu m$ and $D_{\text{max}} = 42\mu m$, respectively. This is the same as used in the previous sections. $\Theta_0 = 300K$ which is also the temperature of the plate. The elements start the simulation with the temperature $\Theta_i = 2500K$. The spherical elements, with the expanded radii, are then packed into a rectangular column with dimensions $45\mu m \times 45\mu m \times 7000\mu m$ centered around $x = y = 0$ and with the minimum $z$ position at 0. The elements are given an initial velocity in the negative $z$ direction of $0.15m/s$. Because gravity only acts on particles in $z < 0$, there is no force on the particles until they fall below $z = 0$. This sets where(x,y position) and what rate powder would be coming out of a nozzle at $z = 0$.

The drag parameters are set to be quite large, $\gamma_{||,\text{part-part}} = \gamma_{||,\text{part-rect}} = 8.4 \times 10^{-6}N\frac{m}{s}$, in order to reproduce the low coefficient of restitution expected from a liquid drop splatting on a surface. The drag perpendicular is high but not as high, $\gamma_{\perp,\text{part-part}} = \gamma_{\perp,\text{part-rect}} = 8.4 \times 10^{-7}N\frac{m}{s}$, to represent the viscous flow of a molten metal. For the actual computations, a unit system is chosen to avoid large differences in magnitude of the floating point numbers. The length unit used is $\text{LengthUnit} = 100\mu m$, the time unit is $\text{TimeUnit} = 0.01\text{s}$, and the mass unit is chosen so the an element of diameter $D_{\text{mean}}$ would have a magnitude of 1 which is found by $\text{MassUnit} = \rho \frac{4}{3}\pi \left(\frac{D_{\text{mean}}}{2}\right)^3$. A full list of parameters can be found in appendix D.2.

### 7.3.2 Results

To simulate the production of a ring, the positions (currently a column) are first transformed into helix so that they fall onto the plate in a circular pattern. The helix transformation takes the form

$$r_x = r_x + 250\mu m \cos \left(\pi \frac{r_z}{500\mu m}\right)$$

$$r_y = r_y + 250\mu m \sin \left(\pi \frac{r_z}{500\mu m}\right)$$

$$r_z = 1.25 \times r_z$$
This produces a helix with a diameter of $250\mu m$ which finishes a revolution every $1000\mu m$ rise in the $z$-direction. Figure 7.8 shows the helix first few cycles of the helix. Figure 7.9 shows a few representative time steps in the evolution of the deposition. Finally, Figure 7.10 shows final state but with elements removed and only the lines representing the bond shown.

The same thing can be done to model the deposition of a straight wall. Instead of forming initial positions into a helix, the initial positions are transformed to be stacked angles lines as shown in Figure 7.8. A wall is built up layer by layer as bonds are formed between cooling elements as seen in Figure 7.11. Here the temperature gradient in the wall during deposition is more apparent. Ultimately, defects in the wall cause it to tip over as seen in the right image in Figure 7.11. This is caused by the random nature of the particle size and deposition location which left an unsupported region in the base reducing strength.

Finally, an example of a pillar being deposited is shown in Figure 7.12. This is created by the untransformed initial pillar of elements. The most interesting thing to observe here is that the resulting pillar is much larger than the incoming stream of elements. This is because the elements stay hot on a time scale much longer than the flow rate of the hot elements. Thus the elements hit the top of the pillar and have a long time to flow to the perimeter be solidifying and bonding resulting in a solid pillar much larger than the initial stream. Understanding effects of cooling rates on final part production would be very valuable. For example, if conduction alone results in a cooling rate that is too slow additional cooling mechanisms such as forced air cooling could be designed to lower cooling time. This would allow for thinner parts to be made.

The models presented require less than a core hour on a personal laptop making them
Figure 7.9: Figures showing representative stages in evolution of the deposition of the ring of material. The lines between elements are bonds that have formed.
Figure 7.10: The final state of the simulation shown in the bottom right image of Figure 7.9 without the discrete elements to show the bonding structure.

Figure 7.11: Left: Wall is being created by layered deposition of metal onto the top: Right: The final wall as it falls over. The random nature of the particle size and deposition location left a defect near the base causing it to tip.
efficient enough to be a viable method for design purposes. In future work, forces that result from the differential cooling rates will calculated and measurements of residual stress made using techniques presented in Zhu. et al. [109] which provides an overview of method used to calculate the stress tensor based on forces in a volume. This same model could be used along with the Gaussian heat source to model part production in SLS and can be used to study surface roughness of final parts but is left to future work. Also, at the moment the model is not capable of producing observed wetting effects such as clumping of melt on an solid substrate[34]. In future work, the addition of near field forces as described in Chapter 5 will be investigated as a means to reproduce surface tension effects, a method used by the author in previous work[37].

7.4 DEM Enhanced Eulerian FEM for Laser Melting

Selective Laser Melting (SLM) is another powder based additive manufacturing which is nearly identical to Selective Laser Sintering (SLS). It also works by depositing thin layers of metal power into the work area from reservoir. The powder in the work area is selectively heated by a laser until the powder fully melts and then re-solidifies to form the next layer of the part. This process repeats until an entire part is formed. The process is essentially identical to the SLS process in Figure 7.1 except in SLM the powder is heated to full melt while in SLS the powder is usually only heated enough to induce bonding between adjacent powder particles without significantly changing the shape of the particle.

An interesting phenomena observed in SLM is the denudation region which is an area
Figure 7.13: Left: A single track of laser melting showing the expected denudation zone beside the melt track. [106] Right: ALE3D simulation of a single melt tracking show the failure to reproduce this denudation zone. In the red circle, advection welded particles are not pulled in by the melt that touches them. Areas are colored by temperature. Image provided by Andy Anderson and Saad Khairallah.

next to the melt pool that is devoid of powder which can be seen in Figure 7.13\(^1\). The effect is generally attributed to the surface tension forces of the melt pool pulling in powder particles that it touches leaving regions without powder. The size of the effect is enhanced by the shrinking of the melt pool as it cools. Saad Khairallah and Andy Anderson presented a detailed mesoscopic simulation of a laser melting powder using ALE3D [59]. However, their simulation does not reproduce the denudation region beside the melt pool. This is seen in Figure 7.13 in the region circled in red where it is observed that particles exist directly adjacent to the melt pool. Also seen are bridges of solidified melt which have reached the particle but did not induce any motion.

In the paper, Khairallah and Anderson fully reproduce the melt and surface tension using an Eulerian finite element simulation. In collaboration with them, it was determined that the likely cause for the lack of denudation zone is an effect sometimes referred to as advection welding. In an Eulerian formulation, moving material must be advected through the mesh. However, because the mesh does not track the material boundaries as two objects approach and when the distance between them is less then the size of the finite element cell there is no longer anyway for the formulation to distinguish between the two objects. If there is supposed to be a discontinuity, e.g. between two rigid bodies, the discontinuity is lost and to the formulation it appears as a single solid object. The result being that the bodies are effectively become welded giving rise to the name advection welding. In this case, the powder particles are advection welded to the substrate and are thus unable to move when pulled by the surface tension forces of the melt. Since they are not pulled in as a real powder would be, the denudation zone is not reproduced. This can

be seen in Figure 7.13 in the region in the red circle where the melt has reached out to a solid particle but the particle is unmoved.

Here, it is proposed that discrete elements are an effective and computationally cheap method by which to solve for motion of the material and track the discontinuity associated with a solid powder particle. For this, the particle is still fully shaped into the Eulerian finite element simulation from the start. A discrete element is also created at the same point in space with the same shape, in this case a sphere.

The behavior expected of two distinct rigid bodies is that there should not be able to support any tensile forces at the boundary. This is true in discrete elements because of the $\delta < 0$ conditional in the force models. In finite elements, the unintended addition of the tensile strength at the boundary is the root of the of the advection welding. These can be prevented by completely removing the strength associated to the material that falls within the discrete element. Because there is no strength, two adjacent nodes do not produce any forces between each other in the finite element simulation. In the continuum balance of momentum equations

$$\rho \frac{d^2 \mathbf{x}}{dt^2} = \nabla \cdot \mathbf{\sigma} + \mathbf{b}$$

where $\mathbf{\sigma}$ is the Cauchy stress tensor, this means that the $\nabla \cdot \mathbf{\sigma}$ term is zero in elements that are inside the region of the discrete element and thus no force is added to the dynamics of the nodes in and adjacent to the element thus preventing the welding effect.

While the tensile force has terms have been removed, so has all strength associated to the body. The rigid body constraint for the particle must now be satisfied as well as the balance laws. For this, the DEM formulation presented in Chapter 5 is used. The total force on the discrete element is calculated at each step and the ODEs governing dynamics, equations 4.18, are integrated using Forward Euler to get the next position and velocity for the material associated to the discrete element. The motion of the particle’s material in the FEM simulation is entirely dictated by the state of the discrete element. This is accomplished by using a velocity boundary condition in the FEM simulation. Any node that falls within the bounds of the DEM element are set to have the velocity of the DEM element. By mirroring the material behavior of the discrete element in the FEM simulations, the rigid body constraint is satisfied and balance laws are also satisfied in the FEM simulation.

It should be noted that while the material has no strength, the material associated to the DEM element in the FEM simulation still plays a role in force calculation. Because the particle is shaped and moved in the FEM simulation, it interacts with other material via other force models. In the FEM simulation, these forces effect the dynamics of material around the element but do not inherently effect the dynamics of the material within the element. To fix the lack of balance, at each time step the forces not associated to the material strength from the FEM simulation must be summed and added to the total force acting on the DEM element to retain a global balance. This system of using the FEM simulation to also calculate forces on the DEM element is useful for two reasons,
first some models are more appropriate for the continuum and secondly, the existing free surface tracking in ALE3D of the fluid and the built in surface tension models can be reused providing a significant savings in development time and validation.

The simulation is marched with explicit time stepping as shown in Figure 7.14. First, the initial conditions are set for the problem and the initial velocity of the particle is set both in the DEM simulation and on the associated nodes in the FEM simulation. Next, ALE3D conducts a time step of a length selected adaptively based on physics and stability to update the state of the finite element simulation to time $t + \Delta t$. To update the DEM state, the forces calculated from the FEM state at time $t$ are calculated and added to the force balance as well as contact models for the DEM interaction with solids in the simulation, which is a linear spring model equation 5.4 aka equation 5.5 with $\alpha = 1$, $K_{eff} = K$. The DEM state is update with Foward Euler using the time step length to match ALE3D. This process is repeated with the ALE3D simulation advancing with new set of nodes and velocity set by the new state of the DEM elements.

To study the ability of this procedure to fix the advection welding problem, a simple test problem is used of a single particle bouncing off a flat body at a 45 degree angle. First, on the left hand side of Figure 7.15 (visualized with VisIt[23]) shows the baseline simulation using only the Eulerian FEM simulation with ALE3D. The particle and flat base substrate are shaped into the mesh and material in the particle is given an initial velocity at 45 degrees. The simulation is allowed to evolve without intervention. The result is that the particle moves through the mesh until it reaches the substrate where it hits and the resistance to compression stops it motion. However, the particle and the substrate share elements and become welded by the tensile strength causing the particle to permanently stick to the substrate. On the right of Figure 7.15 is the DEM enhanced simulation. The spherical discrete element (not shown) moves causing the particle material in ALE3D to move with it. When its hits the surface coincident with the substrate, The discrete element bounces as does the material in the FEM simulation. The expected bounce is reproduced in the FEM simulation demonstrating that the welding problem has be corrected.

The method represents a very cheap solution to the welding problem. The mesh for a rectilinear Eulerian simulation requires about 10 elements per side to resolve the particle.
Figure 7.15: Left: A enhanced ALE simulation of a solid particle hitting a solid substrate demonstrate the sticking as a result of advection welding. Right: The same simulation but enhanced with a Lagrangian discrete element.
This means that in a 3-D simulation there is about 1000 nodes associated to a single particle. The discrete element will have at most 12 degrees of freedom for positional and rotational degrees of freedom which only increases the amount of information associated with a particle by 1.2%. Here only translational dynamics are considered so only 6 additional degrees of freedom are required per particle representing an even smaller addition to the complexity of a particle. Globally, the effect of the additional degrees of freedom per particle is even smaller because there are yet more degrees of freedom associated with the substrate and volume around the particles so the relative increase is even smaller for the simulation as a whole.

As with all discretization, the method does have limitations. Inherent in the discrete element is a rigid body constraint so if the deformation of the particle is important the model will not represent the physics as well. Also, the spherical element effects nodes on a rectilinear grid so the spherical geometry of the element is not represented exactly in the FEM simulation. Some deformation can result from this as can be seen in the last frame on the right side of Figure 7.15. Also seen in the last frame is that sometimes the discrete element overlaps with the substrate and erroneously controls the substrate material for a brief time leading to some unintended deformation. As is true with all discretization there are pros and cons, this method solves a significant problem in a cost efficient ways with a small loss in fidelity of the shape and possible deformation of the material.
Chapter 8

Conclusions

The goal of this dissertation has been two fold. First, to present a novel approach to computational mechanics via a Domain Specific Language specifically designed for the goals of computational mechanics. The DSL presented facilitates the development of models for computational mechanics by providing a syntax which makes it easy to make new discretizations and test different constitutive models. Second, discrete element multiphysical models for powder based additive manufacturing processes are presented. Using the DSL, these models are implemented and investigated.

The Domain Specific Language is a tool for computational mechanics which is designed to allow an engineer to focus more on model development and investigation. In order to achieve this goal, a holistic ecosystem for managing data and specifying models that is easy to use is created. As a first step to usability, all actions by the user are performed within the Python language. This high level language provides unparalleled ease in manipulating data and managing programs. With the syntax defined, the user can setup the entire physical description including discretization and constitutive models within Python. There are three powerful features of the DSL that are responsible for the great ease of setting up a model.

First, the language provides a syntax for creating objects which describe the fundamental physical units in the problem. This implements a system for naming and tracking all of the data. The objects are used for allocating the data arrays and provide an easy management system for the user by automatically providing a namespace that matches the names used to create the fundamental data units. Also, when the items in the namespace are accessed data is returned from the larger data array into appropriately shaped subarrays to match the physical context of the data.

Second, all constitutive modeling is input using symbolic notation. The user works by creating symbols to match the data units which in turn can be assembled into linear algebra concepts such as vectors and matrices. Using the features of SymPy, constitutive equations can be expressed with a very high level syntax that match the usual mathematical concepts used in model development. This includes operations such as taking dot products, multiplying matrices and vectors, derivatives, matrix inversions and norms. By
describing the models with this very high level syntax it is quick and easy to create new models and be confident that they match the intended description. If errors do occur, the model is read in the language of math for debugging, not computer code.

Finally, the first two allow for the completely automated generation of low level code for both the function, which describes the system, and its gradient. As part of the code generation, the idea of indexing is almost completely removed. Because the data has been allocated and tracked by the DSL from the start, the code generation knows exactly which part of the data array corresponds to which concepts and can determine the correct index to extract the appropriate piece of data. This alone results in a significant reduction in errors by the developers. The other major gain with the symbolic descriptions of the model is that the gradient can also be automatically calculated. For a methods developer who is constantly exploring new material models and discretization, the process of deriving and programing gradients can be time consuming. By making the creation of the gradient automatic, an engineer is free to explore both linear and non-linear problems, both dynamic and steady state, without out any extra work in describing the model.

The second major pursuit of the dissertation is the investigation of discrete element multiphysical models for additive manufacturing. This application is cumbersome in that it requires a variety of heterogeneous data models to described the concepts of the discretization, the geometrically different objects in the simulation domain and bonding behaviors. The DSL makes it easy to setup the disparate data structures and their interactions without having to manage the tedious low level code. A model for a spherical discrete element which has a temperature and can expand thermally is presented. Also, methods for defining lines and rectangles are presented and used to create complex geometries in the simulation domain. Finally, multiphysical constitutive models are established for the interactions between elements for contact, bonding, heat transfer and heat sources.

Characteristics of the model were first investigate. A convergence study of discrete elements using both explicit and implicit Runge-Kutta integration methods were performed. It is noted that the smoothness of the constitutive models directly affects the observed convergence rate of the integration technique. The convergence rate is limited by the how many continuous derivative the constitutive models have. This conclusion is important to discrete element models as often the form of the constitutive model is dictated by the physics of the system being studied. Also investigated is the potential to make unphysical bonding models. Specifically, the inclusion of the perpendicular drag term has the effect of preventing a rigid body created with bonded discrete elements from rotating. When contacting an item that should rotate, the body bounces as if the force where applied at the center of mass. The problem is corrected by removing the perpendicular drag term. As a credit to the DSL, this behavior was discovered while setting up a sieving simulation. The ease of setting up and testing new models made it possible to setup up enough simulations that this issue was found.

Finally, the multiphysical discrete element formulations is then used to study powder packing in a manner that is physically consistent with SLS and SLM machines leading to conclusions about changes in packing density and powder size distribution resulting from
the wiping processes used to pack the work area. Also, the hypothesis that the thermal expansion of the powder in the bed is responsible for the observation that powder pops out of the work area was tested and determined to be an unlikely cause of the phenomena. Next, a temperature dependent bonding model is used to create a efficient simulation for studying Laser Metal Deposition. In this simulation, hot elements form bonds when in contact with other elements or the substrate as they cool. The model is efficient enough that is possible to use it for parts design. Finally, a model for using a discrete element to enhance Eulerian finite elements to prevent advection welding is proposed. This enhancement could be used to allow for a more physically correct simulation of SLM by allowing meshed in particles to move relative to the substrate before they melt.

The work in this dissertation describes a new system for model development for computational mechanics facilitated by the DSL. The DSL allows an engineer to focus on the model development and science by providing an easy syntax for setting up models and automatic generation of low level code. With it the author has investigated a range models to to simulate powder based additive manufacturing systems.
Bibliography


Appendix A

Contact Geometry

Assume that contact occurs along the line connecting the circle centers and along the plane connect the the intersection points which are perpendicular to each other. To find the angle $\theta$ the law of cosines is used

$$R_j^2 = R_i^2 + \|r_{ij}\|^2 - 2 \|r_{ij}\| R_i \cos \theta$$

$$\cos \theta = \frac{-R_j^2 + R_i^2 + \|r_{ij}\|^2}{2 \|r_{ij}\| R_i}$$

Then $l_i$ the length from $r_i$ to the contact point is

$$l_i = R_i \cos \theta$$

$$l_i = \frac{-R_j^2 + R_i^2 + \|r_{ij}\|^2}{2 \|r_{ij}\|}$$

(A.1)

Then

$$l_j = \|r_{ij}\| - l_i$$

(A.2)

Figure A.1: Velocity Components of two colliding rigid bodies demonstrated on two spheres.
The length from the contact point circle intersection, \( d \), is

\[
d = R_i \sin \theta
\]
\[
d = \sqrt{R_i^2 - l_i^2}
\]  
(A.3)
Appendix B

Butcher Tableaus

\[
\begin{array}{c|c}
  c & a \\
  \hline
  b[O(h^1)] & b^T
\end{array}
\]

Table B.1: The Butcher tableau key. \(a\) is an \(s \times s\) matrix and \(c\) and \(b\) are length \(s\) vectors used in Runga-Kutta integration described in section 3.2.

B.1 Explicit

B.1.1 Forward Euler

\[
\begin{array}{c|c}
  0 & 0 \\
  \hline
  b[O(h^1)] & 1
\end{array}
\]

Table B.2: The Butcher array forward Euler.

B.1.2 Explicit Midpoint

\[
\begin{array}{c|ccc}
  0 & 0 & & \\
  \hline
  \frac{1}{2} & \frac{1}{2} & 0 \\
  b[O(h^2)] & 0 & 1
\end{array}
\]

Table B.3: The Butcher array for Explicit Midpoint.
B.1.3 RK3

\[
\begin{array}{c|cccc}
0 & & & & \\
\frac{1}{2} & \frac{1}{2} & & \\
1 & -1 & -2 & 0 \\
\hline
b[O(h^3)] & \frac{1}{6} & \frac{2}{3} & \frac{1}{6}
\end{array}
\]

Table B.4: The Butcher array for RK3.

B.1.4 RK38th

\[
\begin{array}{c|cccc}
0 & & & & \\
\frac{1}{3} & \frac{1}{3} & & \\
\frac{2}{3} & -\frac{1}{3} & 1 & \\
1 & 1 & -1 & 1 \\
\hline
b[O(h^4)] & \frac{1}{8} & \frac{3}{8} & \frac{3}{8} & \frac{1}{8}
\end{array}
\]

Table B.5: The Butcher array for RK38th.

B.1.5 RK4

\[
\begin{array}{c|cccc}
0 & & & & \\
\frac{1}{2} & \frac{1}{2} & & \\
\frac{1}{2} & 0 & \frac{1}{2} & \\
1 & 0 & 0 & 1 \\
\hline
b[O(h^4)] & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6}
\end{array}
\]

Table B.6: The Butcher array for RK4.

B.1.6 Dormand-Prince

This is an embedded integration scheme which can be used for time adaptivity.
Table B.7: The Butcher array for Dormand-Prince method[85]

<table>
<thead>
<tr>
<th>0</th>
<th>1.5</th>
<th>1.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>3/10</td>
<td>3/40</td>
<td>9/40</td>
</tr>
<tr>
<td>4/5</td>
<td>44/45</td>
<td>-56/15</td>
</tr>
<tr>
<td>8/9</td>
<td>19372/6561</td>
<td>-25360/6561</td>
</tr>
<tr>
<td>1</td>
<td>9017/3168</td>
<td>-355/33</td>
</tr>
<tr>
<td>1</td>
<td>35/384</td>
<td>0</td>
</tr>
<tr>
<td>b[O(h^5)]</td>
<td>35/384</td>
<td>0</td>
</tr>
<tr>
<td>b[O(h^4)]</td>
<td>5179/57600</td>
<td>7571/57600</td>
</tr>
<tr>
<td>error</td>
<td>71/57600</td>
<td>0</td>
</tr>
</tbody>
</table>

Table B.8: The Butcher array backward Euler

B.2 Implicit

B.2.1 Backward Euler

\[
\begin{array}{cccc}
0 & 1 \\
\hline
b[O(h^1)] & 1 \\
\end{array}
\]

Table B.9: The Butcher array backward Euler

B.2.2 Implicit Midpoint

\[
\begin{array}{cccc}
.5 & .5 \\
\hline
b[O(h^1)] & 1 \\
\end{array}
\]

Table B.9: The Butcher array backward Euler

B.2.3 Gauss Legendre 4

This is a fully implicit 2-stage methods that is 4th order. It is based off the Gauss-Legendre integration method.
\[
\begin{array}{ccc}
\frac{1}{2} - \frac{\sqrt{3}}{6} & \frac{1}{4} & \frac{1}{4} - \frac{\sqrt{3}}{6} \\
\frac{1}{2} + \frac{\sqrt{3}}{6} & \frac{1}{4} + \frac{\sqrt{3}}{6} & \frac{1}{4} \\
\text{b}[O(h^4)] & \frac{1}{4} - \frac{\sqrt{3}}{6} & \frac{1}{4} + \frac{\sqrt{3}}{6}
\end{array}
\]

Table B.10: The Butcher array for Gauss Legendre 4.
Appendix C

Polynomial Forces

C.1 Cubic Interpolation

C.1.1 General cubic interpolation

Given two points with defined derivatives, a unique cubic function, \( P(x) = a + bx + cx^2 + dx^3 \) which is continuous and the first derivative is continuous can be found. The derivative is, \( P'(x) = b + 2cx + 3dx^2 \). Since it is given that we know \( P(x_1), P(x_2), P'(x_1), P'(x_2) \), a system of linear equations which needs to be satisfied can be written in the matrix form

\[
\begin{pmatrix}
1 & x_1 & x_1^2 & x_1^3 \\
1 & x_2 & x_2^2 & x_2^3 \\
0 & 1 & 2x_1 & 3x_1^2 \\
0 & 1 & 2x_2 & 3x_2^2
\end{pmatrix}
\begin{pmatrix}
a \\
b \\
c \\
d
\end{pmatrix}
= 
\begin{pmatrix}
P(x_1) \\
P(x_2) \\
P'(x_1) \\
P'(x_2)
\end{pmatrix}
\]

Then the coefficients \( a, b, c, d \) are found by inverting the matrix

\[
\begin{pmatrix}
a \\
b \\
c \\
d
\end{pmatrix}
= 
\begin{pmatrix}
1 & x_1 & x_1^2 & x_1^3 \\
1 & x_2 & x_2^2 & x_2^3 \\
0 & 1 & 2x_1 & 3x_1^2 \\
0 & 1 & 2x_2 & 3x_2^2
\end{pmatrix}^{-1}
\begin{pmatrix}
P(x_1) \\
P(x_2) \\
P'(x_1) \\
P'(x_2)
\end{pmatrix}
\]

C.1.2 Special Case

In the special case when the value and derivative are both zero at one of the points i.e. \( P(x_1) = 0, P'(x_1) = 0 \), the solution is easily attained[45]. First, that the condition \( P(x_1) = 0 \) is satisfied if

\[
P(x) = a(x - x_1)^3 + b(x - x_1)^2 + c(x - x_1)
\]

Then \( P'(x) = 3a(x - x_1)^2 + 2b(x - x_1) + c \) so \( P'(x_1) = 0 = c \). So the interpolating function is now

\[
P(x) = a(x - x_1)^3 + b(x - x_1)^2
\]
Solving the system of equations

\[
P(x_2) = a(x_2 - x_1)^3 + b(x_2 - x_1)^2 \\
\frac{dP}{dx}(x_2) = 3a(x_2 - x_1)^2 + 2b(x_2 - x_1)
\]

for the coefficients a and b gives

\[
a = \frac{\frac{dP}{dx}(x_2)(x_2 - x_1) - 2P(x_2)}{(x_2 - x_1)^3} \\
b = \frac{3P(x_2) - \frac{dP}{dx}(x_2)(x_2 - x_1)}{(x_2 - x_1)^2}
\]

In total

\[
P(x) = \frac{\frac{dP}{dx}(x_2)(x_2 - x_1) - 2P(x_2)}{(x_2 - x_1)^3}(x - x_1)^3 + \frac{3P(x_2) - \frac{dP}{dx}(x_2)(x_2 - x_1)}{(x_2 - x_1)^2}(x - x_1)^2
\]
Appendix D

Simulation Parameter listing

D.1 Powder Packing Parameters

The values of the parameters below are reported in SI units but are converted for the simulation to the unit system. The length unit used is $LengthUnit = 100\mu m$, the time unit is $TimeUnit = 0.01$ seconds, and the mass unit is chosen so the an element of diameter $D_{\text{mean}}$ would have a magnitude of 1 which is found by $MassUnit = \rho \frac{4}{3} \pi \left( \frac{D_{\text{mean}}}{2} \right)^3$.

- width = 200$\mu m$
- midlen = 100$\mu m$
- resvbedlen = 200$\mu m$
- workbedlen = 200$\mu m$
- midheight = $3 \times 42 \mu m = 126 \mu m$
- TotHeight = midheight + 200$\mu m$
- leftmid = −midlen
- leftside = leftmid − resvbedlen
- rightside = workbedlen
- $D_{\text{mean}} = 27 \mu m$
- $FWHM = 1.17 D_{\text{mean}} = 31.6 \mu m$
- $\sigma = 13.4 \mu m$
- $D_{\text{min}} = 17 \mu m$
• $D_{max} = 42 \mu m$
• $\rho = 7900 \frac{kg}{m^3}$
• $E_{\text{particle}} = E_{\text{infwall}} = E_{\text{rect}} = 8141.7 Pa$
• $\nu_{\text{particle}} = \nu_{\text{infwall}} = \nu_{\text{rect}} = .3$
• $E_{\text{wiper}} = 81.4 Pa$
• $\nu_{\text{wiper}} = .3$
• $g = 9.8 \frac{m}{s^2}$
• $\gamma_D = 8.4 \times 10^{-10} \frac{N}{s}$
• $\alpha_D = 1$
• $\gamma_{\perp,\text{part-part}} = 2.35 \times 10^{-8} \frac{N}{s} - 1 \times 10^{-6} \frac{N}{s}$
• $\gamma_{\perp,\text{part-infwall}} = \gamma_{\perp,\text{part-rect}} = 8.4 \times 10^{-10} \frac{N}{s}$
• $\gamma_{\parallel,\text{part-part}} = \gamma_{\parallel,\text{part-infwall}} = \gamma_{\parallel,\text{part-rect}} = 8.4 \times 10^{-10} \frac{N}{s}$
• $\alpha_{\text{part-part}} = 1.5$
• $\alpha_{\parallel,\text{part-infwall}} = \alpha_{\parallel,\text{part-rect}} = 1$
• $\alpha_{\perp,\text{part-part}} = \alpha_{\perp,\text{part-infwall}} = \alpha_{\perp,\text{part-rect}} = 1$
• $K_{eff} = \frac{4}{3} E^* \sqrt{R^*}$
• $E^* = \frac{E_i E_j}{E_j (1-\nu_i^2) + E_i (1-\nu_j^2)}$
• $R^*_{\text{part-part}} = \frac{R_i R_j}{R_i + R_j}$
• $R^*_{\text{infwall-part}} = R^*_{\text{rect-part}} = R_{\text{sphere}}$
• $l_{\text{lower}} = 30 \mu m$
• $l_{\text{raise}} = 42 \mu m$
• $l_{\text{sweep}} = \text{rightside - leftside}$
• $T_i = 0 \text{seconds}$
• $T_f = 0.5 \text{seconds}$
• $dt = 1 \times 10^{-6} \text{seconds}$
• Integration Method: Forward Euler
D.2 Laser Metal Deposition Parameters

The values of the parameters below are reported in SI units but are converted for the simulation to the unit system. The length unit used is LengthUnit = 100µm, the time unit is TimeUnit = 0.01seconds, and the mass unit is chosen so the an element of diameter $D_{\text{mean}}$ would have a magnitude of 1 which is found by $\text{MassUnit} = \rho \frac{4}{3} \pi \left(\frac{D_{\text{mean}}}{2}\right)^3$.

- $g = 9.8 m/s^2$
- $\gamma_D = 8.4 \times 10^{-10} \frac{N}{\mu m}$
- $\alpha_D = 1$
- $\gamma_{\perp,\text{part-part}} = \gamma_{\perp,\text{part-rect}} = 8.4 \times 10^{-7} \frac{N}{\mu m}$
- $\alpha_{\perp,\text{part-part}} = 1.5$
- $\alpha_{\parallel,\text{part-part}} = \alpha_{\parallel,\text{part-rect}} = 1$
- $\alpha_{\perp,\text{part-part}} = \alpha_{\perp,\text{part-rect}} = 1$
- $E_{\text{particle}} = E_{\text{rect}} = 8141.7 Pa$
- $K_{\text{eff}} = \frac{4}{3} E^* \sqrt{R^*}$
- $E^* = \frac{E_i E_j}{E_i (1 - \nu_j^2) + E_j (1 - \nu_i^2)}$
- $R^*_{\text{part-part}} = \frac{R_i R_j}{R_i + R_j}$
- $R^*_{\text{rect-part}} = R_{\text{sphere}}$
- $K_{\text{bond}} = 8.1 \times 10^{-2} N/m$
- $K_{\text{bondStatus}} = 10000$
- $\Theta_{\text{solid}} = 1700 K$
- $\Theta_{\text{Bond}} = 1710 K$
- $C = 500 J/KgK$
- $K = 1 \times 10^8 \text{SimUnits}$
- $dt = 1.25 \times 10^{-6} \text{seconds}$
- Integration Method: Forward Euler