SOLVING RIGID MULTIBODY PHYSICS DYNAMICS USING PROXIMAL POINT FUNCTIONS ON THE GPU

By

Jeremy James Betz

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Major Subject: COMPUTER SCIENCE

Approved:

____
Dr. Jeff Trinkle, Thesis Adviser

Rensselaer Polytechnic Institute
Troy, New York

July 2011
(For Graduation August 2011)

CONTENTS

LIST OF TABLES vii

LIST OF FIGURES viii

ACKNOWLEDGMENT xi

ABSTRACT xii

1. INTRODUCTION 1
   - 1.1 Background 1
   - 1.2 Previous Works 3
   - 1.3 Outline 4

2. RIGID MULTIBODY DYNAMICS SIMULATION 5
   - 2.1 Basics and Notation 5
   - 2.2 Dynamics Formulations 6
     - 2.2.1 Discrete Time vs Instantaneous Models 6
     - 2.2.2 Generalized Matrices 6
       - 2.2.2.1 Basic Matrices 7
       - 2.2.2.2 Mass Matrix 7
       - 2.2.2.3 Wrench Matrices 7
     - 2.2.3 Complementarity Formulation 8
       - 2.2.3.1 The Complementarity Condition 8
       - 2.2.3.2 Mixed Linear Complementarity Problem 9
     - 2.2.4 Complementarity Based Dynamics 9
       - 2.2.4.1 Normal Contact 10
       - 2.2.4.2 Friction 10
       - 2.2.4.3 Instantaneous Form 12
       - 2.2.4.4 Discrete Form 13
       - 2.2.4.5 Matrix Form 14
     - 2.2.5 Proximal Point Formulation 14
       - 2.2.5.1 Set Valued Normal Contact 15
       - 2.2.5.2 Set Valued Friction 16
       - 2.2.5.3 Discrete Time Proximal Point 17
   - 2.3 Solving Proximal Point Problems 17

2.3.1 Fixed Point Iteration 17
2.3.2 Independent Auxiliary Parameter (r-value) 18
2.3.2.1 Relaxed Richardson 19
2.3.2.2 Relaxed Jacobi 19
2.3.2.3 Relaxed Gauss-Seidel 20
3. SOLVER IMPLEMENTATION 21
3.1 Implementation Basics 21
3.2 CPU (ProxSerial) 21
3.2.1 Build Data Structures 21
3.2.2 Solve Proximal Point Dynamics 22
3.2.2.1 Calculate r-value 22
3.2.2.2 Solve Normal Impulse 23
3.2.2.3 Solve Friction Impulse 23
3.2.2.4 Update Body Dynamics 24
3.2.2.5 Check for Convergence 24
3.2.3 Return Velocities 24
3.3 GPU Programming 25
3.3.1 Basic GPU Architecture 25
3.3.2 GPU Optimization 26
3.4 GPU (ProxCUDA) 27
3.4.1 Build Data Structures 27
3.4.2 Solve Proximal Point Dynamics 27
3.4.2.1 Calculate r-value 28
3.4.2.2 Solve Normal Impulse 28
3.4.2.3 Solve Friction Impulse 28
3.4.2.4 Update Body Dynamics 28
3.4.2.5 Check for Convergence 29
3.4.3 Return Velocities 29
4. RESULTS 30
4.1 Test System 30
4.2 Test Cases 30
4.2.1 Shapes 31
4.2.1.1 Hexagon 31
4.2.1.2 Regular Octagon 31

4.2.2 Friction Angle 31
4.2.3 Large Group 32
4.3 Accuracy 33
4.3.1 Special Case 33
4.3.2 Convergence 34
4.3.2.1 Richardson vs Jacobi 34
4.3.3 Normal Contact 35
4.3.3.1 ProxSerial 35
4.3.3.2 ProxCUDA 37
4.3.4 Friction 39
4.3.4.1 ProxSerial 41
4.3.4.2 ProxCUDA 44
4.3.5 Single vs Double Precision 46
4.4 Parameter Tuning 47
4.4.1 Divergence Detection 49
4.5 Performance 50
4.5.1 GPU Single vs Double Precision 55
5. CONCLUSION 61
REFERENCES 64
APPENDICES
A. PHYSICS REVIEW 67
A.1 Basic Physics 67
A.1.1 Kinematics 67
A.1.2 Dynamics 68
A.1.2.1 Normal Contact 69
A.1.2.2 Friction 69
A.1.2.3 External Forces 70
B. SOURCE CODE LISTINGS 71
B.1 Timestepper 71
B.1.1 STProx.h 71
B.1.2 STProx.cpp 71
B.2 Prox Solvers 73
B.2.1 CPSolverServer.h 73

B.2.2 ProxSerial 74
B.2.2.1 proxSerialDatastruct.h 74
B.2.2.2 proxSerial.h 75
B.2.2.3 proxSerial.cpp 76
B.2.3 ProxCUDA 88
B.2.3.1 proxCUDAInterface.h 88
B.2.3.2 proxCUDAInterface.cu 90
B.2.3.3 proxCUDA.h 92
B.2.3.4 proxCUDA.cpp 93
B.2.3.5 proxCUDAKernels.cuh 99
B.2.3.6 proxCUDAKernels.cu 100

LIST OF TABLES

4.1 Hardware specifications of test computer 30

4.2 Software specifications of test computer 30

4.3 Simulation run times for one group of 525 bodies 54

4.4 Simulation run times for four groups of 525 bodies (2100 bodies total) . 54

4.5 Simulation run times for ten groups of 525 bodies (5250 bodies total) . 54

LIST OF FIGURES

4.1 The Hexagon 31

4.2 The Regular Octagon 32

4.3 The basic test environment for friction, with 25 bodies falling on an angled surface 32

4.4 The basic test environment for many bodies, with 525 bodies falling on angled surfaces 33

4.5 The edge bump condition as handled by all three dynamics solvers . . . 34

4.6 Comparison of the X position of hexagonal bodies using PATH and proxSerial on a 5 degree surface 35

4.7 Comparison of the X position of octagonal bodies using PATH and proxSerial on a 5 degree surface 36

4.8 Comparison of the X position of hexagonal bodies using PATH and proxSerial on a 10 degree surface 37

4.9 Comparison of the X position of octagonal bodies using PATH and proxSerial on a 10 degree surface 37

4.10 Comparison of the X position of octagonal body 17 using PATH and proxSerial on a 10 degree surface, with different relaxation coefficients . 38

4.11 Comparison of the X position of hexagonal bodies using PATH and proxCUDA on a 5 degree surface 38

4.12 Comparison of the X position of octagonal bodies using PATH and proxCUDA on a 5 degree surface 39

4.13 Comparison of the X position of hexagonal bodies using PATH and proxCUDA on a 10 degree surface 40

4.14 Comparison of the X position of octagonal bodies using PATH and proxCUDA on a 10 degree surface 40

4.15 Comparison of the X position of hexagonal bodies using PATH and proxSerial on a 5 degree surface with a coefficient of friction of 0.1 . . . 41

4.16 The angle of body 17 during steady state for PATH and proxSerial . . . 42

4.17 Comparison of the effects of the relaxation coefficient on the X position of hexagonal body 17 using PATH and proxSerial on a 5 degree surface with a coefficient of friction of 0.1 43

4.18 Comparison of the X position of octagonal bodies using PATH and proxSerial on a 5 degree surface with a coefficient of friction of 0.1 . . . 44

4.19 Comparison of the X position of hexagonal bodies using PATH and proxSerial on a 10 degree surface with a coefficient of friction of 0.1 . . 45

4.20 Comparison of the X position of octagonal bodies using PATH and proxSerial on a 10 degree surface with a coefficient of friction of 0.1 . . 46

4.21 Comparison of the effects of the relaxation coefficient on the X position of hexagonal body 17 using PATH and proxSerial on a 5 degree surface with a coefficient of friction of 0.2 47

4.22 Comparison of the X position of octagonal bodies using PATH and proxSerial on a 5 degree surface with a coefficient of friction of 0.2 . . . 48

4.23 Comparison of the X position of hexagonal bodies using PATH and proxSerial on a 10 degree surface with a coefficient of friction of 0.2 . . 49

4.24 Comparison of the X position of octagonal bodies using PATH and proxSerial on a 10 degree surface with a coefficient of friction of 0.2 . . 50

4.25 Comparison of the X position of hexagonal bodies using PATH and proxSerial on a 5 degree surface with a coefficient of friction of 0.5 . . . 51

4.26 Comparison of the X position of octagonal bodies using PATH and proxSerial on a 5 degree surface with a coefficient of friction of 0.5 . . . 52

4.27 Comparison of the X position of hexagonal bodies using PATH and proxSerial on a 10 degree surface with a coefficient of friction of 0.5 . . 53

4.28 Comparison of the X position of octagonal bodies using PATH and proxSerial on a 10 degree surface with a coefficient of friction of 0.5 . . 55

4.29 Comparison of the effects of the relaxation coefficient on the X position of hexagonal body 17 using PATH and proxSerial on a 10 degree surface with a coefficient of friction of 0.5 56

4.30 Comparison of the X position of hexagonal bodies using PATH and proxCUDA on a 5 degree surface with a coefficient of friction of 0.1 . . 56

4.31 Comparison of the X position of octagonal body 17 using PATH and proxCUDA on a 5 degree surface with a coefficient of friction of 0.1 and 0.2 57

4.32 Comparison of the X position of hexagonal body 17 and octagonal body 17 using PATH and proxCUDA on a 5 degree surface with a coefficient of friction of 0.5 57

4.33 Comparison of the X position of hexagonal body 17 and octagonal body 17 using PATH and proxCUDA on a 10 degree surface with a coefficient of friction of 0.1 58

4.34 A section of the Large Group test simulation run after 1 second with each of the dynamics solvers 59

4.35 Result of CUDA Profiler on large group with 5250 bodies 59

4.36 Scaling of running times for all solvers 60

ACKNOWLEDGMENT

First, I would like to thank my parents, whom without their support, encouragement and motivation this would not have been possible. I would also like to thank my brother Jason, as well as my extended family who always had faith that I would get this thesis completed. I would like to thank my friends at school, for the conversations about topics related, and not so related, their help and their support. I would also like to thank my friends at home for their understanding when I was away, and their company when I could be around.

I would also like to thank all those in the Computer Science department, in particular, Terry Hayden, Pam Paslow, and Shannon Carothers for all their help in the administrative organizing, paperwork, and purchasing to make this possible. I would also like to thank all my professors for sharing their knowledge with me and expanding my horizons. I would also like to thank my lab group, in particular Cihan Caglayan, Jed Williams, Chris Jordan, Will Macaluso, Hans Vorsteveld, Kane Hadley, Matt Dallas, Binh Nguyen, and especially John Behmer for his help and discussions of ideas, and Emma Li Zhang, whom I worked closely with my first year in the lab. Everyone there made it a truly enjoyable and unique experience I won't forget.

Finally, I would like to express my greatest gratitude to my adviser, Jeff Trinkle, who gave me a chance in Computer Science coming from a background in Mechanical Engineering. He has showed both brilliance, teaching me more and more every time we talked, and compassion, for when times were tough. He went out of his way to help me, as he did for all his students. Without his dedication and support, you would not be reading this now.

ABSTRACT

Physical simulation is important for a wide range of problems, particularly so in the field of robotics. The need for faster simulation to provide larger amounts of data is increasingly growing. The trend in computing is growing towards more cores as opposed to faster cores, and the graphical processing unit, or GPU, shows great promise to provide high computational performance. Previously, the dynamics of physical simulation have been solved by using the complementarity formulation. This work explores a different formulation of dynamics using set based force laws, called the proximal point formulation. The formulation of the complementarity based dynamics is reviewed, and this format is used as a base to derive the proximal point formulations, showing equivalence in the process. To test the proximal point's ability to be used in physics simulation, a plugin for dVC2D, a planar physics simulator, is written to implement the proximal point dynamics formulation. In addition, this implementation is also ported to the GPU. The accuracy of these implementations to the complementarity formulation solved with the PATH solver is compared. Finally, the time performance between the implementations and the PATH solver are compared.

CHAPTER 1
INTRODUCTION

1.1 Background

Physics simulation has a wide variety of uses. In robotics in particular, there is a very large potential for physical simulation to advance the capabilities of robotics. Physical simulation allows for better motion planning, by allowing complex robots with many degrees of freedom to better model the consequences of their actions. This is of particular use in robotic grasp planning, where there are many ways to grasp an object, but some work better than others. Being able to accurately predict the consequences of actions is growing increasingly important as robots come into closer and more frequent contact with people, both in industrial settings, and in the push to bring more robotics technology into the home. Robots hitting people, knocking glass onto the floor, breaking furniture, or otherwise causing damage that could harm people and cost money are situations that must be maximally avoided if the dream of widespread robotics is to ever be realized.

In order for physical simulation to be useful, it needs to have two main properties. It must be fast, and it must be accurate. The requirement of accuracy is pretty straightforward, for an inaccurate simulation isn't anything more than random math. How accurate a simulation must be is a more complicated question, that depends largely on the application. In video games, more realistic looking physics is desirable to provide an immersive gaming experience. Technologies such as NVIDIA's PhysX [1] provide very fast physics simulations that are designed to look realistic, but the proximity to actual reality are of no consequence other than potentially the player's enjoyment. Other approaches give higher weight to accuracy, but sacrifice speed as a consequence, such as using the PATH solver to solve complementarity based dynamics [2].

In grasp planning, simulation allows a robot to test many different grasps in a simulated space, and from this simulation space, pick a grasp that best accomplishes the desired goal. One approach is to pre-compute grasps for common objects, such as the Columbia Grasp Database [3]. With this approach, simulation times can be long, but knowledge of what a robot will encounter is needed well in advance, minimizing the robot's ability to adapt to new circumstances.

Additionally, physics simulation can be used to learn about an environment. For example, a robot could try to push an object on a table, recording the results. It could then test combinations of parameters like coefficient of friction to see what yields similar results [4]. The robot can then use this information in the future to determine if an action will be safe. In order to accomplish this type of task, the ability to perform many simulations quickly is critical for it to be useful.

With current trends in computing going toward many cores instead of faster cores, parallelization of computations is needed in order to increase simulation speed without sacrificing accuracy. Computation on the graphical processing unit (GPU) promises to provide greatly increased performance by utilizing the highly parallel nature of the GPU. The GPU has a more specific architecture that is well suited to certain tasks, but not to others. Specifically, the GPU is best suited to applications where the same set of equations or mathematical steps are applied to a large set of data. This case can be seen when applying the dynamics equations to a large set of bodies and contact points. NVIDIA's CUDA [5] has made programming for the GPU substantially easier, allowing normal programming techniques to be used to port code to the GPU.

With this knowledge, this work will explore an implementation of the proximal point method of formulating dynamics, and the potential of these dynamics equations to be ported to the GPU. The proximal point formulation uses convex set theory to formulate the dynamics constraints in terms of sets instead of the more commonly used complementarity approach.

Though physics simulation has many steps, only the dynamics step will be explored in this thesis. Other steps, such as determining what objects have the potential to collide, and updating positions after the dynamics update, will be carried out with the physics simulation program dVC2D [2]. There is potential for optimizing the other steps such as collision detection using particle filtering [6], but they won't be explored here.

1.2 Previous Works

Work on using the complementarity problem for dynamics has been going on for many years. Of particular interest was the development of the dVC2D planar physics simulation program, used in this thesis [2]. dVC2D initially uses the complementarity formulation to solve for dynamics, but it has the ability to support plugins for a variety of other methods. We will be using this plugin capacity to implement our proximal point formulation. In addition to supporting other dynamics solvers, it also supports other time-stepping methods, of which there are quite a variety [7] [8]. To solve the complementarity problem, dVC2D uses the PATH solver [9], but other methods exist as well [10].

Looking at large scale computing, there have been attempts to optimize the computations of the complementarity problem. Tasora et al. [11] [12] implemented a new formulation of the complementarity problem that could be multi-threaded to give increased performance. The GPU has been used in the past to try to solve the complementarity problem, even before technologies like CUDA existed [13]. These used graphics processing to do math, though the process was much harder then. Now formulations of the complementarity problem have been used on the GPU to speed up the dynamics simulation with success [14]. In the work of Tasora et al., they were able to get a performance speed up of approximately one order of magnitude when working with up to 8000 rigid bodies. In addition, their approach solves for normal and frictional forces. One attempt also looked at porting other steps to the GPU [15], though it made some trade offs with versatility. The simulation in [15] also solved the complementarity problem using a Jacobi style solver, which resulted in some limitations in the problem sets that could be solved. The work in this thesis also works with a Jacobi style solver, though ways of mitigating its limitations are explored. In addition, they all completed collision detection on the CPU.

The proximal point formulation has been investigated and formulated in the past [16] [17] [18], though many of these are simply investigations into the mathematics of the proximal point formulation, and how they compare to known systems of dynamics. The proximal point formulation has been used in practice in the specific environment of continuously variable transmissions [19], or CVTs. In this work, the use of sets to formulate the specific dynamics equations showed a good correlation to the measurements available for comparison.

1.3 Outline

This thesis is divided into 5 chapters. The first chapter contains the introduction, previous works, and this outline. The rest of this thesis is organized as follows:

Chapter 2 first covers the basics and notations used in the derivations. Then it looks at the complementarity problem, and the formulation of complementarity based dynamics. From there, the basics of the proximal point are covered. Then, the proximal point dynamics are derived based off of the derivation of the complementarity dynamics. After the proximal point dynamics are derived, the method used to solve them is explored.

Chapter 3 then looks at how the proximal point solver was implemented. First, the basic implementation on the CPU (proxSerial) is covered, then the GPU implementation (proxCUDA) is covered, as well as some of the basic special concerns that GPU programming involves.

Chapter 4 looks at the results of the CPU and GPU implementations of the proximal point dynamics, by comparing the results to the PATH solver complementarity implementation. First the non-friction case is explored, then the frictional implementation. Then the performance of the implementations is examined.

Finally in Chapter 5, lessons learned about the proximal point formulations and implementations are covered in the conclusion section during which potential avenues for future work are explored.

CHAPTER 2
RIGID MULTIBODY DYNAMICS SIMULATION

2.1 Basics and Notation

The dynamics of rigid multibody systems is based on classical Newtonian mechanics. The basic ideas used in this physics simulation will be covered briefly, though a more through overview can be found in Appendix A.

The state of individual bodies can be described using its position ($\vec{q}$), as well as its time derivatives such as position ($\vec{\dot{q}}$). The position of a body has two components, linear and angular. The position of a body typically describes the position about the center of mass of a body. Velocity ($\vec{\nu}$) is the rate that a body's position changes.

For 2D cases, the position and velocity vectors are as follows:

$$\vec{q} = \begin{bmatrix} X \\ Y \\ \theta \end{bmatrix} \tag{2.1}$$

$$\vec{\nu} = \begin{bmatrix} v_X \\ v_Y \\ \omega \end{bmatrix} \tag{2.2}$$

When two bodies come into contact, a normal force (λ_n) exists to keep the bodies from penetrating. The normal force is so called because it is normal to the surface. The normal force only pushes bodies away, and only exists when there is contact between them.

The friction force (λ_f) is a resistance to two bodies sliding against each other. It is tangential to the surface, and like the normal force, also only exists when the two bodies are in contact. When a body is sliding, the force of friction is equal to the normal force times a coefficient of friction (μ). When there is no velocity in the sliding direction, the friction force can be less than that, and resists the body starting to slide.

If a force doesn't pass through the center of mass of an object, it will apply a torque, or moment, to the body. A moment causes a body to rotate, and can be thought of as twisting the body. Most forces applied to a body will also cause a moment. In order to simplify some explanations, we may talk about "particles", which are infinitely small bodies, and therefore can't rotate, so moments can be neglected and only pure translation considered.

2.2 Dynamics Formulations

For the following sections, only the 2D case will be covered. It is worth noting that all of the ideas expand into 3D, but since this thesis only covers 2D problems, only the 2D cases of problems and situations will be covered.

2.2.1 Discrete Time vs Instantaneous Models

Before we look at specific ways of formulating the dynamic models, it is important to look at the 2 ways that we can look at solving the models. Instantaneous time is the most intuitive form, and relies solely on formulating the equations using differential calculus (e.g. velocity is the derivative of position). Discrete time is a little more complicated, but easier to solve for. This involves looking as specific, discrete points in time, and just solving for those points. A superscript l, such as q^l, represents that a quantity is at a given time t_l. q^{l+1} would be at the next time interval. The time between time intervals, also known as the time step is defined as h, and in this thesis, we will only be working with constant time-steps.

The method used to approximate derivatives is:

$$\dot{\vec{q}} = \frac{\vec{q}^{l+1} - \vec{q}^{l}}{h} \tag{2.3}$$

2.2.2 Generalized Matrices

In the following sections, certain matrices will be used to represent various mathematical and dynamical quantities.

2.2.2.1 Basic Matrices

If you have taken a linear algebra course before, the following matrices will probably be familiar, but will be covered anyway to provide a refresher and clarify notation.

Since I is used for the moment of inertia, U will be used for the identity matrix, which is a square matrix with 1s on the main diagonal, and 0s elsewhere.

$$I_n = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} \tag{2.4}$$

2.2.2.2 Mass Matrix

The mass matrix contains all the masses and moments of inertia of the bodies, located along the main diagonal. In a 2D situation (such as the one we are examining in this thesis) $M \in \mathbb{R}^{3nx3n}$, where n is the number of bodies.

$$M = \begin{bmatrix} m_1 U_{2\times 2} & 0 & \cdots & 0 & 0 \\ 0 & I_1 & \cdots & 0 & \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & m_n U_{2\times 2} & 0 \\ 0 & 0 & \cdots & 0 & I_n \end{bmatrix} \tag{2.5}$$

where m_i is the mass of body i and I_i is the moment of inertia of body i.

2.2.2.3 Wrench Matrices

A contact wrench describes how a contact force magnitude will impact a body. Given a unit vector of the contact force direction in the global coordinate frame, $\hat{v} = \left[\begin{smallmatrix} v_x \\ v_y \end{smallmatrix}\right]$, and the distance of the contact location from the center of mass $D = \left[\begin{smallmatrix} D_x \\ D_y \end{smallmatrix}\right]$,

$$W_i = \begin{bmatrix} v_x \\ v_y \\ D \otimes v \end{bmatrix} \tag{2.6}$$

where $\otimes$ is the cross product. A full W matrix is constructed from the individual W_i matrices, where each row is for a body, and each column for a potential contact point. If a contact point does not reference a body, it will be a vector of 3 0s at that location .

2.2.3 Complementarity Formulation

The system of equations and conditions used to solve for the normal and friction forces in dVC2D and other simulators is a system called the complementarity formulation.

2.2.3.1 The Complementarity Condition

The complementarity condition can be stated in a variety of ways, but the simplest case is in the Linear Complementarity Problem (LCP). The LCP can be formulated as such:

Given a matrix $M \in \mathbb{R}^{nxn}$ and a vector $q \in \mathbb{R}^n$, find two vectors, $z \in \mathbb{R}^n$ and $w(z) \in \mathbb{R}^n$ that satisfy the following conditions:

$$w = Mz + q \tag{2.7}$$

$$0 \leq w(z) \perp z \geq 0 \tag{2.8}$$

It is worth reminding the reader here that when equations such as the one above are written for vectors, $w(z)$, z, and q are all lists of many values (for example $z = [z_1\ z_2\ z_3]^T$ is a vector). This is something I often overlooked in the beginning, and it lead me to far more confusion than it should have. Additionally, it is worth noting that $w(z)$ indicates that the value of w is a function of z, or in other words the value of w is dependent on the value of z. If w and z were completely independent, it would be rather trivial to solve this problem.

Satisfying the first condition should be fairly straightforward to anyone who has used matrices in algebra, and therefore won't be given much time here.

The second condition is where the complementarity condition is where the problem becomes unique (and gets its name). The symbol $\perp$ sets $w^T z = 0$. Vectors that satisfy this condition are called "orthogonal" to each other. Put another way, the second condition requires the values of w and z must be greater than zero, and for every value $w_i \geq 0$, $z_i = 0$. The reciprocal is also true; that is for every $z_i \geq 0$, $w_i = 0$.

The second condition could also be written a second way:

$$w \geq 0, z \geq 0 \tag{2.9}$$

$$w_i z_i = 0 \text{ for all } i \tag{2.10}$$

Both forms express the same idea, and have the same requirements, though in the rest of this thesis, only the first form will be used.

2.2.3.2 Mixed Linear Complementarity Problem

The LCP from the previous section can be expanded further, in a form called the Mixed Complementarity Problem, or MCP. The MCP has several differences from the LCP. First, the "Mixed" refers to the fact that in addition to the conditions of the LCP, an MCP can also contain other equations that must be satisfied. These additional equations are usually set to equal zero by convention, but not required.

The other difference between the LCP and the MCP is that the MCP does not require the relationship between w and z to be linear in nature. Put differently $w(z)$ can be a nonlinear function.

2.2.4 Complementarity Based Dynamics

In forming the complementarity based dynamics, the MCP form of the complementarity problem will be used. Additionally a full and exhaustive explanation of complementarity based dynamics is beyond the scope of this thesis, but having and understanding of how they work is important for understanding the proximal point formulation, for the proximal point form is based off of the complementarity form. More information on complementarity based dynamics can be found in prior works [20] [2] [21].

Below will be a simple derivation of the complementarity forms of normal contact and frictional forces in the 2D case.

2.2.4.1 Normal Contact

The complementarity condition is used in several ways in dynamics formulation. In the case of normal contact, it can be logically seen as representing the fact that either there is a positive distance between two bodies, OR there is a normal force between the bodies; there can not be both.

Let $\vec{\lambda_n}$ be the vector of all the forces normal to their respective contact planes (one force for each potential contact point) otherwise known as the normal forces, and $\vec{\Psi_n}(\vec{q}, t)$ be a vector of the gap functions. Each potential contact point has a unit vector in the normal direction denoted $\hat{n_i}$ for the ith contact. The gap function is simply the distance of a potential contact point from penetrating a body, where a positive value is no contact, a negative value is penetration, and a 0 value is contact.

Calculating the gap function is simply a matter of geometry, though there are some techniques that are better than others. Exploring those techniques are beyond the scope of this thesis. Additionally, the gap function is calculated in dVC [2] and the gap distance is provided.

With the gap function provided, the normal contact constraint can be written as such:

$$0 \leq \vec{\lambda}_n \perp \vec{\Psi}_n \geq 0 \tag{2.11}$$

2.2.4.2 Friction

In the case of frictional forces, otherwise known as tangential contact constraints, the use of the complementarity problem is less intuitively obvious. The unit vector for the tangential friction force is denoted $\hat{t_i}$ for the ith contact, and is perpendicular to n_i. As noted earlier, the vectors in the complementarity condition must be non-negative. Since friction can be in the positive or negative directions, the frictional force is broken into two values at each contact point, one in the positive direction $(\hat{t_i})$, and one in the negative $(-\hat{t_i})$. Let λ_{fi} be the value of the friction force along the tangential direction, and λ_{fi}^* be the friction force formulated for the complementarity problem.

$$\lambda^*_{fi} = \begin{bmatrix} \lambda_{fi1} \\ \lambda_{fi2} \end{bmatrix} \tag{2.12}$$

$$\lambda_{fi} = \lambda_{fi1} - \lambda_{fi2} \tag{2.13}$$

It is worth reminding that λ_{fi1} and λ_{fi2} are both positive in value, and their vectors are opposite in direction.

In addition to the above requirement, the frictional force has two other requirements. If the body is sliding, then the magnitude of the friction force is equal to $\mu\lambda_n$. If the body is not sliding, then the friction force is between $-\mu\lambda_n$ and $\mu\lambda_n$. Since λ_n is present in these requirements, a third requirement that is there is no friction force unless there is a normal force is therefore implied.

In order to accomplish the above requirements, a variable σ is used which approximates the sliding speed at the contact point (since it is speed, and not velocity, it will always be non-negative). With this extra variable, the complementarity form for friction between a particle on a fixed surface can be written as such:

$$0 \leq \lambda_{fi1} \perp \hat{t}\vec{\nu}_i + \sigma_i \geq 0 \tag{2.14}$$

$$0 \leq \lambda_{fi2} \perp -\hat{t}\vec{\nu}_i + \sigma_i \geq 0 \tag{2.15}$$

$$0 \leq \vec{\sigma}_i \perp \lambda_{ni} - \lambda_{fi1} - \lambda_{fi2} \geq 0 \tag{2.16}$$

Using a particle allows us to neglect moments, simplifying the equations to help get the idea across. The full form allowing multiple bodies and many contacts can be written as such:

$$0 \leq \vec{\lambda}^*_f \perp W_f^T \vec{\nu} + \vec{\sigma} \geq 0 \tag{2.17}$$

$$0 \leq \vec{\sigma} \perp \vec{\lambda}_n - E^T \vec{\lambda}_f \geq 0 \tag{2.18}$$

In the above equation, E is a block diagonal matrix where each block i on the main diagonal is a vector of 1s of size 2, since there are 2 directions of friction that need to be added for each contact point.

$$E = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \\ 0 & 0 & \cdots & 1 \end{bmatrix} \tag{2.19}$$

This form, though more useful in practice, hides the increased complexity in the λ_f^* vector and the W_f matrix. If the friction is between two bodies, then both bodies would be represented in the wrench matrix (W_f), since its the relative velocity of the contact point that matters, not the velocity in the global frame. If you would like a more complete explanation, then please see [20] [22] [23]

2.2.4.3 Instantaneous Form

In addition to the above equations, there is the additional constraint that $\vec{f} = m\vec{a}$ and $\vec{\tau} = I\vec{\dot{\omega}}$. This requirement, also known as the Newton-Euler equations, is what allows the forces applied above to achieve their goals of reducing sliding and keeping bodies from penetrating. This is achieved by requiring that the mass of a body times its acceleration is equal to all the forces being applied to it at any given point in time. Knowing that the forces present in our system are the normal forces, frictional forces, and external forces (such as gravity), we can write this requirement as:

$$M\vec{\dot{\nu}} = W_n(\vec{q}, t)\lambda_n + W_f(\vec{q}, t)\lambda_f + \lambda_{ext}(t) \tag{2.20}$$

The final set of equations for all of our contact dynamics is:

$$M\vec{\dot{\nu}} = W_n(\vec{q},t)\lambda_n + W_f(\vec{q},t)\lambda_f + \lambda_{ext}(t) \tag{2.21}$$

$$0 \le \vec{\lambda}_n \perp \vec{\Psi}_n \ge 0 \tag{2.22}$$

$$0 \le \vec{\lambda}_f^* \perp W_f^T\vec{\nu} + \vec{\sigma} \ge 0 \tag{2.23}$$

$$0 \le \vec{\sigma} \perp \vec{\lambda}_n - E^T\vec{\lambda}_f \ge 0 \tag{2.24}$$

2.2.4.4 Discrete Form

In order to take the instantaneous forces and spread them over a time step, we need to take our forces and convert them to impulses, which is just a force applied for an amount of time. First we can update our Newton-Euler equation to use impulses instead of forces:

$$M(\vec{\nu}^{l+1} - \vec{\nu}^l) = W_n p_n^{l+1} + W_f p_f^{l+1} + p_{ext}^{l+1} \tag{2.25}$$

$$M\vec{\nu}^{l+1} = M\vec{\nu}^l + W_n p_n^{l+1} + W_f p_f^{l+1} + p_{ext}^{l+1} \tag{2.26}$$

The impulse applied to a body is equal to the change in velocity of that body which is where the first equation comes from, but the second form is a more useful form when it comes to solving for the unknown values in the next time step (at t_{l+1}). We also need to know how to update the position of the body between time-steps:

$$\vec{q}^{l+1} = \vec{q}^l + h\vec{\nu}^{l+1} \tag{2.27}$$

We use ν^{l+1} and not ν^l because ν^l is already known, and no amount of force in the future can change that. By using ν^{l+1}, our impulses at t_{l+1} can change the velocity, and therefore dynamics of the system.

For the normal force, we need to replace our instantaneous gap function with a new equation:

$$0 \le \vec{p}_n^{l+1} \perp \frac{\vec{\Psi}_n^l}{h} + W_n^T\vec{\nu}^{l+1} \ge 0 \tag{2.28}$$

In this equation, we use $\frac{\vec{\Psi}_n^l}{h}$ to determine how fast we can travel in the time between t_l and t_{l+1} and not penetrate. We then use $W_n^T\vec{\nu}^{l+1}$ to determine how fast we will be traveling.

The changes for the frictional force are much simpler, and simply changing all the forces to impulses is sufficient to convert the equations to the discrete form.

$$0 \leq \vec{p}_f^{*l+1} \perp W_f^T \vec{\nu}^{l+1} + \vec{\sigma}^{l+1} \geq 0 \tag{2.29}$$

$$0 \leq \vec{\sigma}^{l+1} \perp \vec{p}_n^{l+1} - E^T \vec{p}_f^{l+1} \geq 0 \tag{2.30}$$

Though not noted in the above equations, the wrench matrices are calculated at the beginning of a time step, and are constant throughout that time step. To help clarify, here are all the equations used in formulating the discrete form:

$$M\vec{\nu}^{l+1} = M\vec{\nu}^l + W_n p_n^{l+1} + W_f p_f^{l+1} + p_{ext}^{l+1} \tag{2.31}$$

$$\vec{q}^{l+1} = \vec{q}^l + h\vec{\nu}^{l+1} \tag{2.32}$$

$$0 \leq \vec{p}_n^{l+1} \perp \frac{\vec{\Psi}_n^l}{h} + W_n^T \vec{\nu}^{l+1} \geq 0 \tag{2.33}$$

$$0 \leq \vec{p}_f^{*l+1} \perp W_f^T \vec{\nu}^{l+1} + \vec{\sigma}^{l+1} \geq 0 \tag{2.34}$$

$$0 \leq \vec{\sigma}^{l+1} \perp \vec{p}_n^{l+1} - E^T \vec{p}_f^{l+1} \geq 0 \tag{2.35}$$

2.2.4.5 Matrix Form

Equations 2.31 to 2.35 can be placed into matrix form, which is the format used by most solving algorithms. The right side of 2.33 is referred to as $\vec{\rho}_n^{l+1}$, the right side of 2.34 is referred to as $\vec{\rho}_f^{l+1}$, and the right side of 2.35 is referred to as $\vec{s}^{l+1}$. In addition, some rearranging of terms is needed.

$$\begin{bmatrix} 0 \\ \vec{\rho}_n^{l+1} \\ \vec{\rho}_f^{l+1} \\ \vec{s}^{l+1} \end{bmatrix} = \begin{bmatrix} -M & W_n & W_f & 0 \\ W_n^T & 0 & 0 & 0 \\ W_f^T & 0 & 0 & E \\ 0 & U & -E^T & 0 \end{bmatrix} \begin{bmatrix} \vec{\nu}^{l+1} \\ \vec{p}_n^{l+1} \\ \vec{p}_f^{l+1} \\ \vec{\sigma}^{l+1} \end{bmatrix} + \begin{bmatrix} M\vec{\nu}^l + \vec{p}_{ext} \\ \frac{\vec{\Psi}_n^l}{h} \\ 0 \\ 0 \end{bmatrix} \tag{2.36}$$

2.2.5 Proximal Point Formulation

The proximal point formulation takes a different approach to looking at the contact laws. Instead of complementarity, it considers the contact forces in terms of set theory, and in particular, convex set theory. A convex set is defined as a set where if any two points in the set were connected by a line, all points along the line will also be in the set. In this thesis, we will only be dealing with 1D sets, so the definition can be simplified a little more; a 1D set is convex if it consists of only a single range of numbers.

Next we need to define what the proximal point to a convex set is. Given a convex set $C \subset \mathbb{R}^n$ and $n \in \mathbb{R}$ (where n is the number of dimensions of the set):

$$\text{prox}_C(x) = \underset{x^* \in C}{\text{argmin}} ||x - x^*||, \ x \in \mathbb{R}^n \tag{2.37}$$

To clarify, there are essentially two possible situations that you can encounter when evaluating equation 2.37. The first is that the point x is located in the set C ($x \in C$). In this case, $\text{prox}_C(x) = x$. As an example, if $C = \{1 \leq x \leq 5\}$, then $\text{prox}_C(2) = 2$ since 2 is located in C.

The second possible situation is that x is not in the set C ($x \notin C$). In this case, $\text{prox}_C(x)$ would be equal to the closest point to x that is in the set C. Using our example C from above, $C = \{1 \leq x \leq 5\}$, $\text{prox}_C(7) = 5$, since 7 is not in C, and 5 is the closest number to 7 that is.

2.2.5.1 Set Valued Normal Contact

In order to use the proximal point function, we need to rewrite our dynamics constraints. The set of all possible normal forces (C_n) is pretty straightforward:

$$C_n = \{\lambda_n \in \mathbb{R} | \lambda_n \geq 0\} \tag{2.38}$$

In a more verbal form, the set C_n consists of all non-negative real numbers. Next, we need to formulate an equation that will give us the same properties as the complementarity formulation, namely that the normal force prevents penetration of the bodies, and that the normal force only exists if the gap distance is non-positive.

$$\lambda_n - \text{prox}_{C_n}(\lambda_n - r\Psi_n) = 0 \tag{2.39}$$

There are several important points to how this equation works. First is that λ_n minus prox_{C_n} equals 0 guarantees that λ_n will be within C_n, and therefore a valid value.

Secondly, r is called the "independent auxiliary parameter" and controls how quickly the solver will converge on a solution, or if a solution can be converged upon. It is also called the "r-value" in this thesis. How the r-value is determined will be discussed later, but it will always be a positive value ($r \in \mathbb{R}^+$), and is unique for each normal contact.

Additionally, if there is a distance between the bodies ($\Psi_n \geq 0$), then λ_n will be 0, as we would expect. If ($\Psi_n < 0$), then the equation will never be valid. I encourage the reader to try some values in the equations to see this for themselves if this conclusion doesn't seem intuitive. In the instantaneous model, this means that a value of λ_n will be found to prevent penetration. How to solve this problem in the discrete model will be discussed later.

2.2.5.2 Set Valued Friction

The set of all possible friction forces is also fairly straightforward:

$$C_f(\lambda_n) = \{\lambda_f \in \mathbb{R} | -\mu\lambda_n \leq \lambda_f \leq \mu\lambda_n\} \tag{2.40}$$

The set C_f is slightly more complicated than C_n was. For starters, since the limits of friction are a function of the normal force, what C_f contains is also a function of λ_n. Next we need an equation that applies the full $\pm\mu\lambda_n$ if there is relative sliding between the bodies, or applies a friction force in such a way as to keep the sliding velocity to zero.

$$\lambda_f - \text{prox}_{C_f(C_n)}(\lambda_f - rW_f^T\vec{\nu}) = 0 \tag{2.41}$$

This form is very similar to the normal contact form in equation 2.39, with only a few differences. Since the friction force is dependent on sliding velocity, the gap distance is replaced with the sliding velocity. The r-value is the same in principle as the normal contact, but their values can (and probably will be) different.

To show that our equation yields the correct results, we can look at the two possible cases of sliding, or not sliding. In the case of not sliding ($W_f^T \vec{\nu} = 0$) and $\lambda_f \in C_f$, the equation will be valid. If $\lambda_f \notin C_f$, then the situation will probably degenerate into the case of having sliding.

If there is sliding at the contact ($W_f^T \vec{\nu} \neq 0$), then λ_f will be on the edge of C_f, which is $\pm\mu\lambda_n$. If λ_f were within the bounds, then $\text{prox}_{C_f(C_n)} \neq \lambda_f$, and therefore the equation could not be satisfied, forcing our constraints to hold.

2.2.5.3 Discrete Time Proximal Point

In addition to the above proximal point functions, the Newton-Euler equations listed in the complementarity sections as equations 2.31 and 2.32 also are needed to fully formulate the dynamics of the system.

The derivations to switch from instantaneous form to discrete form are the same as those in section 2.2.4.4 (Discrete Form). We will be using those variables to formulate the discrete time proximal point functions. The proximal point functions translate well from the complementarity form, with the only real difference being how the equations are solved. By switching from forces to impulses and moving the prox functions to the right, we can arrive at the following equations:

$$M\vec{\nu}^{l+1} = M\vec{\nu}^{l} + W_n p_n^{l+1} + W_f p_f^{l+1} + p_{ext}^{l+1} \tag{2.42}$$

$$\vec{q}^{l+1} = \vec{q}^{l} + h\vec{\nu}^{l+1} \tag{2.43}$$

$$\rho_n^{l+1} = \frac{\vec{\Psi}^l_n}{h} + W_n^T \vec{\nu}^{l+1} \tag{2.44}$$

$$p_n^{l+1} = \text{prox}_{C_n}(p_n^{l+1} - r\rho_n^{l+1}) \tag{2.45}$$

$$\rho_f^{l+1} = W_f^T \vec{\nu}^{l+1} \tag{2.46}$$

$$p_f^{l+1} = \text{prox}_{C_f(p_n^{l+1})}(p_f^{l+1} - r\rho_f^{l+1}) \tag{2.47}$$

2.3 Solving Proximal Point Problems

2.3.1 Fixed Point Iteration

The proximal point equations are solved through a process called fixed point iteration. Defining k as the current iteration, the basic fixed point iteration equation looks like:

$$x_{k+1} = f(x_k),\ k = 0, 1, 2 \ldots \tag{2.48}$$

where for each x_k iterated, the system comes closer to converging on a solution. The above equations follow this form, and it is worth noting that ρ_n and ρ_f both also are functions of their respective impulses, through the ν that is dependent on impulses. Rewriting the above equations for impulse (equations 2.45 and 2.47) to use fixed point iteration, they would look like:

$$p_n^{k+1} = \text{prox}_{C_n}(p_n^k - r\rho_n^k) \tag{2.49}$$

$$p_f^{k+1} = \text{prox}_{C_f(p_n^{k+1})}(p_f^k - r\rho_f^k) \tag{2.50}$$

where all the values are at $l + 1$ (note that the superscripts are now in terms of k).

There are situations that could cause the system to diverge away from a solution instead of converging on it. It is for this reason that the choice of the independent auxiliary parameter (r-value) is important. In particular, as r approaches 0, it will take an infinite amount of time to converge on a solution, and as r gets large, the details of the system behavior get concealed and cause the solution to diverge away.

2.3.2 Independent Auxiliary Parameter (r-value)

In this thesis, we tested two ways of choosing the r-value, the Relaxed Richardson and the Relaxed Jacobi. In addition, there is a third method, the Relaxed Gauss-Seidel, that shows promise for alleviating the problems of the first two. All of these methods are called "relaxed" because of a relaxation parameter, noted as ω. This parameter acts as a divisor to modify the r-value calculated by a constant factor across all contacts, its main use being to reduce the r-value to increase the systems ability to converge on a solution.

The Delassus Matrix [24] is used in the Jacobi and Gauss-Seidel schemes. The Delassus Matrix is defined as:

$$G = W^T M^{-1} W \tag{2.51}$$

where W is a wrench matrix. In the implementation of the proximal point equations, a different Delassus Matrix is built for the normal impulses and the frictional impulses. The two matrices could probably be combined into a single matrix, with possibly better results. The forms for the following implementations treat r as a vector, with one element for each contact. Since there are two Delassus Matrices in this implementation, there would also be two r vectors, one for each contact force.

2.3.2.1 Relaxed Richardson

The Relaxed Richardson is the simplest method to implement r, with r simply being equal to ω, or more formally put:

$$r = \omega E \tag{2.52}$$

where E is a vector of ones, with its length equal to the number of contact points.

Though the Relaxed Richardson is clearly the simplest system for calculating the r-value, it also gives the worst performance, especially when largely different masses are in the simulation, since the mass of a body affects the size of the impulse needed to move it a certain distance.

2.3.2.2 Relaxed Jacobi

The Relaxed Jacobi scheme is the second method implemented, and uses the Delassus Matrix (equation 2.51). It is defined as:

$$r = \left(\frac{\mathrm{diag}(G)}{\omega}\right)^{-1} \tag{2.53}$$

where $\mathrm{diag}(G)$ defines a matrix the same size as G where the only non-zero elements are on the diagonal; in other words the diagonal of G.

What this yields in practice is that the mass and moment of inertia of the bodies involved in a contact point are used to scale the r-value, and thus, the impulse applied. The Jacobi method in linear algebra is guaranteed to converge if the matrix is strictly diagonally dominant, and without the relaxation parameter ($\omega = 1$), the same would apply here. The reason being if multiple contacts have similar wrench matrices, their reaction forces will add, and can overreact, causing divergence.

2.3.2.3 Relaxed Gauss-Seidel

Though the Relaxed Gauss-Seidel was not implemented in this thesis, a brief mention is still warranted, since it holds promise for alleviating some of the limitations of the Relaxed Jacobi. In linear algebra, the Gauss-Seidel method is guaranteed to converge if the matrix is symmetric or positive definite, in addition to the diagonal dominance that will allow Jacobi to converge. The relaxed Gauss-Seidel can be defined as:

$$r = \left(\frac{\mathrm{diag}(G)}{\omega} + \mathrm{tril}(G) \right)^{-1} \tag{2.54}$$

where $\mathrm{tril}(G)$ is the same size as G but the only non-zero elements are the lower triangular, excluding the main diagonal.

What including the lower triangular components yields are r-values that are scaled not only based on the mass and moment of inertia of the bodies involved in the contact, but also on what other contacts will affect the bodies involved in a contact. This allows higher r-values for bodies with few contacts, and lower ones for bodies with many, to increase stability only where needed. Computing the lower triangular component for a very large matrix can be computationally much harder when many contacts, due to polynomial growth, instead of linear, as in the Relaxed Jacobi.

CHAPTER 3
SOLVER IMPLEMENTATION

3.1 Implementation Basics

The implementation of the prox solver is broken up into three basics steps. First, the data structures are built to pass the information needed to solve the proximal point equations. Then, through the fixed point iteration, the proximal point equations are solved for. Then, when the proximal point equations are solved, the final body velocities are passed back to dVC2D in order to update the state of the system for the next time step. The following sections will describe these steps in more detail, first describing the CPU implementation. With the CPU implementation described, major differences between the CPU and GPU implementation will be described, including best practices followed for GPU programming, and CUDA programming in particular. There are a few components in the code that were used in testing, but were not used in the tests documented in this thesis. In order to keep the following explanations as straightforward as possible, they will not be included in this explanation. Specifics of the data structures can be found in the code in Appendix B.

In addition, dVC2D allows configuration settings to be passed from XML files created for each scene. The proximal point implementations use this config file to determine various things like which method should be used to calculate the r-value, the relaxation coefficient, maximum number of iterations, and sizes of data structures.

3.2 CPU (ProxSerial)

3.2.1 Build Data Structures

Due to the implementation of dVC2D, the data structures used to hold the starting conditions for a time step are in a format best for solving complementarity problems. In order to simplify programming, and in the case of the GPU speed up memory transfers, new data structures are built.

Two primary types of data structures are used. First there is an array of a struct for bodies. There is one struct per body that may be involved in a contact. In this struct, the bodies velocity at time l, mass, moment of inertia, any external impulses (such as gravity), and the velocity at time $l+1$ are stored.

Second there are structs for the friction and normal impulse calculations. There arrays for each, with one element in the array for each potential contact point. Both arrays store the wrench matrix for that impulse on each body (noted as G in the code instead of W due to a different notation style referenced while writing the code). In addition, each struct also contains the r-value for that contact and impulse, the impulse calculated for the current iteration, a boolean value for whether the impulse has converged on a solution, and the index value of where the two potential bodies are stored in the array of bodies. In addition to the above, the struct for the normal impulses also contains the gap at time step l.

The size of these data structures are determined from the config file, where maximum sizes are set. Reallocating the data structures for each time step can create problems, especially when the data structures can be fairly large.

3.2.2 Solve Proximal Point Dynamics

To solve the proximal point dynamics, first the r-values are calculated using the information at the beginning of the time step. Then, the impulses and dynamics updates are solved iteratively until a solution is converged on or a preset number of iterations have passed. More detail on these steps are below.

3.2.2.1 Calculate r-value

To calculate the r-value, which method (Richardson or Jacobi) is determined from the config file for the simulation. Regardless of the method used, each uses a `for` loop to iterate through all the contacts being used, and assign the r-value to each normal contact. If friction is in use, then all the contacts will be iterated through again to assign the r-value to the friction contact struct. The math for calculating the r-value can be found in section 2.3.2, or in the source code. In the Jacobi formulation, instead of building the matrix and calculating the diagonal, an equation for the diagonal was constructed.

3.2.2.2 Solve Normal Impulse

To solve the normal impulse, a `for` loop iterates over all of the contacts. First, ρ_n is calculated using equation 2.44. Then ρ_n is compared to a threshold for convergence (during testing it was set to -1×10^{-6}). If ρ_n is greater than that value, then it is considered to be non-penetrating, and a boolean value indicating this value has converged is set to true for use later. Next, the values inside the proximal point function are calculated ($p_n^k - r\rho_n^k$), and are then compared to C_n, and is noted p_n^*. If the value is less than 0, then p_n^{k+1} is set to 0, otherwise, it is set to the value calculated earlier.

An extra function that can be enabled in the compiler flags is a method that was tested to detect system divergence. If an impulse exceeds a preset limit, a flag is set, and the system reduces the relaxation coefficient and restarts the time step from the beginning. In addition, the maximum number of iterations is doubled to compensate for the increased time that will be needed to reach a solution. The idea behind this is that if a value is diverging, the impulses will get incredibly high, at times approaching infinity. Additional systems were tested to try to detect divergence, but were not as effective. Further information on attempts to fix divergence are in section 4.4.1.

3.2.2.3 Solve Friction Impulse

The friction impulse is solved in the same way as the normal impulse, with a few exceptions. First, if the normal impulse for that contact is 0, then the friction impulse is set to 0 and is finished, since friction can't exist without a normal force. Next, the equation for ρ_f is different than ρ_n, so equation 2.46 is used instead. After ρ_f is calculated, it's absolute value is compared to a threshold for convergence (during testing it was set to 1×10^{-6}). The absolute value is taken because friction can be positive or negative, and the threshold is set to a value greater than zero to take into account small numerical errors. Then the value inside the proximal point function is calculated and compared to C_f, and is noted p_f^*. In code this consists of checking if $p_f^* < -\mu p_n$. If it is, it is set as equal to $-\mu p_n$. Otherwise, a check is made, $p_f^* > \mu p_n$, which if true then $p_f^* = \mu p_n$ is set. These take care of the proximal point function. From there, the value is saved, and the next contact is checked in the `for` loop.

3.2.2.4 Update Body Dynamics

Once all the impulses are solved for, then the dynamics need to be updated. This is essentially an implementation of the Newton-Euler equation, noted in equation 2.42. Each body has a value of the total impulse applied to it over the time step by all contacts (separated into X,Y and rotation impulse). To solve the Newton-Euler equation, first a `for` loop iterates over all of the contacts. The impulse calculated for the normal impulse is added to a cumulative value for the bodies involved in the contact[1]. If there is friction, then the same step is performed for the frictional impulse for the contact.

After all the impulses are summed up for every body, another `for` loop is used to solve the Newton-Euler equations. For each body, the velocity is set as:

$$\vec{\nu}^{l+1} = \vec{\nu}^{l} + M^{-1}(\vec{p}_{ext}^{\,l+1} + \vec{p}_{total}) \tag{3.1}$$

where the vectors and mass matrix are for the single body.

3.2.2.5 Check for Convergence

After the above steps are done, the boolean variables for all the contacts indicating if that particular contact point converged are checked in a `for` loop. If any of them are false (that contact hasn't met the criteria for convergence), then it returns false and another iteration is run. If every contact returns true for being converged, then the loop to solve the proximal point dynamics ends.

3.2.3 Return Velocities

The last step when the dynamics equations are fully solved, and the body velocities have been updated with the final impulse values, is to pass the final body velocities to the simulator. With the velocities, the simulator can calculate the state

[1]More specifically, the impulse multiplied by the wrench matrix for that impulse (which results in a vector) is added to the vector of impulse for the body

of the bodies for the next time step.

3.3 GPU Programming

Understanding GPU programming requires two basic pieces of information. The first is the architecture of the GPU, and in particular how it differs from the CPU architecture, to help in understanding how to modify programs from conventional programming paradigms. Second are particular techniques for optimizing GPU code that give the greatest benefit for programs written for the GPU. Both are covered in the following sections.

3.3.1 Basic GPU Architecture

The architecture of the GPU is important because it governs how you program for it. By understanding how it preforms, you can play to its strengths, and try to mitigate its weakness. Since the GPU architecture is substantially different from the standard CPU, this is probably the hardest part of programming for the GPU. Since various GPUs are different, this thesis will focus on NVIDIA's CUDA and in particular, their Fermi architecture. There currently isn't much difference between CUDA architectures at the moment, but its hard to tell what the future will bring.

At the lowest level in the GPU is a thread, which executes a sequence of instructions, very much like a CPU thread would. Each thread is capable of knowing its identity (or ID number). Threads are then grouped into sets of 32 called warps. These warps all execute the same instruction, just on different memory addresses. If there are conditional branches (like `if` statements), then some of the threads will sit idle. Each GPU card has several multiprocessors (for example, the card used in this thesis has 15. Multiprocessors run many warps, swapping them out when appropriate, similar to how a CPU swaps out threads. Multiprocessors also have a group of shared memory that all the warps on that multiprocessor share, on the order of approximately 65,000kB. It is the limit of this memory that mainly limits how many warps a multiprocessor can run at a time. All the multiprocessors also share a global memory, and this is where memory is copied to from the host (CPU) and is usually around 1GB. In addition the program that the GPU runs is called a kernel, and current CUDA enabled cards allow each multiprocessor to run a different kernel. Global memory is much slower than the multiprocessors local memory, but that latency can be helped in part by a cache shared by all the multiprocessors.

3.3.2 GPU Optimization

When programing for CUDA and the GPU, there are three primary properties you want to optimize [25]. First, you want to maximize utilization. There are several ways to accomplish this. First, whenever possible, you want the CPU to execute code while the GPU is busy, maximizing your use of both resources. Additionally, you want to keep the GPU running code even during memory transfers if possible. There is also the ability to run multiple kernels on the GPU at the same time, though we found no way to utilize this ability.

Next you want to maximize memory throughput. Memory latency for global memory access is extremely high, as is memory latency from the hosts main memory to the GPU memory. In order to keep the transfer of memory from the host to the GPU to a minimum, you can run calculations that are not highly parallel to build intermediary data structures. Additionally, when the device accesses its global memory, there are special techniques that can be used to speed up this access. Implementing these procedures can be complicated and time consuming, so they were not implemented here. Finally, by having many more threads executing on the GPU than there exists physical threads in the GPU, you can hide memory access latencies. This is achieved by switching out waiting threads with threads that have their memory already fetched and waiting.

Finally, you want to maximize instruction throughput. The main way to do this is to minimize divergent branching. This is when a conditional statement, causes some threads in a warp to take one path, while the other threads do another. Since all the threads of a warp must do the same instruction or none at all, this can vastly slow down execution.

3.4 GPU (ProxCUDA)

3.4.1 Build Data Structures

The data structures for proxCUDA contain the same information as proxSerial, but are broken up into more separate components. This allows only the specific data that needs to be transferred to do so. In addition, much of the data to be transferred to the device is stored in memory allocated for “write combining”, which allows the memory writes to be done in bursts, but doesn’t guarantee correct ordering if the CPU tried to read it. Since it is acting as a one way buffer, this is fine and worth the speed gain. As some data structures are finished being built, they are copied to the GPU to maximize parallel operations of the system. All of the building of the data structures for the GPU are done on the CPU, since that is where the data currently resides.

3.4.2 Solve Proximal Point Dynamics

The biggest difference between the CPU and GPU implementations is that many operations that were handled in `for` loops in proxSerial are now handled by individual threads in the GPU. The sections of code that were in `for` loops ported very well to proxCUDA, since the `for` loops executed the same code on each iteration. The GPU allows all the iterations of `for` loops to be run in parallel, since there is little conditional logic in the `for` loops used here.

In the proxCUDA implementation, the calculation of the r-value, solving the normal and frictional impulses, and updating the body dynamics are all handled in separate GPU kernels. First the r-value is calculated and the body dynamics are updated for the $l + 1$ time step. Then a while loop is executed that runs till convergence is detected or the maximum number of iterations are run. Each loop is one iteration of the fixed point iteration. In the while loop, first the normal impulse and the friction impulse are solved for. Then the convergence data is copied to the CPU, during which the body dynamics are updated. When the convergence data is finished copying, the check for convergence is then run on the CPU, in parallel with the update body dynamics on the GPU. If the convergence check passes, the loop ends.

3.4.2.1 Calculate r-value

The calculation of the r-value is almost a direct port, where the CPU host code determines what method to use (Richardson or Jacobi), and calls a GPU kernel which solves for the r-values of that method. Each contact point is given a thread, so the thread id is the same as the `for` loops iteration number.

3.4.2.2 Solve Normal Impulse

Solving the normal impulse was also a very direct port, replacing the `for` loop with individual threads. The kernel to solve the normal impulse is called, and the steps to solve are similar to those used in proxSerial. The main difference between the implementation between the two is that the GPU code does not contain any of the experimental divergence detection that the CPU code contains, since adding variables to be communicated between the GPU and CPU is more complicated, and proxSerial should provide an equally valid test platform.

3.4.2.3 Solve Friction Impulse

The friction impulse kernel is only called if there is a non-zero coefficient of friction, otherwise this step is skipped. If there is a non-zero coefficient of friction, then the friction impulse kernel is called, and solved in the same manner as the proxSerial function to solve for the friction impulses.

3.4.2.4 Update Body Dynamics

The kernel to update the body dynamics was the least amenable to modification to parallelization. As implemented, each body is given a thread. It was done this way since each body's velocity needs to maintain the correct ordering for reading and writing, and synchronization across threads and multiprocessors is very inefficient. From there, all the threads then use a `for` loop to iterate through all the contacts to check if a contact includes that body. If it does, then the impulses are added to that bodies total impulse value. Though this involves conditional branching, most of the time all the threads of a warp will not be involved in that contact, and therefore not branch. After all the contacts are iterated through, then all the threads update the dynamics for their respective bodies using the Newton-Euler equations, in the same manner as proxSerial.

3.4.2.5 Check for Convergence

After the data structures with the convergence booleans are copied from the GPU to the CPU, the checks are done in the same ways as proxSerial. As stated earlier, the CPU is able to run the convergence check while the GPU is updating the body dynamics. The CPU loops through all the contacts, checking if they have set the converged boolean to false. If it has found any that are false, it returns false for the system being converged, and waits for the body dynamics to update to finish if it hasn't yet.

3.4.3 Return Velocities

After the impulses have reached convergence only the final velocities of the bodies are copied back to the CPU. From there, the body velocities are passed to the simulator so that the body states may be updated, and the next time step started. The interface to get the body velocities was abstracted so that the time-stepper doesn't need to know which prox solver was used.

CHAPTER 4
RESULTS

4.1 Test System

The following tables (tables 4.1 and 4.2) are the main specifications of the hardware and software of the test system that these results were compiled on.

4.2 Test Cases

In order to explore the performance of this implementation of the proximal point formulation, several test cases have been designed. These test cases were then run with a variety of modifications to explore the performance. To keep comparisons consistent, two shapes were used for the below tests.

Table 4.1: Hardware specifications of test computer

Component	Specification
CPU	Intel Core i7 930 @ 2.8GHz 64bit
Memory	6GB Triple Channel
GPU	NVIDIA GTX 480 1.5GB @ 1.45GHz

Table 4.2: Software specifications of test computer

Software	Version
Operating System	Ubuntu 10.04 64bit
Compiler	GCC Version 4.4.3
CUDA Toolkit	Version 4.0

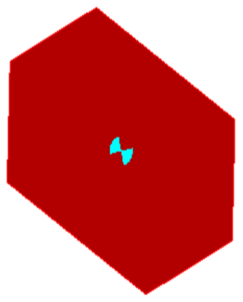


Figure 4.1: The Hexagon, the first test body

4.2.1 Shapes

4.2.1.1 Hexagon

The first shape used was a hexagon, and was used in most of the experiments. An example of the hexagon can be seen in figure 4.1. Worth noting is that the hexagon is not completely symmetric, with 2 of the sides longer than the others. The dimensions of the hexagon are 10 cm $\times$ 16 cm, with a mass of 1 kg and a moment of inertia of 30 kg cm^2. Its center of mass is located centrally, and denoted by the symbol located there.

4.2.1.2 Regular Octagon

Additionally, a second shape, a regular octagon, was used in some tests to help show behavior across multiple body types. The fact that it is regular indicates that all the sides are of equal length. An example of the octagon can be seen in figure 4.2. The mass of the octagon was set to 1 kg and a moment of inertia of 30 kg cm^2.

4.2.2 Friction Angle

The first test case consists of 25 hexagons lined up, set to fall on a fixed platform. The platform is slanted at about 5 degrees (0.09 radians), and a second test is run with the platform at about 10 degrees (0.18 radians). By setting different coefficients of friction, various circumstances can be explored. The friction angle test initial conditions are shown in figure 4.3.

The bodies don't all hit the platform instantaneously, separating what state

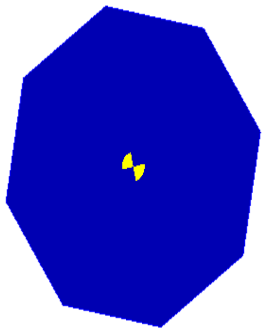


Figure 4.2: The Regular Octagon, the second test body

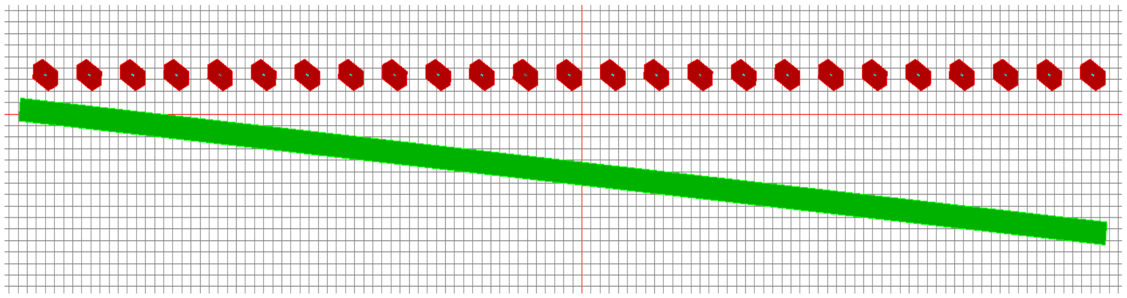


Figure 4.3: The basic test environment for friction, with 25 bodies falling on an angled surface

certain bodies are in at a given point in time, better representing a more mixed scenario, rather than an artificial, homogeneous environment. Depending on the coefficient of friction, the bodies may come to a rest, or they may slide off of the end.

4.2.3 Large Group

The next test is to check how the prox solvers compare to PATH in terms of speed, and to help make the comparisons clear, large test sets are used. In addition, the GPU performs best with large groups, so larger data sets should help indicate its performance. The large test set used in this thesis can be seen in figure 4.4, and consists of 7 rows of 75 hexagons, totaling 525 bodies falling on surfaces angled at about 1 degree (0.02 radians). There is also a slight step between the 3 surfaces, to help encourage the bodies to fall into each other. In order to create larger data sets, these groups are stacked on top of each other including the platform surfaces,

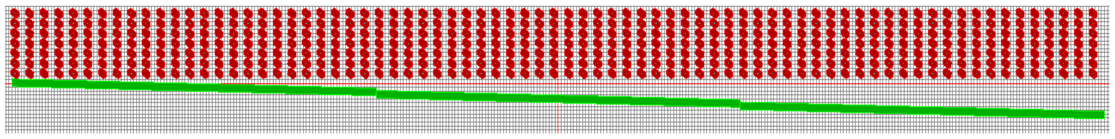


Figure 4.4: The basic test environment for many bodies, with 525 bodies falling on angled surfaces

in sets of 4 (for 2100 bodies) and 10 (for 5250 bodies). Realistically, the groups are independent, but the simulator keeps the groups all in one simulation, resulting in collision checking between all bodies

4.3 Accuracy

When we examine the solvers that were implemented, they are compared to the PATH solver, since the MCP implementation with path has been shown to give accurate results [4]. Additionally, since the proximal point formulations were shown to be equivalent to the MCP formulation, you would expect the two formulations to provide the same answers. It is in this section that we explore how well the solvers as implemented were able to achieve this case. The Friction Angle test case was used to produce all the results in this section.

4.3.1 Special Case

It is worth noting that a special case exists in the collision detection system of dVC2D that when two surfaces sliding along each other meet at their corners, the collision checker considers this to be a wall, until a body gets nudged past it. Each dynamics solver may handle this special case uniquely, but they are all affected by it.

As can be seen in figure 4.5, each solver reacts to the bump differently, but all experience it. Once the corner gets past the edge, sliding is no longer restricted. Differences between the solvers can most likely be explained by how long it takes to pick an impulse to get past the corner. Since there are multiple solutions, the exact solution to the simulation is indeterminate.

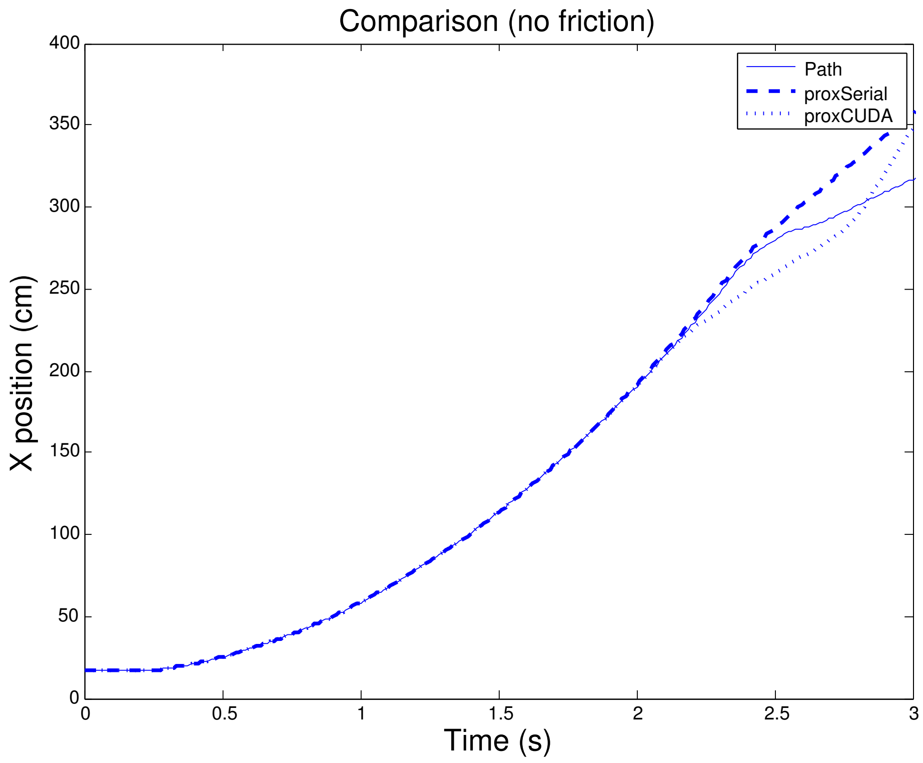
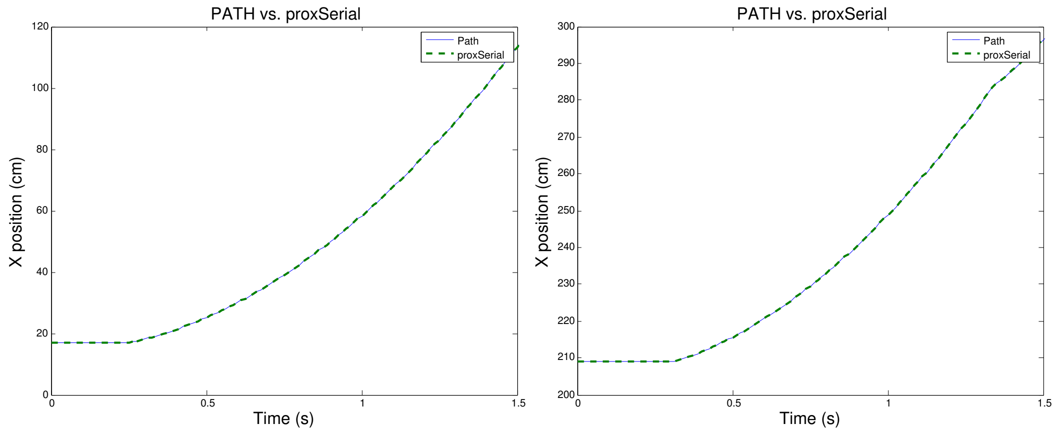


Figure 4.5: The edge bump condition as handled by all three dynamics solvers

4.3.2 Convergence

4.3.2.1 Richardson vs Jacobi

The Relaxed Richardson scheme of calculating the r-value, though computationally simple, is rather useless in most cases. In order to achieve a stable system, the relaxation coefficient has to be set extremely low. How low to set it requires some knowledge of the system, and/or extremely conservative guessing. Since the impulse required to move a body a certain distance is dependent on the mass of that body, largely different masses require largely different impulses for a given ρ. After some casual experimenting, the potential of using the Relaxed Richardson to calculate r-values was abandoned in favor of the Relaxed Jacobi. It is the Relaxed Jacobi that is used in the following examples. How to choose a relaxation coefficient will be discussed in section 4.4, Parameter Tuning.



(a) Body 17 (b) Body 209

Figure 4.6: Comparison of the X position of hexagonal bodies using PATH and proxSerial on a 5 degree surface

4.3.3 Normal Contact

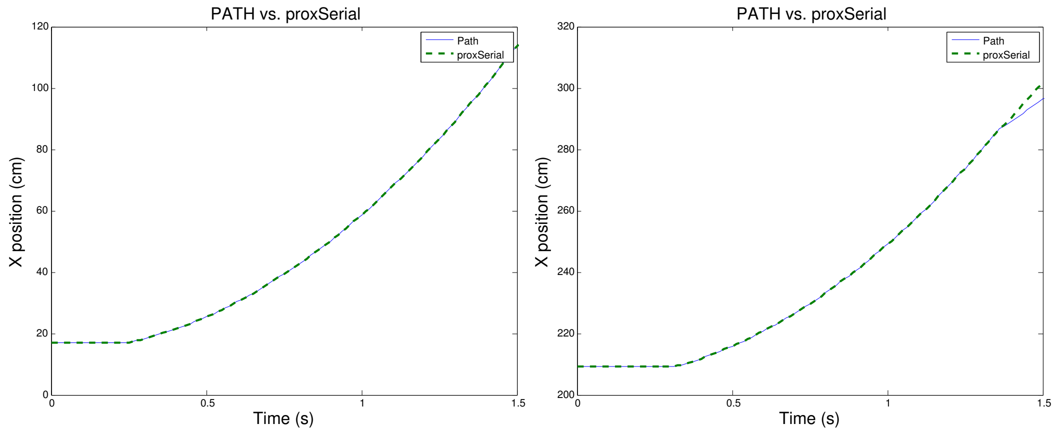
The first test is to check how the solvers compare when only normal contact is involved (that is, there is no friction force). First we check the CPU implementation (proxSerial), then the GPU implementation (proxCUDA). When bodies are referenced, they are referenced by their starting X position. Since the hexagons are resting on the platform, the Y positions and the rotation (θ) are not compared, only the X positions. Additionally, the following tests were done with the maximum number of iterations set to 500 per time-step, and the relaxation coefficient set to 0.5. These were found to be mostly stable and reliable, and further combinations justifying this will be explored later.

4.3.3.1 ProxSerial

The first test case explored is the friction angle case with the platform at 5 degrees, and checked with the hexagons.

As can be seen from figure 4.6, there is a strong agreement between the results of PATH and proxSerial in this case. These two bodies are representative of the results seen from all 25 bodies. Next, we look at the same setup, but with the regular octagons.

In figure 4.7 we can see the same kind of agreement as we saw with the



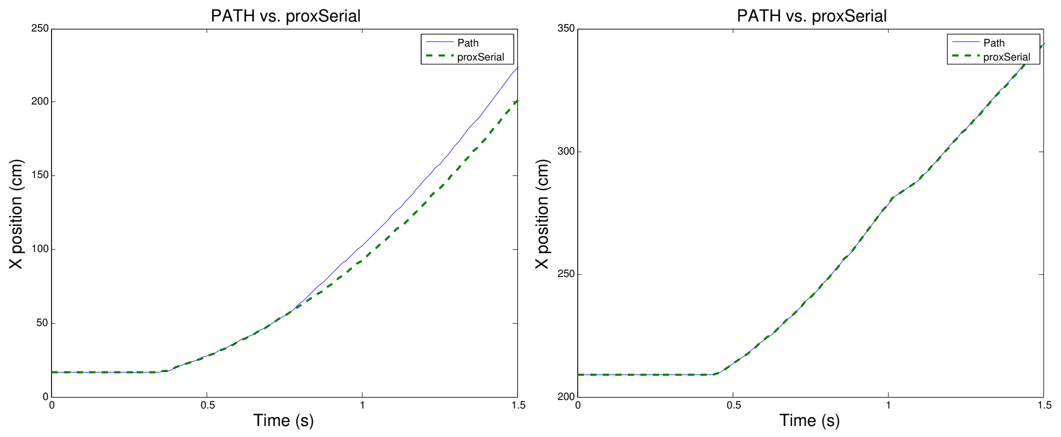
(a) Body 17 (b) Body 209

Figure 4.7: Comparison of the X position of octagonal bodies using PATH and proxSerial on a 5 degree surface

hexagons. The only minor difference here is how body 209 falls off of the edge of the platform. ProxSerial does not slow down the body as much when it hits the edge, and therefore has a greater X velocity when it falls off. Once it falls off, there is no more dynamics solving, and each free falls at a constant horizontal velocity (constant slope on the graph).

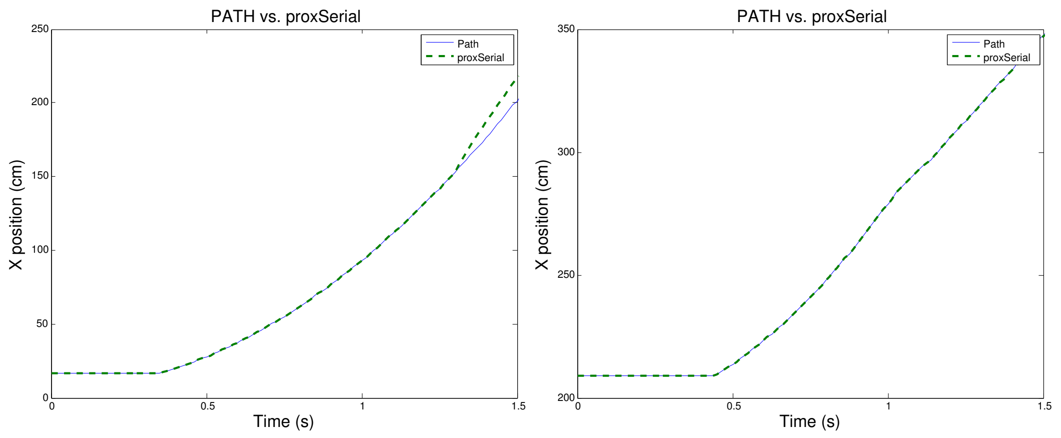
Next, we look at figure 4.8, where we again look at the hexagons, but this time on the 10 degree surface. Here, the normal forces won't be as great, and the velocities will be higher. When looking at body 17, there is a small divergence toward the center of the graph, but the lines from there run relatively parallel. This would seem to suggest a small error by proxSerial at one of the time-steps caused a change in velocity, but after that time-step, they continued to solve in a similar way. This is further corroborated by body 209, which is in near perfect agreement between the solvers, even as it hits the edge (at about 1s) and falls off.

Using the same starting conditions as figure 4.8, but with the octagons, we get very similar results, as can be seen in figure 4.9. Also like in the above case, there is a small divergence with body 17, though this time it is later, at about 1.25 seconds, also most likely from numerical error. This conclusion is further corroborated by setting the relaxation coefficient lower to help with stability. This comparison can be seen in figure 4.10. Here, we get a very strong correlation between the two solvers



(a) Body 17 (b) Body 209

Figure 4.8: Comparison of the X position of hexagonal bodies using PATH and proxSerial on a 10 degree surface



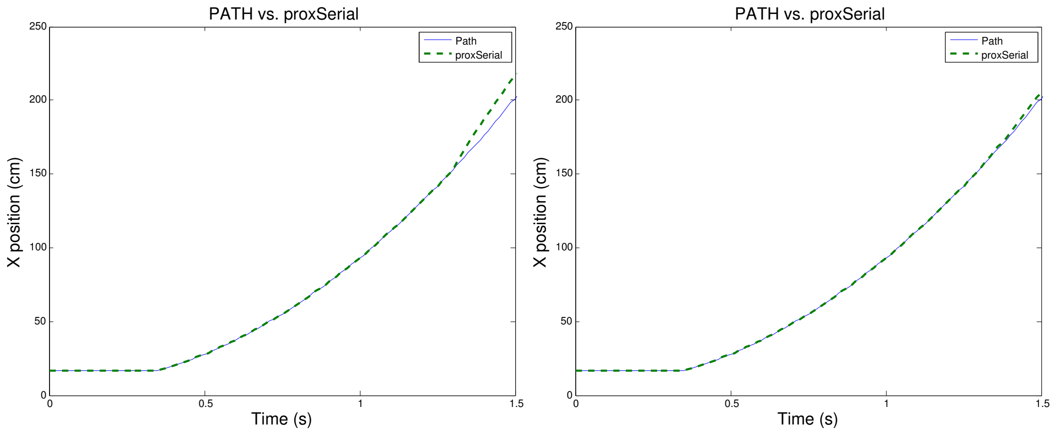
(a) Body 17 (b) Body 209

Figure 4.9: Comparison of the X position of octagonal bodies using PATH and proxSerial on a 10 degree surface

when the relaxation coefficient is set lower, for greater numerical stability.

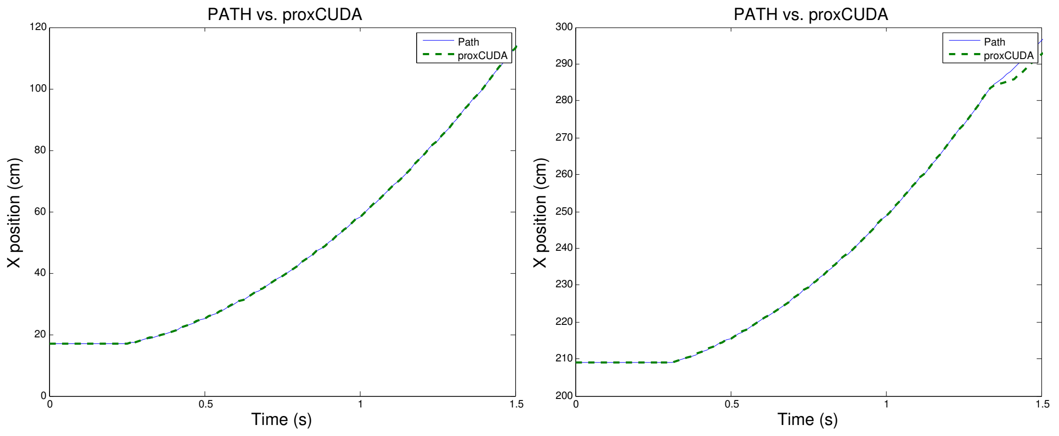
4.3.3.2 ProxCUDA

As with proxSerial, the first test case we will look at is the hexagons falling on the platform set at 5 degrees. As can be seen in figure 4.11, there is strong agreement between the two bodies. There is a small divergence when body 209 hits the end of the platform, but that is understandable. Otherwise, the proxCUDA



(a) Relaxation Coefficient 0.5 (b) Relaxation Coefficient 0.25

Figure 4.10: Comparison of the X position of octagonal body 17 using PATH and proxSerial on a 10 degree surface, with different relaxation coefficients

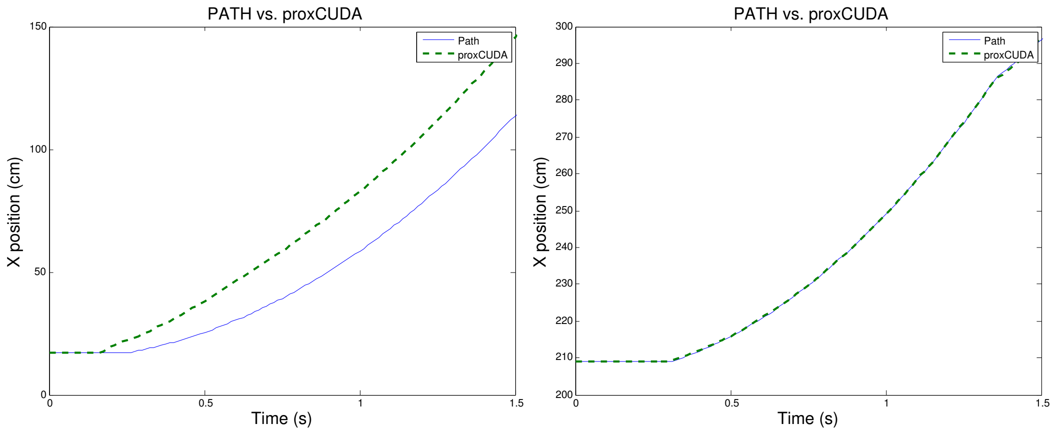


(a) Body 17 (b) Body 209

Figure 4.11: Comparison of the X position of hexagonal bodies using PATH and proxCUDA on a 5 degree surface

implementation agrees with proxSerial and PATH, as we would expect.

Next, we checked the octagonal bodies, the results of which are in figure 4.12. Due to stability problems, the relaxation coefficient had to be set to 0.25 to allow the simulation to run. With these settings, body 209 has a near perfect match between PATH and proxCUDA, while body 17 has a large divergence. Body 17 starts sliding sooner, but the velocity of the two bodies appears to be in agreement. It is hard to tell exactly what went wrong here, but with the exception of the starting conditions,



(a) Body 17 (b) Body 209

Figure 4.12: Comparison of the X position of octagonal bodies using PATH and proxCUDA on a 5 degree surface

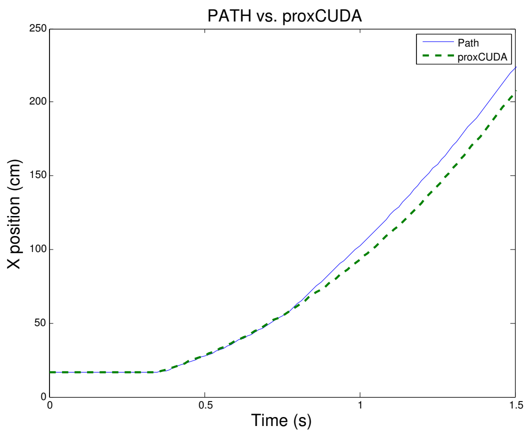
it appears that the two solvers again are working similarly, within numerical error.

Testing the hexagonal bodies on the 10 degree surface, we went back to using the relaxation coefficient of 0.50. Figure 4.13 shows the results. ProxCUDA here showed a slight divergence with body 17, where it would appear a numerical error caused a slight change in velocity, but over time the velocities again match between the two. Body 209 has a much different error, where it appears that it started off with a much higher velocity after landing on the platform. It is hard to discern what would cause such an error.

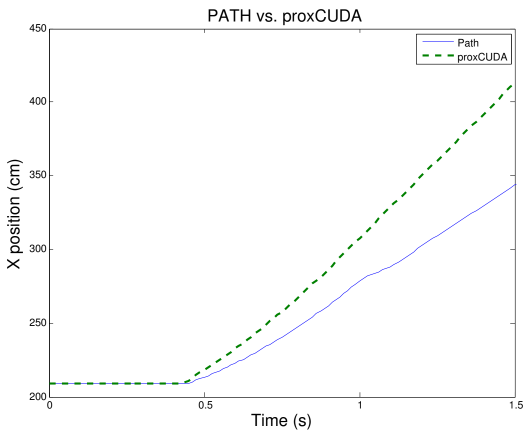
When testing the octagonal bodies on the 10 degree surface, the relaxation coefficient had to again be set to 0.25 in order to converge on a solution. The results of this test run can be seen in figure 4.14. There is again a noticeable amount of difference on both bodies from the results of PATH, but worth noting is that the velocities are about same, resulting in what would be a relatively constant amount of error. Important to note is that the results are of a relatively similar nature, just with a larger amount of positional error.

4.3.4 Friction

In examining the results of adding the frictional impulses to the system, the above scenarios were run again, with varying coefficients of friction (μ). The coeffi-

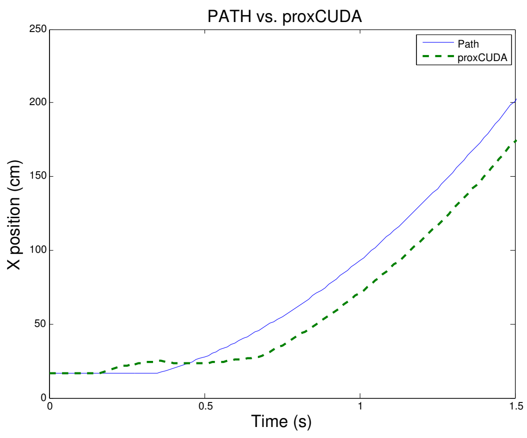


(a) Body 17

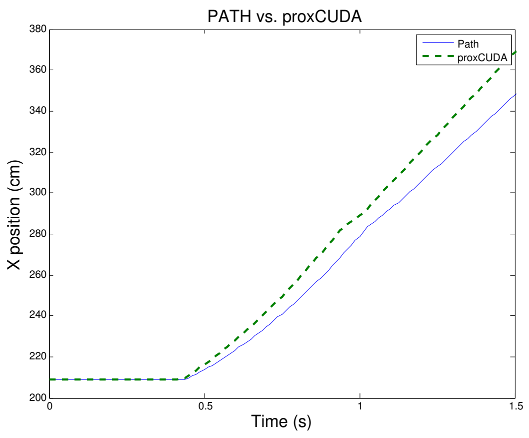


(b) Body 209

Figure 4.13: Comparison of the X position of hexagonal bodies using PATH and proxCUDA on a 10 degree surface



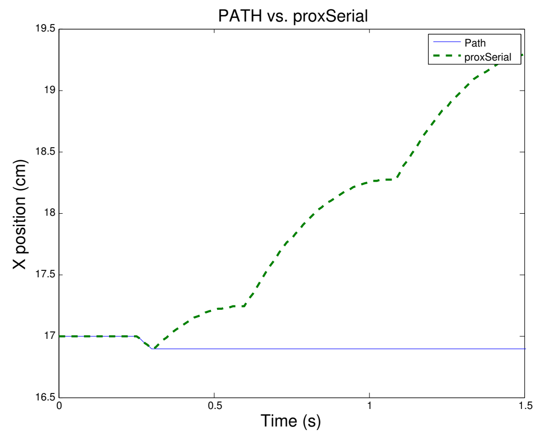
(a) Body 17



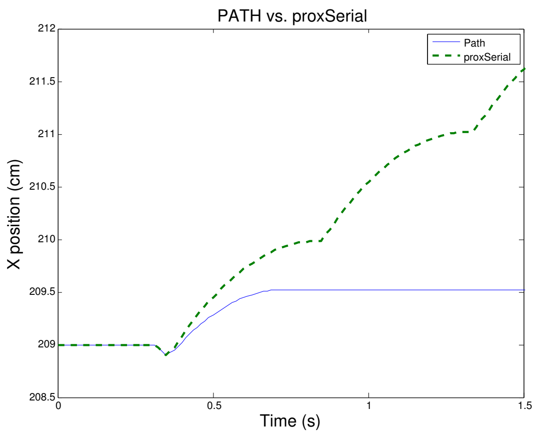
(b) Body 209

Figure 4.14: Comparison of the X position of octagonal bodies using PATH and proxCUDA on a 10 degree surface

cients of friction used were 0.1, 0.2, 0.5, and 1.0. These coefficients of friction have real world equivalents of steel on steel, steel on cast iron, steel on brass, and rough cast iron on rough cast iron, accordingly [26]. The results from the 1.0 tests won't be discussed here, since they are largely similar to the 0.5 results, and don't add any new meaningful observations.



(a) Body 17



(b) Body 209

Figure 4.15: Comparison of the X position of hexagonal bodies using PATH and proxSerial on a 5 degree surface with a coefficient of friction of 0.1

4.3.4.1 ProxSerial

The first test we will look at is the hexagons on the 5 degree platform with a coefficient of friction of 0.1. As can be seen in figure 4.15, the results here are much worse than the frictionless case. Instead of coming to a stop in sliding, the bodies slide for a bit, almost stop, then start sliding again; the end result of which is a sliding motion when there should be sticking. This is almost certainly from a numerical instability, though the exact source is uncertain. A rotation of the body can be seen in figure 4.16, which is characteristic of those bounces that will be seen elsewhere in this section. That bouncing causes the body to penetrate the surface, then get pushed out again. In this process, the body looses its stick and picks up horizontal velocity. Though hard to represent in graphs, watching the simulation makes this behavior clear. Additional evidence towards this being a convergence error can be seen when the relaxation coefficient is set to 0.25, as in figure 4.17. Here, the body mostly holds to the same value that PATH computes, except at 0.75 seconds, when a large divergence is detected. This is most likely caused by another body impacting it, though it is difficult to tell with only this data.

Results with the octagonal bodies was largely similar (figure 4.18), though their behavior with their larger moment arms and smaller bases caused the landing

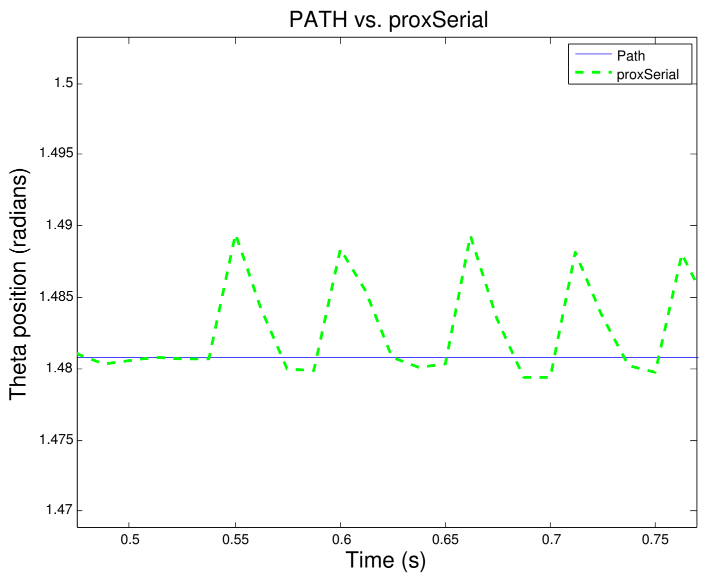


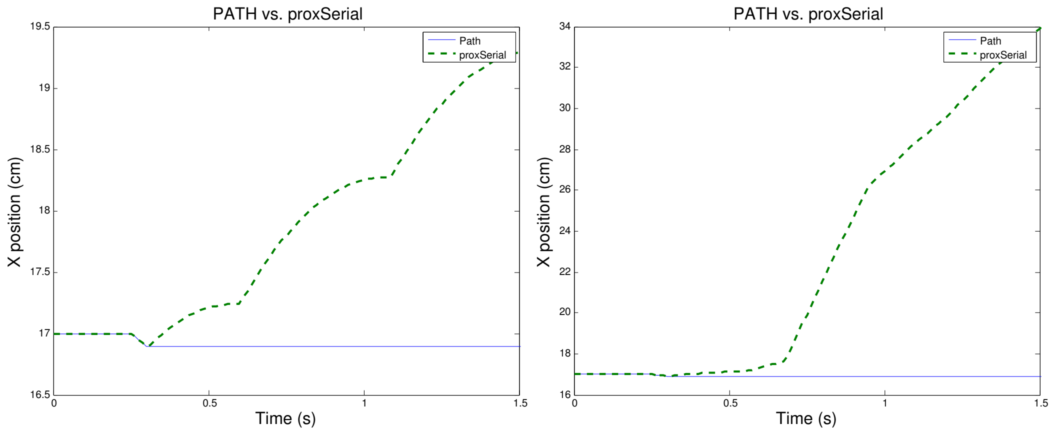
Figure 4.16: The angle of body 17 during steady state for PATH and proxSerial

to also change with body 209. On body 17, you can see that they would have close agreement if it weren't for the bouncing causing sliding instead of the proper sticking.

With the surface angled at 10 degrees, we get much better results. Here, a 0.10 coefficient of friction does not cause the bodies to stick, and therefore not prone to the bouncing observed with the 5 degree surface. These good matches can be seen for both the hexagons (figure 4.19) and for the octagons (figure 4.20).

Going back to the 5 degree surface, but with a coefficient of friction of 0.2, we get similar results. Unlike the case of 0.1 though, both relaxation coefficients at 0.50 and 0.25 have very similar results (figure 4.21). It is worth noting that the results are not identical, suggesting that the relaxation coefficient is causing the results to change, although not significantly like earlier. A much lower relaxation coefficient or different r-value might allow better stability, but that was not investigated here.

When the 5 degree surface is investigated using the octagons (figure 4.22), a different behavior is observed. The PATH results suggests that the body rocks back on landing, then slides forward slightly and stops. ProxSerial however, in not causing the sticking to occur, dampens the rocking. It is still subject to sliding, though it appears that the increased coefficient of friction has reduced the sliding velocity,



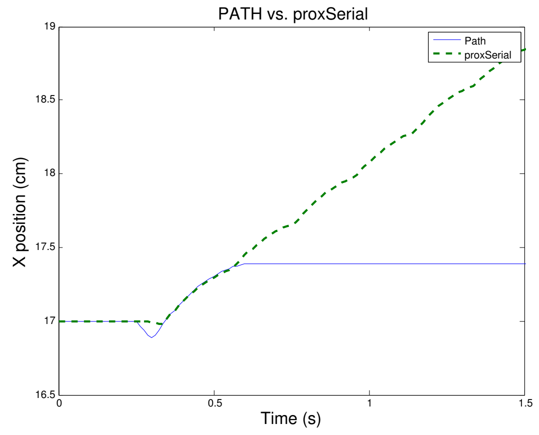
(a) Relaxation Coefficient 0.50 (b) Relaxation Coefficient 0.25

Figure 4.17: Comparison of the effects of the relaxation coefficient on the X position of hexagonal body 17 using PATH and proxSerial on a 5 degree surface with a coefficient of friction of 0.1

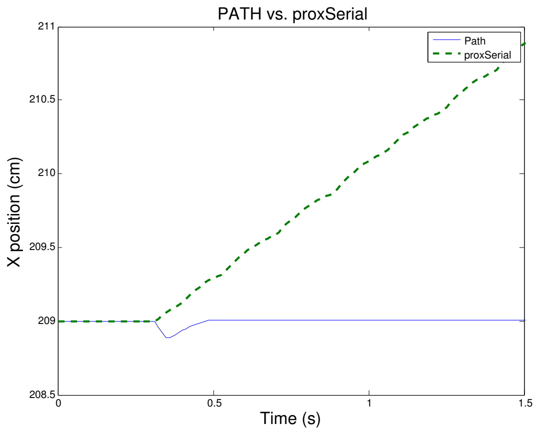
as would be expected given an increased coefficient of friction. So though there is numerical instability, the results are at least heading toward the right direction.

With the 10 degree surface and the hexagons (figure 4.23), body 209 now sticks in PATH, giving similar results to what we saw earlier (proxSerial comes close to sticking, but then slips again, causing relative sliding). Body 17 however slips in PATH, and proxSerial shows similar results. Why both bodies would have different behavior is not entirely clear. Placing the octagons in the same situation (figure 4.24) results in similar differences, where PATH shows sticking, and proxSerial shows sliding. This time, there is not noticeable rocking, suggesting that proxSerial might not slide, even at an optimal solution. Even still, a reduced relaxation coefficient doesn't change the results. Worth noting is that body 17 eventually moves in PATH at about 1 second, suggesting that this could be just at the edge of stick/slip friction at this angle for the hexagons.

Now setting the coefficient of friction to 0.5, both PATH and proxSerial simulate the bodies slipping on the 5 degree surface, both when using hexagons (figure 4.25), and when using octagons (figure 4.26). The bouncing can be seen with both the hexagons and the octagons in the proxSerial solution, but the coefficient of friction is high enough to prevent most slipping anyway. It is notable that proxSerial



(a) Body 17



(b) Body 209

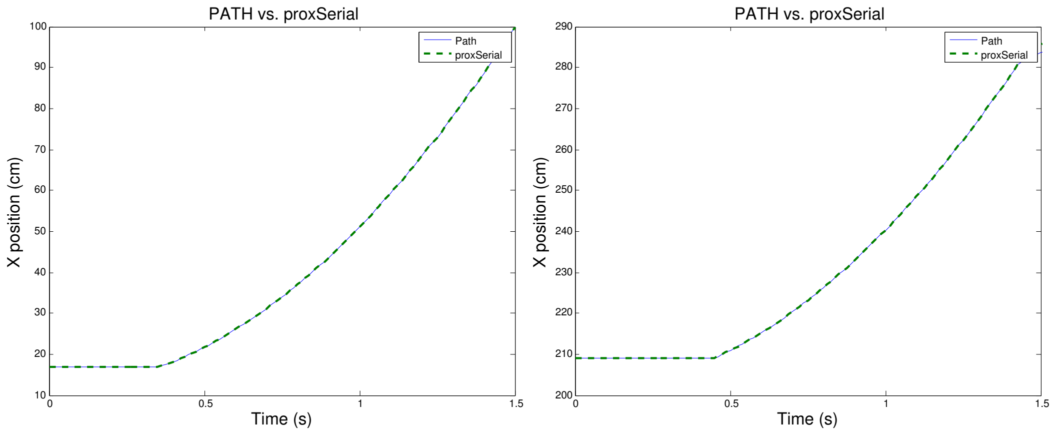
Figure 4.18: Comparison of the X position of octagonal bodies using PATH and proxSerial on a 5 degree surface with a coefficient of friction of 0.1

does allow a lot more slipping than PATH, particularly noticeable with the octagons.

When the surface angle is set to 10 degrees, the hexagonal bodies (figure 4.27) and octagonal bodies (figure 4.28) no longer stick when using proxSerial. It appears that hexagonal body 17 gets bumped, similar as is seen earlier, and proxSerial causes a greater sliding than PATH does. This lack of sticking also appears to be due to numerical errors, where setting the relaxation coefficient lower increases the agreement, but only slightly, as seen in figure 4.29. The lack of sticking finally changes when using a coefficient of friction of 1.0 for the hexagons, but not for the octagons.

4.3.4.2 ProxCUDA

In examining proxCUDA, only the more particular corner cases will be explored, since an exhaustive exploration of the possible scenarios has already been completed in the proxSerial section. As can be seen in figure 4.30, the proxCUDA solver doesn't suffer from the bouncing issue that proxSerial does. Additionally it is worth noting that the Y axis has been scaled to show the maximum amount of detail. In this, figure 4.30(b) shows an agreement in final resting position of 0.5cm. Also worth noting is that both PATH and proxCUDA agree on the body coming to



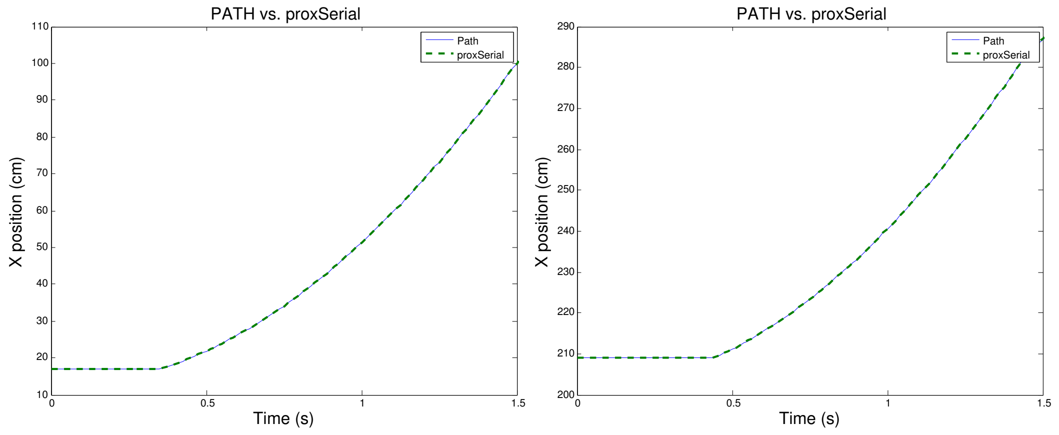
(a) Body 17 (b) Body 209

Figure 4.19: Comparison of the X position of hexagonal bodies using PATH and proxSerial on a 10 degree surface with a coefficient of friction of 0.1

a rest, though body 17 is bumped by another body at 1.25 seconds, most probably a result of the differing amount of time for the bodies to come to rest, which shows some consequence of the differences between the two solvers.

It isn't till the coefficient of friction hits 0.2 that the octagon has the same agreement about stick slip. A comparison of the octagon in the same situation can be seen in figure 4.31. Here you can see the octagon slipping at 0.1, but in the 0.2 it sticks, with only a little steady state error. The steady state error here looks like it is the result of the first half second, where the PATH causes the object to stick and rock, while proxCUDA has the octagon slip on landing.

When the coefficient of friction is increased to 0.5, the hexagon is much more accurate (figure 4.32). It shows results similar to proxSerial, and high agreement with PATH, as well as no bouncing as seen in proxSerial. The octagon shows much worse agreement on position, but does settle into a mostly stuck position, with only a slight slide, most likely the result of small numerical errors. The error with the octagon most likely results from having a much higher center of gravity, but the same moment of inertia as the hexagon. A rock and slide can be seen in the behavior PATH yields, which has a lot of corner cases of sliding and sticking. With that in mind, it seems reasonable that there would be mathematical differences in



(a) Body 17 (b) Body 209

Figure 4.20: Comparison of the X position of octagonal bodies using PATH and proxSerial on a 10 degree surface with a coefficient of friction of 0.1

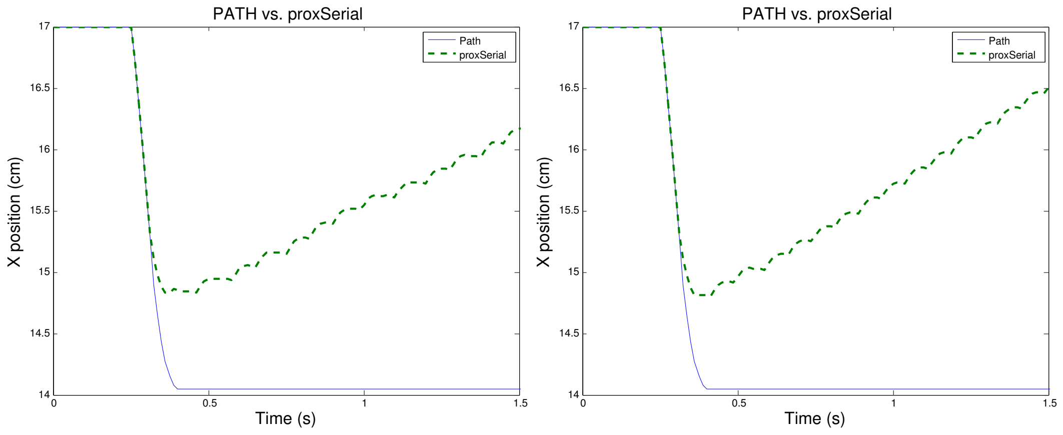
the results, however proxSerial didn't experience these same differences (see figure 4.26).

When PATH shows sliding, as is the case with a coefficient of friction of 0.1 and angle of 10 degrees for the surface (figure 4.33), both the hexagon and the octagon show pretty good agreement. The octagon, however, started sliding much sooner, resulting in some positional error, but close velocities.

As the coefficient of friction increases in the 10 degree case, similar results to above can be seen. The main difference to be noticed is that PATH and proxCUDA have different transition points from sliding to sticking, and that can result in major differences between the two solvers. Additionally, as is well demonstrated in the case of the octagon, when a body lands, the complex combination of falling, landing on an edge, and rotating, rocking and sliding allow small differences in how the impulses are solved to result in large changes in the results.

4.3.5 Single vs Double Precision

When the above tests were run with the floating point variables set to single precision instead of double, no discernible change in the results produced by either proxSerial or proxCUDA occurred. This was expected, since the thresholds for



(a) Relaxation Coefficient 0.50 (b) Relaxation Coefficient 0.25

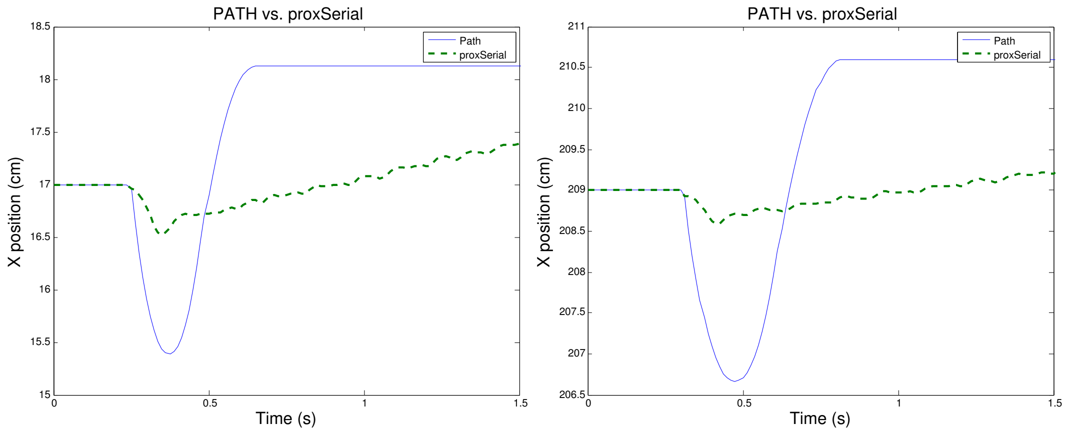
Figure 4.21: Comparison of the effects of the relaxation coefficient on the X position of hexagonal body 17 using PATH and proxSerial on a 5 degree surface with a coefficient of friction of 0.2

convergence are much greater than the precision lost by switching to single point.

4.4 Parameter Tuning

The main parameters that need to be tuned in these simulations are the time-step size, the relaxation coefficient, the maximum number of iterations, and the criteria for when the simulation has reached convergence. All of these with the exception of the time-step size are unique to the proximal point solvers implemented for this thesis. With the time step, the idea behind it is to approximate the integral of the continuous functions. Much work has been done in mathematics on this subject, and so it won't be covered in detail here. The time step size chosen for these simulations was confirmed through a trial and error, where larger time steps resulted in radically broken simulations due to the excessive penetrations that could result in a single time step, and much smaller time steps didn't change the results, but took longer to compute.

Demonstrating the relationship between the relaxation coefficient, maximum number of iterations, and convergence criteria is difficult, since they are all inter-twined properties; changing one has an effect on the others. The convergence criteria is the most independent property, and for all the tests, was set to 1×10^{-6} for the

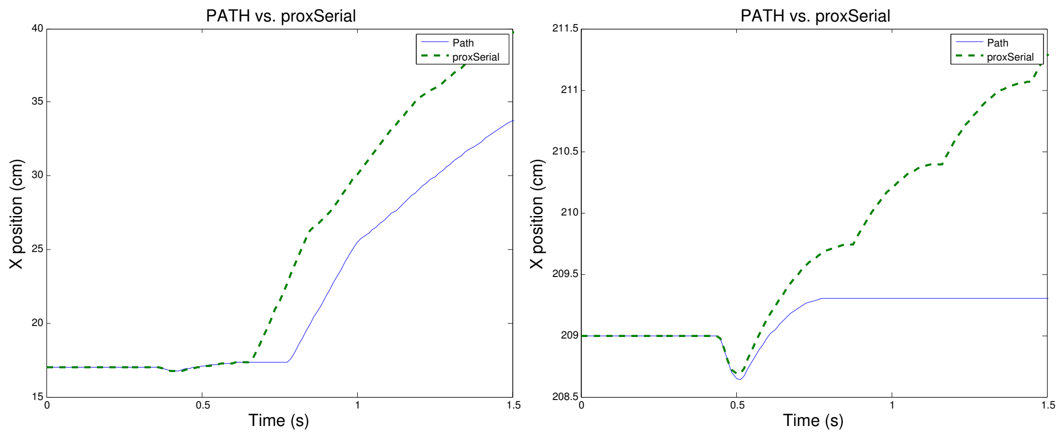


(a) Body 17 (b) Body 209

Figure 4.22: Comparison of the X position of octagonal bodies using PATH and proxSerial on a 5 degree surface with a coefficient of friction of 0.2

normal impulse, and 1×10^6 for the frictional impulse. The convergence criteria is a balance between allowing an acceptable numerical error, and solving for the impulses in a sufficient time. In the tests in this section, the units were in cm, so the normal impulse was considered converged if the penetration depth between the bodies was 1×10^{-8} m, or 10 nm. For the friction impulse, it was considered converged if the relative sliding velocity was less than $1 \times 10^{-8} \frac{\mathrm{m}}{\mathrm{s}}$, or $10 \frac{\mathrm{nm}}{\mathrm{s}}$. This reasonably seemed like a reasonable set of limits, and considering the lack of change between single and double precision, it seems to have been well within the precision limits of the variables. The choice of convergence criteria is dependent on application, but from our work with other simulations not documented in this thesis, 10^6 seems to be a good starting number.

The relaxation coefficient are inversely related in the sense that if you decrease one, you can increase the other and still converge on a solution. Alternatively, if you increase one, you usually will need to increase the other to maintain the stability of the system. Each value has limits that for any given simulation, no amount of change on the other will allow a simulation to work. For example, r-values above 1 rarely if ever work, and setting the maximum number of iterations below 5 will only work in incredibly simple circumstances. In theory, the maximum number



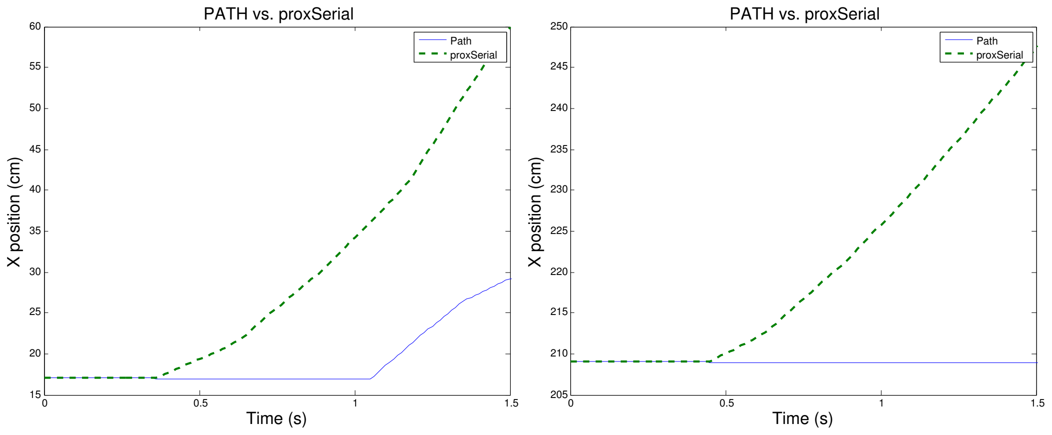
(a) Body 17 (b) Body 209

Figure 4.23: Comparison of the X position of hexagonal bodies using PATH and proxSerial on a 10 degree surface with a coefficient of friction of 0.2

of iterations shouldn't even be needed, since the system should be able to detect convergence and finish when it is done. In reality though, more complex systems, especially when friction is involved, had difficulty all converging at the same time. More testing with convergence criteria would be helpful to fully explore why this is the case. Terminating the simulation after a maximum number of iterations was found to yield stable and accurate simulations, and increasing the maximum number of iterations beyond a point (100 in the case of the friction tests above) was found to have little effect on the results. The simple scenario of the friction tests above found relaxation coefficients of 0.5 to work best, though occasionally 0.25 was used to keep the simulation stable (as was noted above, such as in figure 4.10).

4.4.1 Divergence Detection

The divergence detection system implemented in proxSerial (described in section 3.2.2.2) was a vast improvement over having nothing. Especially when met with long simulations, or automated groups of simulations, having a divergence that results in the simulation failing is unpleasant. Having the maximum impulse set correctly is critical, since values too low can result in the relaxation coefficient going extremely low, where the simulation will take much longer to converge than



(a) Body 17 (b) Body 209

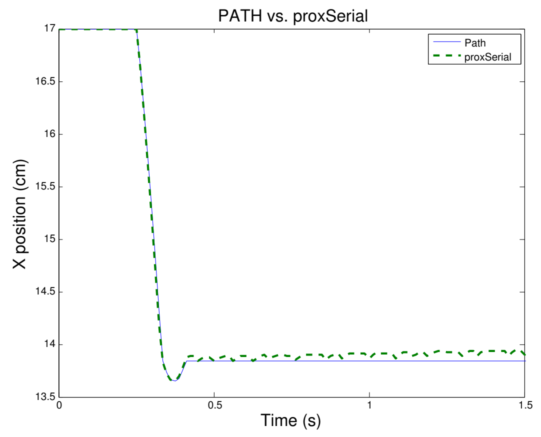
Figure 4.24: Comparison of the X position of octagonal bodies using PATH and proxSerial on a 10 degree surface with a coefficient of friction of 0.2

needed. Having it too high risks allowing the simulation become unstable anyway. In many cases, adding the maximum normal impulse detection improved stability when configured correctly.

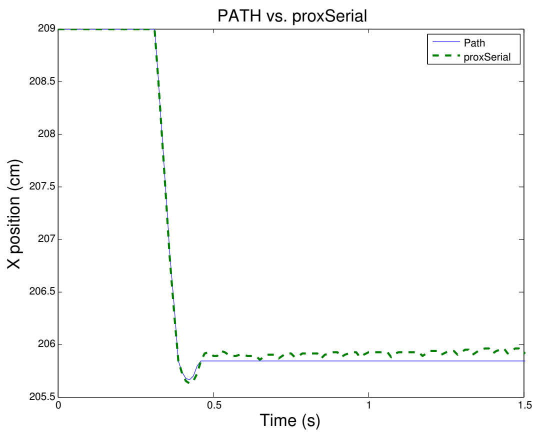
Other systems to detect divergence were tested, and they all relied on checking if the rate of change of p_n was getting bigger or smaller. As a solution is converged upon in the fixed point iteration, it was assumed that the functions would be linear, and therefore as the function converged on a solution, the rate of change of p_n would get smaller. In practice, as bodies interact with each other, and even different contacts on the same body interact, it causes the function of p_n over the iterations (k) to be highly nonlinear, leaving the rate of change for p_n to vary drastically over iterations. Consequently, we deemed it unsuccessful in detecting divergences of the system.

4.5 Performance

In order to test the time performance of the proximal point implementations, the Large Group test was used to test how fast each system could simulate. The tests were run without friction (coefficient of friction set to 0.0), since the accuracy of the non-friction tests were much better than the tests with friction, as shown in



(a) Body 17



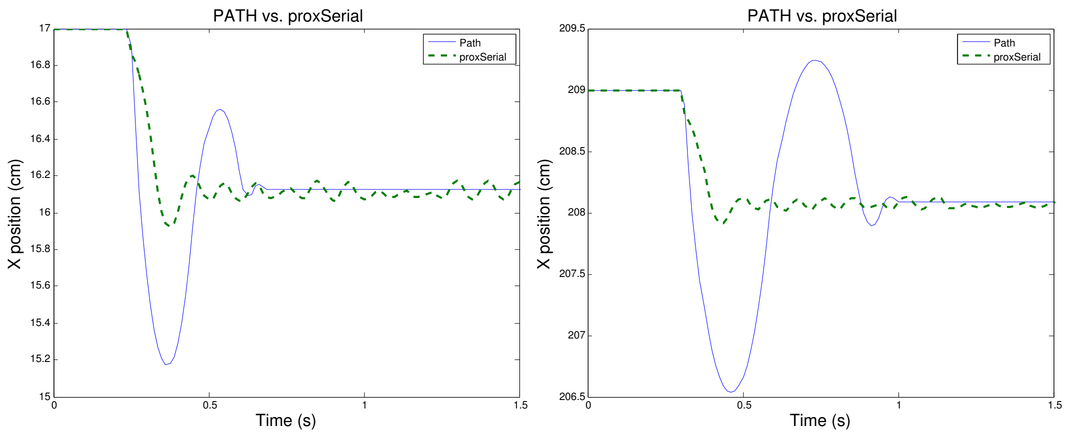
(b) Body 209

Figure 4.25: Comparison of the X position of hexagonal bodies using PATH and proxSerial on a 5 degree surface with a coefficient of friction of 0.5

the previous section. In addition, the relaxation coefficient was set to 0.25, since the large number of colliding bodies resulted in 0.50 providing an unstable simulation prone to divergence. In addition, the maximum number of iterations was set to 2000, as this seemed to provide enough time for the simulation to approach convergence. With so many bodies interacting, previously tested convergence criteria rarely occurred in a reasonable time.

Due to the small differences in the results of each solver, and how the differences compound over time, the end results can be rather different when comparing individual bodies. The overall picture is roughly the same, as can be seen in figure 4.34. In this, you can see how the bodies stack is different, but the overall picture is similar, and more importantly for the purposes of measuring how fast the simulations run, the number of bodies in contact and number of contacts are relatively similar.

Table 4.3 shows how long it takes the three solvers to simulate 1-8 seconds of simulated time for one group of 525 bodies. As the time increases, more bodies land and mix increasing the number of contacts and bodies in collision, resulting in the time per second to increase as the simulation proceeds. The GPU implementation is the slowest, but this is not surprising considering the overhead of running on the



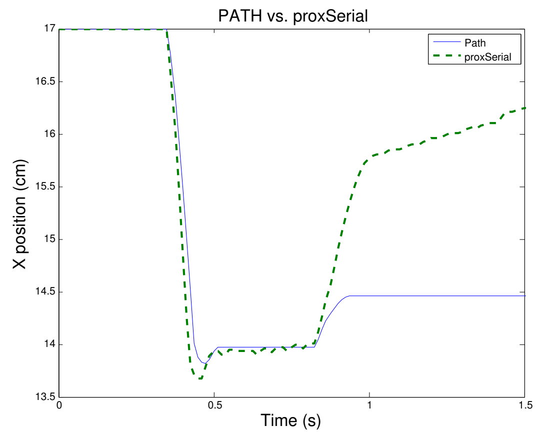
(a) Body 17 (b) Body 209

Figure 4.26: Comparison of the X position of octagonal bodies using PATH and proxSerial on a 5 degree surface with a coefficient of friction of 0.5

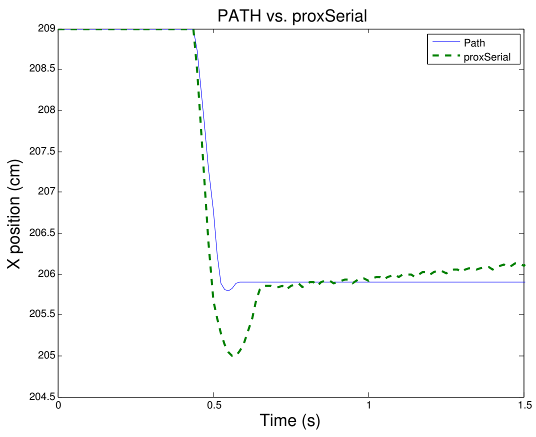
GPU, and the GPU runs best with large numbers of threads running per hardware thread. In this example, the number of threads per hardware thread is close to 1:1. ProxSerial performs much faster than PATH, running in almost half the time. This improvement becomes even more pronounced as the problem size becomes larger. When 4 groups are used (table 4.4), proxSerial runs in less than 1/6th the time it takes for PATH to solve the same problem size. With the group size increased to 4, the performance of proxCUDA is much more favorable to PATH, but still doesn't compare to well to proxSerial, with a run time about 1.5 times as long.

After the group of 4, PATH was no longer compared, because even assuming a linear growth rate, it would take PATH over 9 hours to run. In reality PATH's performance grows much larger than that, making it impractical to run. In running the group of 10, with 5250 bodies, proxSerial and proxCUDA show similar rates of growth (table 4.5), which was not expected. As the number of contacts and bodies increases, proxCUDA should grow more slowly in the time it takes to run. The best explanation for why proxCUDA takes as long as it does is the implementation of the update body dynamics kernel.

Figure 4.35 shows the output of the CUDA Profiler, a program by NVIDIA that allows a developer to analyze the run time performance of GPU kernel calls.



(a) Body 17



(b) Body 209

Figure 4.27: Comparison of the X position of hexagonal bodies using PATH and proxSerial on a 10 degree surface with a coefficient of friction of 0.5

The performance of proxCUDA running the Large Group test of 10 groups was analyzed, and the update body dynamics kernel (named updateBodyNu) took almost 60% of the GPU's running time. By comparison, running the normal impulse took about 5%. Considering how the update body dynamics kernel iterates through all the contacts, many global memory calls are made. In order to hide the latencies of these accesses to global memory, there should be at least 100 threads allocated to each hardware thread. In this simulation, there were only about 10:1. There are a variety of optimizations that could be explored to improve this performance, which will be discussed in the conclusion.

Even with the lower performance, both proxSerial and proxCUDA show vast performance improvements over PATH, graphically represented in figure 4.36. A conservative estimate was made for the run time for PATH with 10 groups, though proxSerial can run 10 groups in less time than it takes PATH to run 4, and proxCUDA only takes a little longer.

Due to the massive run times of running PATH (over 3 hours for the group of 4), only proxSerial and proxCUDA were run in the group of 10 in table 4.5.

Table 4.3: Simulation run times for one group of 525 bodies

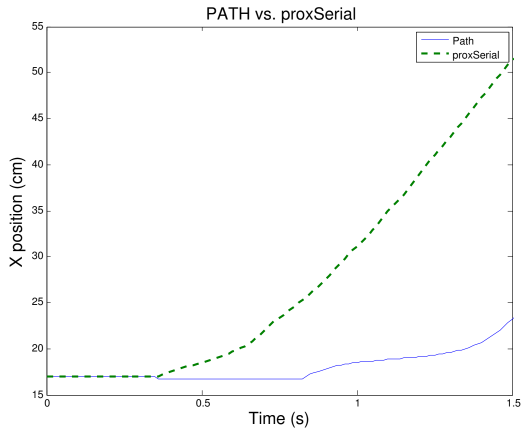
Seconds Simulated	PATH (s)	proxSerial (s)	proxCUDA (s)
1	11.56	11.41	13.04
2	27.86	27.66	49.00
3	82.72	57.18	112.24
4	142.70	90.99	185.29
5	206.21	123.56	259.95
6	274.09	157.99	336.86
7	347.16	190.39	417.33
8	424.71	223.72	504.58

Table 4.4: Simulation run times for four groups of 525 bodies (2100 bodies total)

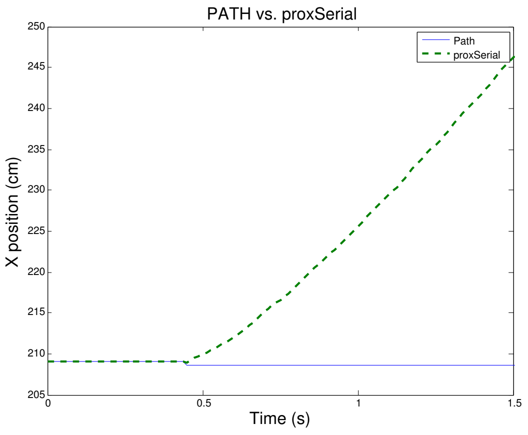
Seconds Simulated	PATH (s)	proxSerial (s)	proxCUDA (s)
1	144.67	141.58	143.76
2	35.01	301.27	342.74
3	1249.64	514.66	642.82
4	3254.85	748.04	980.50
5	5271.30	975.52	1328.49
6	7287.65	1209.98	1688.75
7	9303.84	1438.82	2060.85
8	11320.30	1676.72	2460.73

Table 4.5: Simulation run times for ten groups of 525 bodies (5250 bodies total)

Seconds Simulated	proxSerial (s)	proxCUDA (s)
1	793.83	826.28
2	1637.43	1768.07
3	2604.77	3218.43
4	3619.76	4934.11
5	4626.20	6696.33
6	5650.33	8515.47
7	6657.56	10388.80
8	7686.46	12396.40



(a) Body 17

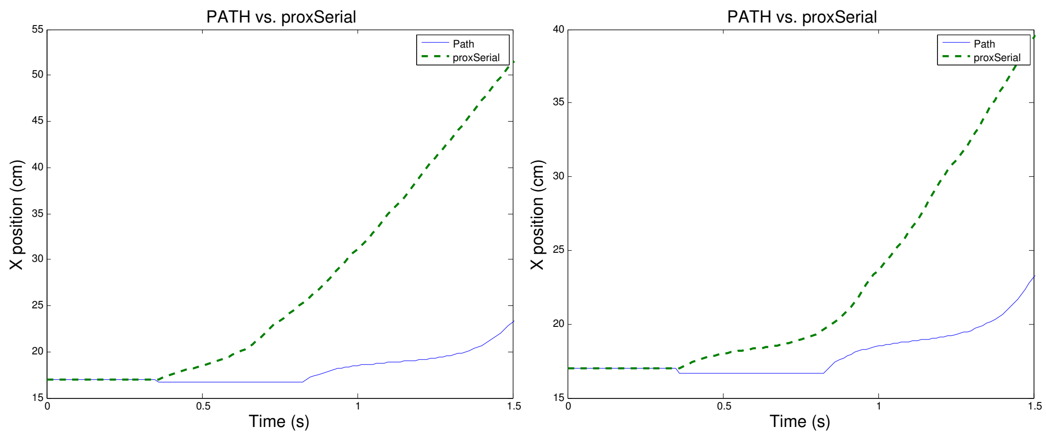


(b) Body 209

Figure 4.28: Comparison of the X position of octagonal bodies using PATH and proxSerial on a 10 degree surface with a coefficient of friction of 0.5

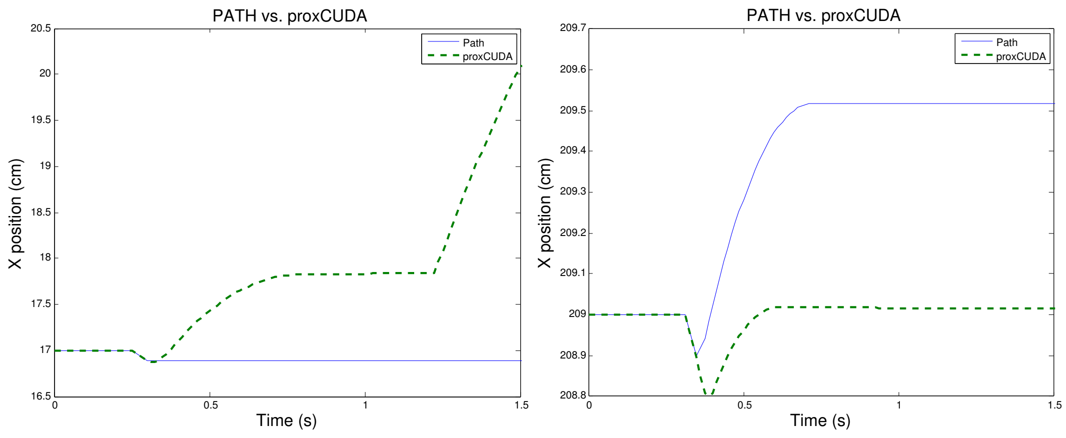
4.5.1 GPU Single vs Double Precision

When comparing the run times of the GPU executions in single precision and double precision, the single precision only produces an average of 5% boost in speed. This was unexpected, since in the GPU tested, the processors can do single precision operations at about 8x the speed of double precision. The fact that the speed up is so small provides further evidence that memory latency issues are at fault for the dynamics update step running for so much longer. If more of the GPU operations were mathematical, we would expect a much greater improvement by using single precision.



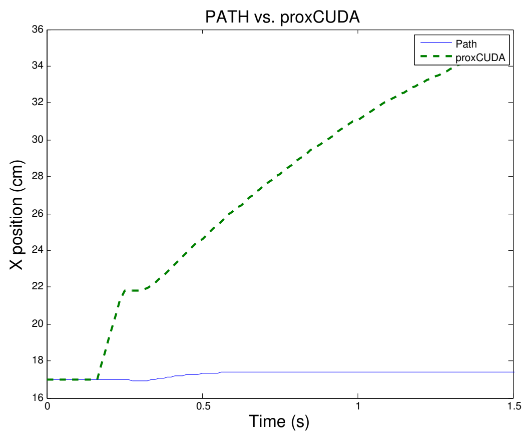
(a) Relaxation Coefficient 0.50 (b) Relaxation Coefficient 0.25

Figure 4.29: Comparison of the effects of the relaxation coefficient on the X position of hexagonal body 17 using PATH and proxSerial on a 10 degree surface with a coefficient of friction of 0.5

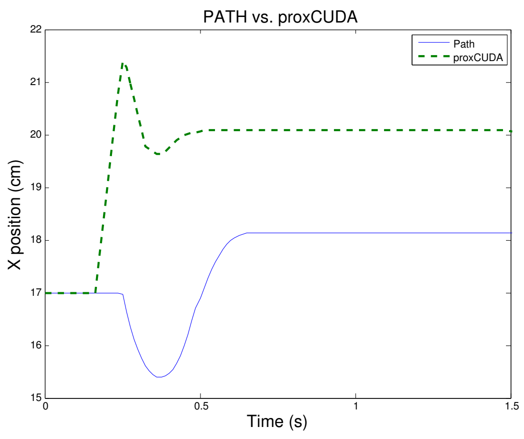


(a) Body 17 (b) Body 209

Figure 4.30: Comparison of the X position of hexagonal bodies using PATH and proxCUDA on a 5 degree surface with a coefficient of friction of 0.1

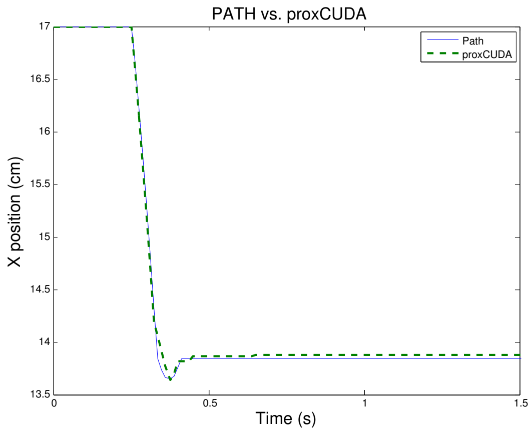


(a) Coefficient of friction at 0.1

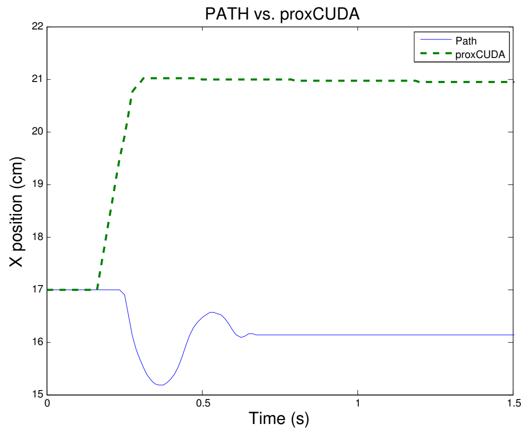


(b) Coefficient of friction at 0.2

Figure 4.31: Comparison of the X position of octagonal body 17 using PATH and proxCUDA on a 5 degree surface with a coefficient of friction of 0.1 and 0.2

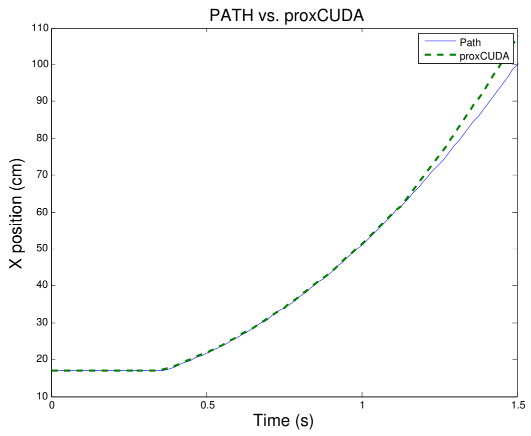


(a) Hexagon

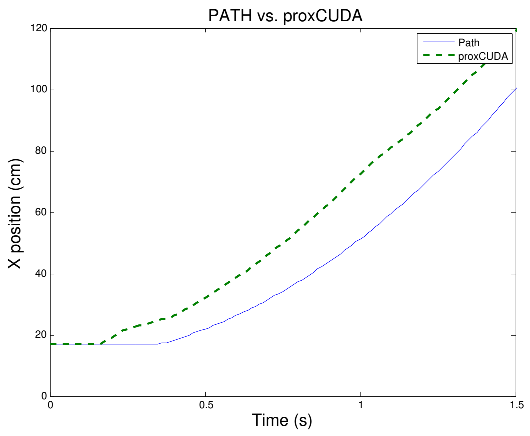


(b) Octagon

Figure 4.32: Comparison of the X position of hexagonal body 17 and octagonal body 17 using PATH and proxCUDA on a 5 degree surface with a coefficient of friction of 0.5

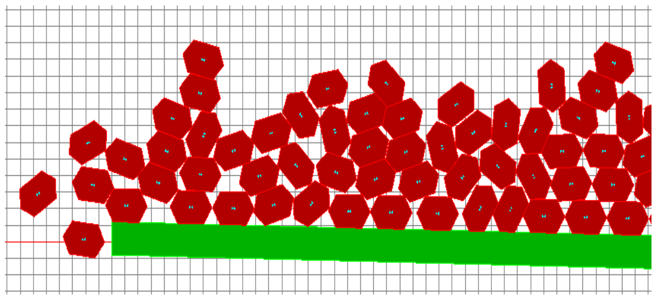


(a) Hexagon

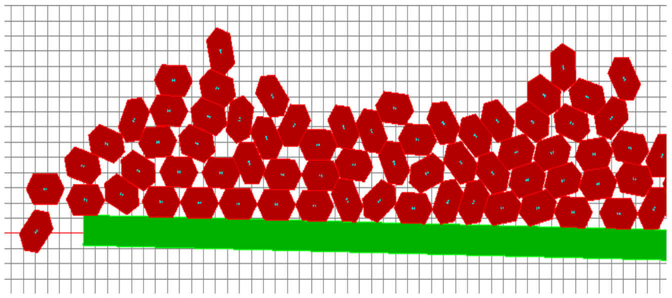


(b) Octagon

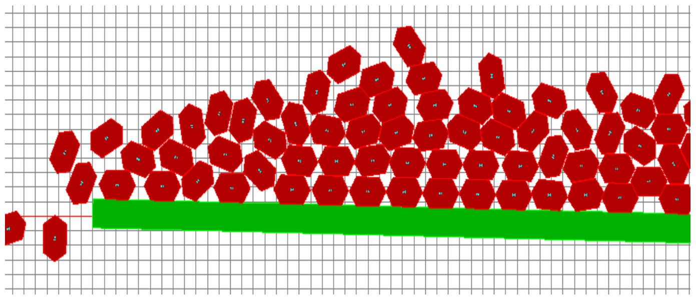
Figure 4.33: Comparison of the X position of hexagonal body 17 and octagonal body 17 using PATH and proxCUDA on a 10 degree surface with a coefficient of friction of 0.1



(a) PATH



(b) proxSerial



(c) proxCUDA

Figure 4.34: A section of the Large Group test simulation run after 1 second with each of the dynamics solvers

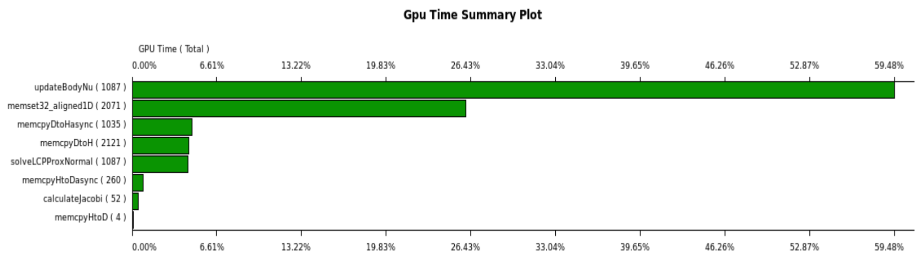


Figure 4.35: Result of CUDA Profiler on large group with 5250 bodies

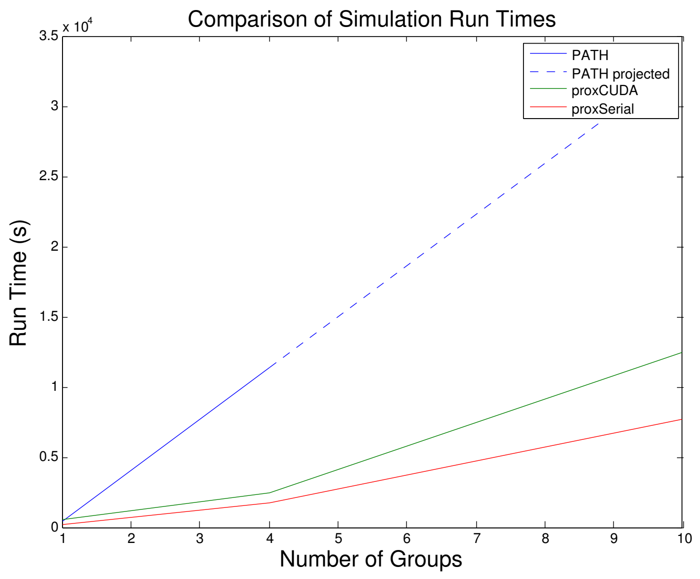


Figure 4.36: Scaling of running times for all solvers (The results of PATH are projected)

CHAPTER 5
CONCLUSION

This thesis explored the ability of the proximal point formulation to be used in multibody physics simulation to replace the complementarity formulation for solving dynamics. In particular, it aimed to explore the possible use of the proximal point formulations on the GPU for improved performance and scaling. On this, the results presented in this thesis show promise towards that possibility, though there are many more problems that need to be explored before this implementation would be well suited to replace PATH and the complementarity formulation.

First, removing or better understanding the differences between PATH and the proximal point solvers' results would be important to prove the validity of the results of the proximal point implementations. Worth exploring in particular are why the bodies bounce when friction is used in proxSerial, and why the steady state errors exist in proxCUDA. On the CPU, some possible avenues of exploration would be to verify how the compiler builds the mathematical operations, and if there are optimizations that need to be disabled, such as with compiler flags. Similar explorations could be made on the GPU, where some operations are known to deviate from the IEEE standards for floating point numbers [27]. It would also be worthwhile to explore how the proximal point implementations compare to actual experiments, rather that assuming that the complementarity formulation and PATH are completely accurate.

Some other possibilities to the above problems would be to explore other ways of detecting convergence, and what to do when a contact does reach convergence. Of particular utility would be the ability to adjust what would constitute convergence automatically, based on the magnitudes of the bodies size, mass, and velocity. It might also be important to adjust convergence on a per contact basis, if large and small bodies are mixed in the same simulation, a case that was not explored in this thesis.

What might further improve performance, and possibly accuracy, would be better choices of the r-value. Though the Relaxed Richardson and Relaxed Jacobi were explored, the Relaxed Gauss-Seidel shows great possibilities to allow a more aggressive relaxation coefficient, while still maintaining system stability. Some of the numerical errors caused by the addition of friction could be a result of instabilities. Additionally, combining the Delassus matrices from normal and friction might help with the instabilities as well. The performance hit taken by having such a large matrix to perform calculations might be partially mitigated by using the CUDA linear algebra library, CUBLAS [28]. If the GPU is able to provide a performance boost, it may also be possible to use multiple GPUs working on the same simulation to gain even more performance for very large simulations. In such a case, minimizing communications between GPUs would be needed to keep memory latency to a minimum, but building boundaries where a penalty method is used at the boundary is one possibility [29].

With the CPU implementation of the proximal point formulation performing far better than expected, it would also be worthwhile to make proxSerial multi-threaded on the CPU, since the `for` loop nature of the current implementation is a good candidate for a multi-threaded port, and CPUs trending towards also having many cores. Additionally, a hybrid solution of updating the body dynamics on the CPU, and running the proximal point equations on the GPU may also be worth exploring, though more detailed information about the performance of the specific parts of the simulation would be helpful to determine if that is an avenue worth pursuing.

It would also be worthwhile to explore separating the simulations from one large simulation, to many simulations run in parallel, such as exploring a space of possibilities in simulation. In this case, it would be known what bodies could collide with what, making collision checking and dynamics updates substantially faster. In a case like this, PATH would scale linearly with the number of simulations, but the dynamics update on the GPU would also be much faster, providing another situation that may be better suited for GPU acceleration. In addition, currently each step in the GPU is performed with a separate kernel call, each of which has a certain overhead slowing down the simulation. Synchronization between all the multiprocessors made it difficult to implement a single kernel to fully solve for the dynamics, but if each multiprocessor could solve an isolated simulation, it may be possible to port the entire dynamics update to the GPU. It may also be possible to check for convergence in the GPU, though tests would be needed to determine which method would be faster.

A deeper rewrite of dVC2D to use either the data structures for the proximal point implementations, or a data structure more agnostic to which technique is used might also yield better performance. In addition, it may be possible to port other components of dVC2D to the GPU, such as collision detection.

REFERENCES

[1] NVIDIA Corporation. NVIDIA PhysX, accessed July, 2011. URL: http://www.nvidia.com/object/physx_new.html.

[2] S.G. Berard. *Using simulation for planning and design of robotic systems with intermittent contact*. PhD thesis, Rensselaer Polytechnic Institute, 2009.

[3] Columbia University Robotics Group. Dexterous grasping, accessed July, 2011. URL: http://grasping.cs.columbia.edu/.

[4] L. Zhang, J. Betz, and J. Trinkle. Comparison of simulated and experimental grasping actions in the plane. *The 1st Joint International Conference on Multibody System Dynamics*, 2010.

[5] NVIDIA Corporation. CUDA, accessed July, 2011. URL: http://www.nvidia.com/object/cuda_home_new.html.

[6] H. Nguyen. *GPU Gems 3*. Addison-Wesley Professional, 2007.

[7] D.E. Stewart. Time-stepping methods and the mathematics of rigid body dynamics. *Impact and Friction of Solids, Structures and Machines*, 1, 2000.

[8] J.S. Pang, J. Trinkle, and G. Lo. *A complementarity approach to a quasistatic rigid body motion problem*. Texas A & M University, Computer Science Dept., 1993.

[9] M. Ferris. The path solver: A non-monotone stabilization scheme for mixed complementarity problems. *Optimization Methods and Software*, 5:123–156, 1995.

[10] T.M. Preclik, K. Iglberger, and U. Rude. Iterative rigid multibody dynamics. *Proceedings of Multibody Dynamics*, 2009.

[11] A. Tasora and M. Anitescu. A matrix-free cone complementarity approach for solving large-scale, nonsmooth, rigid body dynamics. *Computer Methods in Applied Mechanics and Engineering*, 2010.

[12] A. Tasora and M. Anitescu. A convex complementarity approach for simulating large granular flows. *Journal of Computational and Nonlinear Dynamics*, 5, 2010.

[13] Á. Moravánszky. Dense matrix algebra on the gpu. *ShaderX2: Shader Programming Tips and Tricks with DirectX*, 9:352–380, 2003.

[14] A. Tasora and D. Negrut. A parallel algorithm for solving complex multibody problems with stream processors. *International Journal for Computational Vision and Biomechanics*, 2009.

[15] S. Strobl. *GPU-Based Rigid Body Dynamics.* PhD thesis, Friedrich-Alexander-Universitat Erlangen-Nurnberg, 2009.

[16] R.I. Leine and H. Nijmeijer. *Dynamics and bifurcations of non-smooth mechanical systems.* Springer Verlag, 2004.

[17] I. Necoara and J.A.K. Suykens. A proximal center-based decomposition method for multi-agent convex optimization. In *Decision and Control, 2008. CDC 2008. 47th IEEE Conference on*, pages 3077–3082, 2008.

[18] F. Pfeiffer. Problems of nonsmooth dynamics. *Lehrstuhl fuer Angewandte Mechanik, TU-Muenchen*, 2008.

[19] T. Schindler. *Spatial Dynamics of Pushbelt CVTs.* PhD thesis, Technische Universitat Munchen, 2010.

[20] S.G. Berard. Cooking with complementarity: A recipe guide for complementarity based rigid-multi-body dynamics simulation, 2006.

[21] R.W. Cottle, J.S. Pang, and R.E. Stone. *The linear complementarity problem.* Society for Industrial Mathematics, 2009.

[22] J. Trinkle, J.S. Pang, S. Sudarsky, and G. Lo. On dynamic multi-rigid-body contact problems with coulomb friction. *ZAMM-Journal of Applied Mathematics and Mechanics/Zeitschrift für Angewandte Mathematik und Mechanik*, 77(4):267–279, 1997.

[23] J. Trinkle, S.G. Berard, and J. Pang. A time-stepping scheme for quasistatic multibody systems. In *Assembly and Task Planning: From Nano to Macro Assembly and Manufacturing, 2005.(ISATP 2005). The 6th IEEE International Symposium on*, pages 174–181. IEEE, 2005.

[24] C. Studer. *Numerics of Unilateral Contacts and Friction: Modeling and Numerical Time Integration in Non-Smooth Dynamics.* Springer Verlag, 2009.

[25] *CUDA C Best Practices Guide.* NVIDIA Corporation, 2011.

[26] J.A. Collins and D.B. Dooner. *Mechanical design of machine elements and machines: a failure prevention perspective.* Wiley, 2003.

[27] *CUDA C Programming Guide Version 4.0.* NVIDIA Corporation, 2011.

[28] *CUDA Toolkit 4.0 CUBLAS Library.* NVIDIA Corporation, 2011.

[29] T. Heyn et al. Enabling computational dynamics in distributed computing environments using a heterogenous computing template. *Proceedings of the ASME 2011 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference*, 2011.

APPENDIX A
PHYSICS REVIEW

The dynamics of rigid multibody systems is based on classical Newtonian mechanics. The most common and well known formulation is Newton's second law:

$$\sum \vec{f} = m\vec{a} \tag{A.1}$$

Where $\sum \vec{f}$ is the sum of the forces applied to the body, m is the mass of the body, and $\vec{a}$ is the acceleration of the body. With this law, the interactions between bodies can be calculated. The motion of bodies is described with Kinematics, and the forces on bodies is dynamics. With these, we can describe the motion and interactions between bodies.

When we talk about bodies, we will be referring to "rigid" bodies, which means that the bodies do not change shape or otherwise deform. Describing the bodies of the system as rigid is a simplifying assumption used to make physical simulation easier. Though no real body is truly "rigid", it can be shown that the rigid assumption is a safe an accurate assumption to make for many cases [4]. It can also be intuitively observed that most objects that we interact with daily (such as cups, phones, pens, tables, etc.) don't visibly deform, and the effects of any small deformation can be neglected and still yield useful results.

A.1 Basic Physics

A.1.1 Kinematics

Kinematics is defined as the branch of mechanics that studies the motion of a body or a system of bodies without consideration given to its mass or the forces acting on it. This allows us to separate the description of the motion of body from the forces acting on the body. In order to describe the motion of bodies in our simulation, we will be using two properties, position and velocity. Each of these both have two components, linear and angular. In two dimensional space, a body has two linear positions (X and Y) and one angular position (commonly θ). The linear position is usually of the center of mass of a body, and the angular position is rotation about that point. The center of mass of a body is the average location of all the mass of a body. Each of these positions can be differentiated, giving two linear velocities ($\dot{X}$ and $\dot{Y}$), and one angular ($\dot{\theta}$ or ω). Velocity can be differentiated further into acceleration as you might expect, but that is not important here.

A.1.2 Dynamics

Dynamics refers to how forces cause motion in bodies. In our study of simulation, there are several possible sources of such forces. The simplest of these forces are directly applied external forces. In addition to forces applied externally to a single body, there are also the forces that result from contact and interaction between multiple bodies. Equation A.1 can be expanded into other forms using differential calculus. When looking at simulation using a time-stepping system, it can be helpful to look at Newton's second law in terms of impulses.

An impulse is the integral of a force applied over a time, or more simply a force applied over a given amount of time. An impulse can also be defined as a change in momentum. Momentum has two components, linear and angular. Linear momentum (p) is equal to the body's mass times its linear velocity.

$$p = m\vec{v} \tag{A.2}$$

Angular momentum (L) is the angular velocity of a body times its moment of inertia, I. The moment of inertia is a measure of a body's resistance to changes in rotation. Just like the mass of a body resists changes in velocity, the moment of inertia of a body is also sometimes called its angular mass.

$$L = I\vec{\omega} \tag{A.3}$$

These forces (or impulses) can come from 3 main sources. They are normal contact, friction, and external forces.

A.1.2.1 Normal Contact

The normal force is the force that a body exerts normal to its surface. As you are probably familiar with in your day to day life, objects do not interpenetrate. The normal force is the force that two bodies in contact exert on each other to keep that from happening. When we talk about the normal force in this way we call it a constraint, since it constrains what our bodies are allowed to do.

The normal contact constraint, also called unilateral constraint, is the simplest and most obvious constraint on body interactions. A gap function, typically noted as Ψ_n, represents the distance between two bodies. Additionally, the normal force, noted as λ_n is the force with which the two bodies push on each other. When $\Psi_n \geq 0$, $\lambda_n = 0$ since the bodies must be in contact in order to push. Consequently, if $\lambda_n > 0$ then $\Psi_n = 0$. Put simply, if there is no normal force then the bodies are not penetrating; if there is a force between two bodies, then the bodies are touching but not penetrating.

A.1.2.2 Friction

Friction is the force that results from two bodies sliding past each other. It is always perpendicular to the normal force at a point of contact, and also like the normal force, can only exist when 2 bodies are in contact. Friction is a very complex set of interactions, and has several types, but we will also only explore dry friction, for any other type of friction would be well beyond the scope of this thesis. Dry friction can be modeled simply using Coulomb's Law of Friction. Coulomb's law states simply that the force of friction λ_f is less than the normal force times a coefficient of friction μ.

$$\lambda_f \geq \mu\lambda_n \tag{A.4}$$

The coefficient of friction is a property of the materials of contact, which can usually be looked up if the two material types are known. The frictional force is also independent of velocity, therefore, if $\vec{\nu} > 0$, then $\lambda_f = \mu\lambda_n$. The stick-slip model of friction described here is also sometimes referred to as stiction.

A.1.2.3 External Forces

External forces are forces that come from outside the system being measured. In our case, the system being measured would be our interacting bodies. In the cases of friction and the normal force, energy and momentum are conserved (slowing down one body will speed up another). External forces are not limited by this requirement, and can therefore add or remove energy from the system.

The most common external force is gravity. The force of gravity acts on the center of mass of a body, and is a constant force acting on a body[2]. The gravitational force for a body is only dependent on the mass of a body, and therefore does not change for a body with constant mass, which is an assumption made for rigid bodies (as in, rigid bodies won't leak, break, or otherwise shed material).

In addition to the gravitational force, dVC allows for applying external forces and moments to a body to act as a controller. These forces act directly on the center of mass of the body and since they are given by the controller, are known absolutely.

[2]Technically, gravity can vary depending on location, but for all but the most extreme of situations, it can be treated as constant with no adverse consequences, and therefore will be treated as such here

APPENDIX B
SOURCE CODE LISTINGS

B.1 Timestepper

B.1.1 STProx.h

```
/** @file plugins/timestepper/StewartTrinkle/STProx.h
 * The Stewart Trinkle Prox
 * @author: Jeremy Betz
 * @ingroup: STStepper
 */

#pragma once

#include "StewartTrinklePlugin.h"

using namespace DVC;

/**
 * The StewartTrinkleProx timestepper class
 * extends the StewartTrinkleStepper class and formulates the
 * @ingroup: STStepper
 */
class StewartTrinkleProx : public StewartTrinkleStepper
{
public:

    StewartTrinkleProx();
    ~StewartTrinkleProx();
    void refresh( const PrefMap & prefs );
    const std::string &getName() const;
    bool pluginStepForward( Input& input );

    int debug;

private:

    bool formNormalProx(); // Form the data structure for normal prox solver

    void initProxBodies(); // Form the data structure for the dynamical bodies

    // Update the sate based on the output of the Prox
    void recordDynamicProx();

    Prox_Solver *proxSolver;


};
```

B.1.2 STProx.cpp

```
/** @file plugins/timestepper/StewartTrinkle/STProx.cpp
 * This file contains StewartTrinkle Prox plugin implementation
 * @author: Jeremy Betz
 * @ingroup: STStepper
 */
```

```
#include "STProx.h"
#include "ForceController.h"
#include "UnilateralConstraint.h"
#include <stdio.h>

#include <stdexcept>

/**
 * Default constructor.
 */
StewartTrinkleProx::StewartTrinkleProx() : StewartTrinkleStepper()
{
   m_integrationOrder = FIRST_ORDER; // ST Prox is velocity based, so integration order is 1

}


/**
 * Destructor.  Clean up the data.
 */
StewartTrinkleProx::~StewartTrinkleProx()
{

}


void StewartTrinkleProx::refresh( const PrefMap & prefs )
{
    StewartTrinkleStepper::refresh(prefs);  //call super's refresh method
    // Create an easy to use pointer that casts the cpSolver as a generic Prox_Solver
    proxSolver = (static_cast<Prox_Solver *>(m_cpSolver));


    //get the timestepper specific settings
    //std::string pluginPath = GetPluginPrefPath();

  //m_useSparse = prefs.get_bool(pluginPath + "/useSparseMatrix");


}

bool StewartTrinkleProx::pluginStepForward(Input& input )
{
    // Determines the indexing scheme among other duties (like computing m_numConstrainedBodies)
    TimeStepper::calculateMatrixIndices();

    // Only need to formulate if there is at least one constrained body
    if ( m_numConstrainedBodies > 0)
    {

  // Tell the solver the stepsize (and other init tasks later?)
  proxSolver->initProx(m_stepSize, m_coefficientFriction, m_simTime, m_gravity);
        // Put data into datastructures
  proxSolver->populateProxDataStructure(m_dynamicalBodies,m_collisions, m_numConstrainedBodies,
      m_numContacts);
  // Solve using only normal impulses
  proxSolver->solveProx();

        recordDynamicProx();

    }

    //update the dynamical bodies that have no constraints using the TimeStepper class' function
    this->UpdateFallingBodies();
    return true; // If we made it here, it was a successful step
}


void StewartTrinkleProx::recordDynamicProx()
{

    DVC::Vector<REAL> nu(3);

    // Now iterate through each body that needs to be updated
    std::list <DynamicalBody *>::iterator bodyItr = m_dynamicalBodies->begin();
    for ( ; bodyItr != m_dynamicalBodies->end(); ++bodyItr )
    {
        DynamicalBody *b = *bodyItr;
        if( !(b->IsInCollision() || b->IsJointed() || b->HasGeneralUnilateralConstraints() )  )
        { // free fall body
           continue;
        }
        unsigned int index = b->GetWrenchRowIndex()/3;

  // Get velocity of body 'index' from the proxSolver, save it in nu
  proxSolver->getProxBodyNu(index,&nu);

  b->SetNu(nu);
  TimeStepper::Integrate(b, m_integrationOrder);
    }

}


// The following methods are required for all plugins //
///  Returns the name of the plugin
/// @return A string containing the plugin's name
const std::string & StewartTrinkleProx::getName() const
{
    static std::string sName("Stewart-Trinkle_Prox_Function_Time_Stepper");
    return sName;
}

/// Retrieve the engine version we're going to expect, to prevent
/// loading of outdated plugins
/// @return the version of the engine when this plugin was compiled.
extern "C" ST_PLUGIN_API int getEngineVersion()
{
    return DvcEngineVersion;
}

/// Register this plugin to dvc.
/// @param K the handle to dvc allowing the plugin to register
/// @param pluginName The name of this plugin
extern "C" ST_PLUGIN_API void registerPlugin(DvcKernel &K, const std::string & pluginName)
{
    StewartTrinkleProx* ptr = new StewartTrinkleProx();
  ptr->SetPluginName(pluginName);
    K.getTimeStepperServer().addTimeStepper(ptr);
}
```

B.2 Prox Solvers

B.2.1 CPSolverServer.h

The relevant section of code is:

```
    /**
    * Class Prox_Solver. Technically not a CP_Solver, but easier to treat it as one
    */


   // *** Prox_Solver Class
   class Prox_Solver : public CP_Solver
   {
   public:
     DVC_ENGINE_API virtual ~Prox_Solver()
     {}

     // Configure settings for prox solver (so far, only stepsize)
     DVC_ENGINE_API virtual void initProx(double m_stepSize, double m_coeffFriction, double
          m_simTime, double m_gravity) {stepSize = m_stepSize; coeffFriction = m_coeffFriction;
          simTime = m_simTime; gravity = m_gravity;}
     // Populate Data Structures (Will be very different between CPU/GPU setups)
     DVC_ENGINE_API virtual bool populateProxDataStructure(const std::list <DynamicalBody *> *
          m_dynamicalBodies,
 const std::list<DvcCollisionResultPtr> *m_collisions,
 size_t m_numConstrainedBodies, size_t m_numContacts) {return false;}
     // Used as primary solver function
     DVC_ENGINE_API virtual bool solveProx() {return false;}

     // Need to put this function in here, since Prox_Solvers can use different data sturctures
     //   and building intermediary data structures would be wasteful
     // Given an index for a body, will return the bodies updated velocity. This can then be
          used by the timestepper to update the system
     DVC_ENGINE_API virtual bool getProxBodyNu(int index, DVC::Vector<REAL> *nu) {return false
          ;}
   protected:
     double stepSize; // Time between timesteps
     double coeffFriction; // Coefficient of friction
     double simTime; // Current timestep
     double gravity; // World gravity
     size_t numContacts;
     size_t numConstrainedBodies;
   };
```

B.2.2 ProxSerial

B.2.2.1 proxSerialDatastruct.h

```
//#define SINGLE_MODE
//#define DOUBLE_MODE

#ifdef SINGLE_MODE
typedef float FULL;
//#define ALIGNMENT_SIZE 16
#else
typedef double FULL;
#endif
     // *** Data types for prox_solver
  struct normalProx_t {
    FULL Gn1[3]; // Normal info from G_n for body 1
    FULL Gn2[3]; // Normal info from G_n for body 2
    //double pos[3]; // X,Y,Theta position (unneeded?)
    FULL gap_n;
    //FULL stepsize; // Redundant? (Should probably be passed to GPU Kernel via argument)
    FULL r;
    FULL p_n;
    //FULL delassus[3]; // Probably unneeded?
    //double rho_n_const; // The parts of rho_n that don't change during timesteps
    FULL convergeError;
    int b1index; // wrenchRowIndex / 3 of b1
    int b2index; // wrenchRowIndex / 3 of b2
    bool converged;
  };
  struct proxBody_t {
    FULL nu[3]; // Velocity at l
    FULL nu_lp1[3]; // Velocity at l+1
    FULL p_ext[3]; // Pext
    FULL mass;
    FULL mInertia; // moment of inertia
    //double m_inv; // = 1/mass
    //double i_inv; // = 1/mInertia
  };
  struct frictionProx_t {
    FULL Gf1[3]; // Normal info from G_n for body 1
    FULL Gf2[3]; // Normal info from G_n for body 2
    //double pos[3]; // X,Y,Theta position (unneeded?)
    //FULL gap_n;
    //FULL stepsize; // Redundant? (Should probably be passed to GPU Kernel via argument)
    FULL r;
    FULL p_f;
    //FULL delassus[3]; // Probably unneeded?
    //double rho_n_const; // The parts of rho_n that don't change during timesteps
    FULL convergeError;
    int b1index; // wrenchRowIndex / 3 of b1
    int b2index; // wrenchRowIndex / 3 of b2
    bool converged;
  };
```

B.2.2.2 proxSerial.h

```
/** @file plugins/proxSolver
 *
 * This file contains
 * @author: Jeremy Betz
 * @ingroup: proxSolver
 */

#ifndef _PROX_SERIAL_PLUGIN_H
#define _PROX_SERIAL_PLUGIN_H

#include <string>
#include "DvcKernel.h"
#include "proxSerialDatastruct.h"

#ifdef WIN32
#ifdef PROX_SERIAL_PLUGIN_EXPORTS
  #define PROX_SERIAL_PLUGIN_API __declspec(dllexport)
#else
  #define PROX_SERIAL_PLUGIN_API __declspec(dllimport)
#endif
//Linux
#else
  #define PROX_SERIAL_PLUGIN_API
#endif

using namespace DVC;

/**
 * class Prox_Serial
 * @ingroup: Prox_Serial
 */
```

```
class Prox_Serial : public Prox_Solver {
public:

  Prox_Serial();
  virtual ~Prox_Serial();
  const std::string &getName() const; // (Should figure out a way to check if a plugin is prox
      based using name)
  virtual void refresh(const DVC::PrefMap& prefs); // Data structures built here (so they don't
      need to be rebuilt every timestep)

  // FROM Prox_Solver
  bool populateProxDataStructure(const std::list <DynamicalBody *> *m_dynamicalBodies,
  const std::list<DvcCollisionResultPtr> *m_collisions,
  size_t m_numConstrainedBodies, size_t m_numContacts);
  bool solveProx(); // Interface from Prox_Solver

  bool getProxBodyNu(int index, DVC::Vector<REAL> *nu);

private:
  //for populateProxDataStructure:
  virtual void initProxBodies(const std::list <DynamicalBody *> *m_dynamicalBodies);
  virtual void formNormalProx(const std::list<DvcCollisionResultPtr> *m_collisions);
  virtual void formFrictionProx(const std::list<DvcCollisionResultPtr> *m_collisions);
  //for solveProx:
  virtual bool calculateR();
    //for calculateR:
    virtual bool calculateRichardson();
    virtual bool calculateJacobi();
  // Solve the prox function
  virtual bool solveLCPProx(bool friction);
    // Solve the normal componenet of the prox function
    virtual bool solveLCPProxNormal(int index);
    // Solve the frictional componenet of the prox function
    virtual bool solveLCPProxFriction(int index);
    // if sim diverges, reset to start point
    virtual void resetTimestep();
  // Take impulses from last timestep, and apply them to bodies
  virtual bool updateBodyNu(); //(can be overwritten with GPU code?)
  // Check the collision datastructure for convergence
  virtual bool checkConverge(); // Will probably be done much differently with GPU code
  //virtual bool solveLCPProx(); // Solve one iteration for inter-body impulses
  normalProx_t *normalProx;
  proxBody_t *proxBody;
  frictionProx_t *frictionProx;
  //double simTime;

  // Values from setup (refresh function/prefmap)
  unsigned int maxBodies;
  unsigned int maxContacts;
  int solveMode;
  FULL relaxCoeff; // also known as omega
  int debug;
  unsigned int maxIters;
  FULL p_nLimit;
  bool overvalue;
  FULL relaxCoeffOrig;
  FULL maxItersOrig;
};

#endif // _PROX_SERIAL_PLUGIN_H
```

B.2.2.3 proxSerial.cpp

```
/** @defgroup *** solver plugin
```

```
 *  @ingroup cpsolver
 *  Group *** is a subgroup of cpsolver
 */

/** @file plugins/.cpp
 * The *** solver plugin
 * This file contains the *** solver plugin implementation
 * @author: Jeremy Betz
 * @ingroup:
 */

#include "proxSerial.h"
#include <string>
#include <fstream>
#include <iostream>
#include <ForceController.h>

#define convergeNormal -1e-6
#define convergeFriction 1e-6
#define PN_LIMIT
#define p_nLimitDefault 1e5

/**
 * Default constructor.
 */
Prox_Serial::Prox_Serial()
{
  // Set values to defaults that will cause predictable breaking if not given proper values (in
       refresh() )
  proxBody = 0;
  normalProx = 0;
  maxBodies = 0;
  maxContacts = 0;
  solveMode = 0;
  relaxCoeff = 0;
  proxBody = NULL;
  normalProx = NULL;
  frictionProx = NULL;
  debug = 0;
}

Prox_Serial::~Prox_Serial()
{
  if (proxBody != NULL) {delete proxBody; proxBody = NULL;}
  if (normalProx != NULL) {delete normalProx; normalProx = NULL;}
  if (frictionProx != NULL) {delete frictionProx; frictionProx = NULL;}
}

void Prox_Serial::refresh( const PrefMap & prefs ) {


  printf("debug qq %d\n",sizeof(normalProx_t));
  std::string pluginPath = GetPluginPrefPath();

  maxBodies = prefs.get_int(pluginPath + "/maxBodies");
  if (proxBody != NULL) {delete proxBody;}
  if (maxBodies < 1){ maxBodies = 10; printf("Invalid number of maxBodies, setting to 10...\n")
       ;}
  proxBody = new proxBody_t[maxBodies];

  maxContacts = prefs.get_int(pluginPath + "/maxContacts");
  if (normalProx != NULL) {delete normalProx;}
  if (frictionProx != NULL) {delete frictionProx;}
  if (maxContacts < 1){ maxContacts = 100; printf("Invalid number of maxContacts, setting to 
      100...\n");}
  normalProx = new normalProx_t[maxContacts];
  frictionProx = new frictionProx_t[maxContacts];

  // should be changed to use enum
  std::string solveStr = prefs.get_string(pluginPath + "/solveMode");
  if (solveStr == "richardson")
    solveMode = 1;
  else if (solveStr == "jacobi")
    solveMode = 2;
  else if (solveStr == "gauss_seidel")
    solveMode = 3;
  else {
    solveMode = 1;
    printf("Invalid solveMode, using richardson...\n");
  }

  overvalue = false;

  maxIters = prefs.get_int(pluginPath + "/maxIters");
  if (maxIters < 1){ maxIters = 500; printf("Invalid number of maxIters, setting to 500...\n");}
  maxItersOrig = maxIters;
  debug = prefs.get_int(pluginPath + "/debug");

  relaxCoeff = prefs.get_double(pluginPath + "/relaxCoeff"); // also known as omega
  if (relaxCoeff == 0) {printf("Invalid relaxCoeff, setting to 1...\n"); relaxCoeff = 1; } //
      relaxCoeff can't be 0, so set to a default of 1
  relaxCoeffOrig = relaxCoeff;

  p_nLimit = prefs.get_double(pluginPath + "/p_nLimit");
  //if no limit, will default to 0, and not be used
}

bool Prox_Serial::populateProxDataStructure(const std::list< DynamicalBody* >* m_dynamicalBodies
    ,
              const std::list< DvcCollisionResultPtr >* m_collisions,
              size_t m_numConstrainedBodies, size_t m_numContacts)
{
  numConstrainedBodies = m_numConstrainedBodies;
  numContacts = m_numContacts;

  // Populate the datastructure for the bodies
  initProxBodies(m_dynamicalBodies);
  // Populate the datastructure for the potential collision sites
  formNormalProx(m_collisions);
  if (coeffFriction != 0.0){
    formFrictionProx(m_collisions);
  }
  return true;
}

void Prox_Serial::initProxBodies(const std::list< DynamicalBody* >* m_dynamicalBodies)
{
  DVC::Vector<REAL> newForce(3);
  DVC::Vector<REAL> newForceSum(3);
  std::list <DynamicalBody *>::const_iterator bodyItr = m_dynamicalBodies->begin();
  for ( ; bodyItr != m_dynamicalBodies->end(); ++bodyItr )
  {
      DynamicalBody *b = *bodyItr;
      if( !(b->IsConstrained()) ) // Body in freefall
      { // free fall body, not formulated in ST LCP
    continue;
      }
```

```
        newForceSum.clear();
        unsigned int rowIndex = b->GetWrenchRowIndex()/3;
        // Init default values
        proxBody[rowIndex].p_ext[0] = 0;
        proxBody[rowIndex].p_ext[1] = 0;
        proxBody[rowIndex].p_ext[2] = 0;
        if (b->IsForceControlled())
        { // sum up the contributing forces from each force controller
      const std::list <const ForceController *>& forceCtrls = b->GetForceControllers();
      std::list <const ForceController *>::const_iterator itr = forceCtrls.begin();
      for( ; itr != forceCtrls.end(); ++itr){
        newForce.clear();
        (*itr)->GetForce(simTime, newForce, b->GetQ(), b->GetNu() );
        newForceSum += newForce;
      }
      // add the impulses from the controllers
      proxBody[rowIndex].p_ext[0] = stepSize*newForceSum[0];
      proxBody[rowIndex].p_ext[1] = stepSize*newForceSum[1];
      proxBody[rowIndex].p_ext[2] = stepSize*newForceSum[2];
        }

        // add the gravitational force
        proxBody[rowIndex].p_ext[1] -= gravity * stepSize * b->GetMass() ;
        // Get other info for prox solver
        proxBody[rowIndex].mass = b->GetMass();
        proxBody[rowIndex].mInertia = b->GetMomentInertia();
        proxBody[rowIndex].nu[0] = b->GetNu()(0);
        proxBody[rowIndex].nu[1] = b->GetNu()(1);
        proxBody[rowIndex].nu[2] = b->GetNu()(2);
  }
}

void Prox_Serial::formNormalProx(const std::list< DvcCollisionResultPtr >* m_collisions)
{
  int i = 0;

  DVC::Vector<REAL> n(2), r(2), Gn_vec(3);

  std::list<DvcCollisionResultPtr>::const_iterator contactItr;
  for (contactItr = m_collisions->begin(); contactItr != m_collisions->end(); ++contactItr )
  {
      const DvcCollisionResultPtr &colRes = *contactItr;

      // Get the two bodies in contact.
      Body * m1 = colRes->b1;
      Body * m2 = colRes->b2;

      // Sanity Check: Should never be collision checking between 2 static objects
      assert( ! ( m1->GetBodyType()==BODY_OBSTACLE && m2->GetBodyType()==BODY_OBSTACLE ) );

      // Get the normal and tangential information
      n(0) = colRes->normalB1toB2[0];
      n(1) = colRes->normalB1toB2[1];

      // Set Timestep size (highly redundant information, remove later?)
      //normalProx[i].stepsize = stepSize;
      normalProx[i].convergeError = 10000;
      normalProx[i].converged = false;
      // Init p_n to 0
      normalProx[i].p_n = 0;
      // Get each body's row location in the wrench and each body's position
      if ( m1->GetBodyType()==BODY_DYNAMICAL )
      { // Only in the wrench if it is not an obstacle
```

```
    //M1RowIndex = static_cast< const DynamicalBody * > ( m1 )->GetWrenchRowIndex();
    const DVC::Vector<REAL> &M1Pos = static_cast< const DynamicalBody * > ( m1 )->GetQ();
    //const DVC::Vector<REAL> &M1Nu = static_cast< const DynamicalBody * > ( m1 )->GetNu();

    r(0) = colRes->b1ContactLoc[0] - M1Pos[0];
    r(1) = colRes->b1ContactLoc[1] - M1Pos[1];
    assert(r(0) == r(0) && r(1) == r(1)  );
    normalProx[i].Gn1[0] = n(0);
    normalProx[i].Gn1[1] = n(1);
    normalProx[i].Gn1[2] = r.cross2D(n);
    normalProx[i].gap_n = colRes->distance;

    /*Gn_vec(0) = normalProx[i].Gn[0]; Gn_vec(1) = normalProx[i].Gn[1]; Gn_vec(2) = normalProx[i
        ].Gn[2];
    normalProx[i].Nu_n = Gn_vec.dot(M1Nu);
    normalProx[i].mass = static_cast< const DynamicalBody * > ( m1 )->GetMass();
    normalProx[i].inertia = static_cast< const DynamicalBody * > ( m1 )->GetMomentInertia(); */
    normalProx[i].b1index = static_cast< const DynamicalBody * > ( m1 )->GetWrenchRowIndex()/3;
      }
      else normalProx[i].b1index = -1;

      if ( m2->GetBodyType()==BODY_DYNAMICAL )
      { // Only in the wrench if it is not an obstacle
    //M2RowIndex = static_cast< const DynamicalBody * > ( m2 )->GetWrenchRowIndex();
    const DVC::Vector<REAL> &M2Pos = static_cast< const DynamicalBody * > ( m2 )->GetQ();
    //const DVC::Vector<REAL> &M2Nu = static_cast< const DynamicalBody * > ( m2 )->GetNu();

    r(0) = colRes->b2ContactLoc[0] - M2Pos[0];
    r(1) = colRes->b2ContactLoc[1] - M2Pos[1];
    assert(r(0) == r(0) && r(1) == r(1)  );
    n(0) = -n(0);
    n(1) = -n(1);
    normalProx[i].Gn2[0] = n(0);
    normalProx[i].Gn2[1] = n(1);
    normalProx[i].Gn2[2] = r.cross2D(n);
    normalProx[i].gap_n = colRes->distance;

    /*Gn_vec(0) = normalProx[i].Gn[0]; Gn_vec(1) = normalProx[i].Gn[1]; Gn_vec(2) = normalProx[i
        ].Gn[2];
    normalProx[i].Nu_n = Gn_vec.dot(M2Nu);
    normalProx[i].mass = static_cast< const DynamicalBody * > ( m2 )->GetMass();
    normalProx[i].inertia = static_cast< const DynamicalBody * > ( m2 )->GetMomentInertia();*/
    normalProx[i].b2index = static_cast< const DynamicalBody * > ( m2 )->GetWrenchRowIndex()/3;
      }
      else normalProx[i].b2index = -1;
      //++ colIndex; // Next column
      i++; //Position in array
  }
  return;
}

void Prox_Serial::formFrictionProx(const std::list< DvcCollisionResultPtr >* m_collisions)
{
  int i = 0;

  DVC::Vector<REAL> n(2), r(2), Gf_vec(3);

  std::list<DvcCollisionResultPtr>::const_iterator contactItr;
  for (contactItr = m_collisions->begin(); contactItr != m_collisions->end(); ++contactItr )
  {
      const DvcCollisionResultPtr &colRes = *contactItr;

      // Get the two bodies in contact.
      Body * m1 = colRes->b1;
```

```
        Body * m2 = colRes->b2;

        // Sanity Check: Should never be collision checking between 2 static objects
        assert( ! ( m1->GetBodyType()==BODY_OBSTACLE && m2->GetBodyType()==BODY_OBSTACLE ) );

        // Get the normal and tangential information
        //(x' = y; y' = -x)
        n(0) = colRes->normalB1toB2[1];
        n(1) = -(colRes->normalB1toB2[0]);

        // Set Timestep size (highly redundant information, remove later?)

        frictionProx[i].convergeError = 10000;
        frictionProx[i].converged = false;

        // Get each body's row location in the wrench and each body's position
        if ( m1->GetBodyType()==BODY_DYNAMICAL )
        { // Only in the wrench if it is not an obstacle
      //M1RowIndex = static_cast< const DynamicalBody * > ( m1 )->GetWrenchRowIndex();
      const DVC::Vector<REAL> &M1Pos = static_cast< const DynamicalBody * > ( m1 )->GetQ();
      //const DVC::Vector<REAL> &M1Nu = static_cast< const DynamicalBody * > ( m1 )->GetNu();

      r(0) = colRes->b1ContactLoc[0] - M1Pos[0];
      r(1) = colRes->b1ContactLoc[1] - M1Pos[1];
      assert(r(0) == r(0) && r(1) == r(1)  );
      frictionProx[i].Gf1[0] = n(0);
      frictionProx[i].Gf1[1] = n(1);
      frictionProx[i].Gf1[2] = r.cross2D(n);
      frictionProx[i].b1index = static_cast< const DynamicalBody * > ( m1 )->GetWrenchRowIndex()
          /3;
        }
        else frictionProx[i].b1index = -1;

        if ( m2->GetBodyType()==BODY_DYNAMICAL )
        { // Only in the wrench if it is not an obstacle
      //M2RowIndex = static_cast< const DynamicalBody * > ( m2 )->GetWrenchRowIndex();
      const DVC::Vector<REAL> &M2Pos = static_cast< const DynamicalBody * > ( m2 )->GetQ();
      //const DVC::Vector<REAL> &M2Nu = static_cast< const DynamicalBody * > ( m2 )->GetNu();

      r(0) = colRes->b2ContactLoc[0] - M2Pos[0];
      r(1) = colRes->b2ContactLoc[1] - M2Pos[1];
      assert(r(0) == r(0) && r(1) == r(1)  );
      n(0) = -n(0);
      n(1) = -n(1);
      frictionProx[i].Gf2[0] = n(0);
      frictionProx[i].Gf2[1] = n(1);
      frictionProx[i].Gf2[2] = r.cross2D(n);
      frictionProx[i].b2index = static_cast< const DynamicalBody * > ( m2 )->GetWrenchRowIndex()
          /3;
        }
        else frictionProx[i].b2index = -1;
        //++ colIndex; // Next column



        i++; //Position in array
    }
    return;
}



bool Prox_Serial::solveProx()
{
  //reset back to defaults (more unstable, but faster)
  //maxIters = maxItersOrig;
  //relaxCoeff = relaxCoeffOrig;
  calculateR(); // Calculate values, using method specified in XML
  updateBodyNu(); // Update the bodies velocity to initialize nu_lp1

  bool friction = (coeffFriction != 0.0); // compare friction to double once, save as boolean (
       probably more efficient?)
  int k = 0;
  do {
    //debugging code
    k++;
    if (!(k%5) && (debug==2)){
      printf("%d_th_loop\n",k);
    }
    solveLCPProx(friction);
    if (overvalue && p_nLimit){ // if p_nLimit set and went overvalue
      resetTimestep();
      k = 0;
      // halve the relaxCoeff to increase stability
      relaxCoeff = relaxCoeff/2;
      // double the maxIters to compensate for increased solution timestep
      maxIters = maxIters*2;
      printf("System_divergence_detected_at_%f_on_iter_%d,_setting_relaxCoeff_to_%f\n",simTime,k
           ,relaxCoeff);
    }
    updateBodyNu();
  } while ((!checkConverge()) && (k<maxIters)); // While it hasn't converged and k is less than
       the max allowed iterations
  if (debug==3){
    printf("finished_at_%d_iters\n",k);
  }
  return true;
}

void Prox_Serial::resetTimestep()
{
  for(unsigned int i = 0; i < numContacts; i++){
    normalProx[i].p_n = 0;
    normalProx[i].converged = false;
  }
  if (coeffFriction != 0.0){
      for(unsigned int i = 0; i < numContacts; i++){
      frictionProx[i].p_f = 0;
      frictionProx[i].converged = false;
    }
  }
  calculateR();
  overvalue = false;
}


bool Prox_Serial::getProxBodyNu(int index, DVC::Vector<REAL> *nu)
{

  (*nu)[0] = proxBody[index].nu_lp1[0];
  (*nu)[1] = proxBody[index].nu_lp1[1];
  (*nu)[2] = proxBody[index].nu_lp1[2];
  return true;
}

bool Prox_Serial::solveLCPProxNormal(int index)
{
  REAL p_n_star = 0;
  REAL rho = 0;

  for(int j=0;j<1;j++){ // don't need now
    p_n_star = 0;
    rho = 0;
    if (normalProx[index].b1index != -1){
      rho += normalProx[index].Gn1[0]*proxBody[normalProx[index].b1index].nu_lp1[0]
    + normalProx[index].Gn1[1]*proxBody[normalProx[index].b1index].nu_lp1[1]
    + normalProx[index].Gn1[2]*proxBody[normalProx[index].b1index].nu_lp1[2];

    }
    if (normalProx[index].b2index != -1){
      rho += normalProx[index].Gn2[0]*proxBody[normalProx[index].b2index].nu_lp1[0]
    + normalProx[index].Gn2[1]*proxBody[normalProx[index].b2index].nu_lp1[1]
    + normalProx[index].Gn2[2]*proxBody[normalProx[index].b2index].nu_lp1[2];
    }

    rho += normalProx[index].gap_n/stepSize;
    // rho += normalProx[index].dGapdTime // If the second object is a kinematic body

    if ((rho > convergeNormal) && (normalProx[index].converged == true)){ // Check for
        convergence
      // If was converged last iteration, just break loop
      //printf("%d rho in tolerance, breaking\n",index); ###
      break;// Otherwise, set to true and continue to save p_n
    }
    else if (rho > convergeNormal){
      normalProx[index].converged = true;
    }
    else normalProx[index].converged = false;


    // RHO FINISHED CALCULATING
    // CALCULATE P_N_STAR
    p_n_star = normalProx[index].p_n - normalProx[index].r*rho;


    // Enforce prox condition (Lambda >= 0)
    if (p_n_star < 0)
      p_n_star = 0;

#ifdef PN_LIMIT
    if (p_nLimit){ // if p_nLimit is set, then run check
      if (p_n_star > p_nLimit){
  overvalue = true;
      }
    }
#endif

    // Calculate convergeError
    REAL cE = p_n_star - normalProx[index].p_n;
    if ((cE - normalProx[index].convergeError) > 1e-6){
  //printf("%d cE increasing %.10f rate %.10f\n",index,cE, (cE - normalProx[index].convergeError
      ));
  //normalProx[index].r[0] = normalProx[index].r[0]/2;
    }
    // Compare chage of convergeError? (cE vs convergeError)
    if ((debug==2)){
      //printf("%dth contact. rho %.7f  P_n_new %.5f Error %.10f dError %.10f\n",index,rho,
          p_n_star,cE,cE-normalProx[index].convergeError);
    }
    if (debug == 1){ // Print convergence data in easy to use format
      //printf("%d, %.5f, %.10f, %.10f, %.10f\n", index, normalProx[index].r, normalProx[index].
          p_n, p_n_star, cE-normalProx[index].convergeError);
    }

    // Save new convergeError
    normalProx[index].convergeError = cE;
    // Save new p_n
    normalProx[index].p_n = p_n_star;
    //printf("%f\n", normalProx[index].p_n);
  }

  return true;
}

// NOT WRITTEN YET ??? !!! ###
bool Prox_Serial::solveLCPProxFriction(int index)
{
  REAL p_f_star = 0;
  REAL rho = 0;

  if (normalProx[index].p_n == 0.0){ // if no normal force, then friction = 0
    frictionProx[index].p_f = 0;
    frictionProx[index].converged = true;
    return true;
  }

  for(int j=0;j<1;j++){ // don't need now
    p_f_star = 0;
    rho = 0;
    if (frictionProx[index].b1index != -1){
      rho += frictionProx[index].Gf1[0]*proxBody[frictionProx[index].b1index].nu_lp1[0]
    + frictionProx[index].Gf1[1]*proxBody[frictionProx[index].b1index].nu_lp1[1]
    + frictionProx[index].Gf1[2]*proxBody[frictionProx[index].b1index].nu_lp1[2];

    }
    if (frictionProx[index].b2index != -1){
      rho += frictionProx[index].Gf2[0]*proxBody[frictionProx[index].b2index].nu_lp1[0]
    + frictionProx[index].Gf2[1]*proxBody[frictionProx[index].b2index].nu_lp1[1]
    + frictionProx[index].Gf2[2]*proxBody[frictionProx[index].b2index].nu_lp1[2];
    }

    //rho += frictionProx[index].gap_f/stepSize;
    // rho += frictionProx[index].dGapdTime // If the second object is a kinematic body
    /*
    if ((abs(rho) < convergeFriction) && (frictionProx[index].converged == true)){ // Check for
        convergence
      // If was converged last iteration, just break loop
      //printf("%d rho in tolerance, breaking\n",index); ###
      break;// Otherwise, set to true and continue to save p_f
    }
    else*/ if (abs(rho) < convergeFriction){
      frictionProx[index].converged = true;
    }
    else frictionProx[index].converged = false;


    // RHO FINISHED CALCULATING
    // CALCULATE p_f_STAR
    p_f_star = frictionProx[index].p_f - frictionProx[index].r*rho;


    // Enforce prox condition
    if (p_f_star < -(coeffFriction*normalProx[index].p_n)){
      p_f_star = -(coeffFriction*normalProx[index].p_n);
      frictionProx[index].converged = true;
    }
    else if (p_f_star > (coeffFriction*normalProx[index].p_n)) {
      p_f_star = (coeffFriction*normalProx[index].p_n);
      frictionProx[index].converged = true;
    }
    // Calculate convergeError
    REAL cE = p_f_star - frictionProx[index].p_f;
    if ((cE - frictionProx[index].convergeError) > 1e-6){
  //printf("%d cE increasing %.10f rate %.10f\n",index,cE, (cE - frictionProx[index].
      convergeError));
  //frictionProx[index].r[0] = frictionProx[index].r[0]/2;
    }
    // Compare chage of convergeError? (cE vs convergeError)
    if ((debug==2)){
      //printf("%dth contact. rho %.7f  p_f_new %.5f Error %.10f dError %.10f\n",index,rho,
          p_f_star,cE,cE-frictionProx[index].convergeError);
    }
    if (debug == 1){ // Print convergence data in easy to use format
      //printf("%d, %.5f, %.10f, %.10f, %.10f\n", index, frictionProx[index].r, frictionProx[
          index].p_f, p_f_star, cE-frictionProx[index].convergeError);
    }

    // Save new convergeError
    frictionProx[index].convergeError = cE;
    // Save new p_f
    frictionProx[index].p_f = p_f_star;
    //printf("%f\n", frictionProx[index].p_f);
  }


  return false;
}


bool Prox_Serial::solveLCPProx(bool friction)
{
  /*if (debug==2){
    printf("new timestep %f\n", simTime);
  }*/


  for(unsigned  int i=0;i<numContacts;i++){
    solveLCPProxNormal(i);
    if (friction){
      solveLCPProxFriction(i);
    }
    //printf("next\n");
  }
  return true;
}

bool Prox_Serial::updateBodyNu()
{
    for(unsigned int i = 0; i < numConstrainedBodies; i++){
    proxBody[i].nu_lp1[0] = 0;
    proxBody[i].nu_lp1[1] = 0;
    proxBody[i].nu_lp1[2] = 0;
  }
  //nu l+1 += loop of collisions
  for(unsigned int i = 0; i < numContacts; i++){ //Note: nu_lp1 is used as value of applied
      impulse, not velocity (mass and mInertia not used yet)
    if (normalProx[i].b1index != -1){
      proxBody[normalProx[i].b1index].nu_lp1[0] += normalProx[i].Gn1[0]*normalProx[i].p_n;
      proxBody[normalProx[i].b1index].nu_lp1[1] += normalProx[i].Gn1[1]*normalProx[i].p_n;
      proxBody[normalProx[i].b1index].nu_lp1[2] += normalProx[i].Gn1[2]*normalProx[i].p_n;
    }
    if (normalProx[i].b2index != -1){
      proxBody[normalProx[i].b2index].nu_lp1[0] += normalProx[i].Gn2[0]*normalProx[i].p_n;
      proxBody[normalProx[i].b2index].nu_lp1[1] += normalProx[i].Gn2[1]*normalProx[i].p_n;
      proxBody[normalProx[i].b2index].nu_lp1[2] += normalProx[i].Gn2[2]*normalProx[i].p_n;
    }
  }
  if (coeffFriction != 0.0){
    for(unsigned int i = 0; i < numContacts; i++){ //Note: nu_lp1 is used as value of applied
         impulse, not velocity (mass and mInertia not used yet)
      if (frictionProx[i].b1index != -1){
  proxBody[frictionProx[i].b1index].nu_lp1[0] += frictionProx[i].Gf1[0]*frictionProx[i].p_f;
  proxBody[frictionProx[i].b1index].nu_lp1[1] += frictionProx[i].Gf1[1]*frictionProx[i].p_f;
  proxBody[frictionProx[i].b1index].nu_lp1[2] += frictionProx[i].Gf1[2]*frictionProx[i].p_f;
      }
      if (frictionProx[i].b2index != -1){
  proxBody[frictionProx[i].b2index].nu_lp1[0] += frictionProx[i].Gf2[0]*frictionProx[i].p_f;
  proxBody[frictionProx[i].b2index].nu_lp1[1] += frictionProx[i].Gf2[1]*frictionProx[i].p_f;
  proxBody[frictionProx[i].b2index].nu_lp1[2] += frictionProx[i].Gf2[2]*frictionProx[i].p_f;
      }
    }
  }
  // nu l+1 += pext all over m + nu l
  for(unsigned int i = 0; i < numConstrainedBodies; i++){
    proxBody[i].nu_lp1[0] = proxBody[i].nu[0] + ((proxBody[i].p_ext[0] + proxBody[i].nu_lp1[0])/
         proxBody[i].mass);
    proxBody[i].nu_lp1[1] = proxBody[i].nu[1] + ((proxBody[i].p_ext[1] + proxBody[i].nu_lp1[1])/
         proxBody[i].mass);
    proxBody[i].nu_lp1[2] = proxBody[i].nu[2] + ((proxBody[i].p_ext[2] + proxBody[i].nu_lp1[2])/
         proxBody[i].mInertia);
  }

  return true;
}

bool Prox_Serial::calculateR()
{
  if (solveMode == 1) { // Richardson
    calculateRichardson();
  }
  else if (solveMode == 2) { // Jacobi
    calculateJacobi();
  }
  else if (solveMode == 3) { // Gauss-Seidel
    printf("Gauss-Seidel_not_yet_implemented,_probably_going_to_crash...\n");
    return false;
  }
  return true;
}

bool Prox_Serial::calculateRichardson()
{
  for (unsigned int i = 0; i < numContacts; i++){
    normalProx[i].r = relaxCoeff;
  }
  if (coeffFriction != 0.0){
    for (unsigned int i = 0; i < numContacts; i++){
      frictionProx[i].r = relaxCoeff;
    }
  }
}
```

```
bool Prox_Serial::calculateJacobi()
{
  for (unsigned int i = 0; i < numContacts; i++){
    FULL delassus = 0;
    if (normalProx[i].b1index != -1){
      delassus += (normalProx[i].Gn1[0]*normalProx[i].Gn1[0])/proxBody[normalProx[i].b1index].
          mass +
        (normalProx[i].Gn1[1]*normalProx[i].Gn1[1])/proxBody[normalProx[i].b1index].mass +
        (normalProx[i].Gn1[2]*normalProx[i].Gn1[2])/proxBody[normalProx[i].b1index].mInertia;
    }
    if (normalProx[i].b2index != -1){
      delassus +=(normalProx[i].Gn2[0]*normalProx[i].Gn2[0])/proxBody[normalProx[i].b2index].
          mass +
        (normalProx[i].Gn2[1]*normalProx[i].Gn2[1])/proxBody[normalProx[i].b2index].mass +
        (normalProx[i].Gn2[2]*normalProx[i].Gn2[2])/proxBody[normalProx[i].b2index].mInertia;
    }
    normalProx[i].r = relaxCoeff/delassus;
  }

  if (coeffFriction != 0.0){
    for (unsigned int i = 0; i < numContacts; i++){
      FULL delassus = 0;
      if (frictionProx[i].b1index != -1){
  delassus += (frictionProx[i].Gf1[0]*frictionProx[i].Gf1[0])/proxBody[frictionProx[i].b1index].
      mass +
          (frictionProx[i].Gf1[1]*frictionProx[i].Gf1[1])/proxBody[frictionProx[i].b1index].mass
               +
          (frictionProx[i].Gf1[2]*frictionProx[i].Gf1[2])/proxBody[frictionProx[i].b1index].
              mInertia;
      }
      if (frictionProx[i].b2index != -1){
  delassus +=(frictionProx[i].Gf2[0]*frictionProx[i].Gf2[0])/proxBody[frictionProx[i].b2index].
      mass +
          (frictionProx[i].Gf2[1]*frictionProx[i].Gf2[1])/proxBody[frictionProx[i].b2index].mass
               +
          (frictionProx[i].Gf2[2]*frictionProx[i].Gf2[2])/proxBody[frictionProx[i].b2index].
              mInertia;
      }
      frictionProx[i].r = relaxCoeff/delassus;
    }
  }
  return true;
}


bool Prox_Serial::checkConverge() //return true if system reached convergence
{
  bool friction = (coeffFriction != 0.0);
    for(unsigned  int i=0;i<numContacts;i++){ // Check for total convergence
      if ((normalProx[i].converged == false) || ((frictionProx[i].converged == false) &&
          friction)){
   return false;
      }
    }
    return true; // If didn't return false yet, then must be converged
}



const std::string & Prox_Serial::getName() const {
      static std::string sName("Prox_Serial_Solver");
      return sName;
}
```

```
///// The following methods are required for all plugins /////////
/// Retrieve the engine version we're going to expect
extern "C" PROX_SERIAL_PLUGIN_API int getEngineVersion() {
  return DvcEngineVersion;
}

/// Tells us to register our functionality to an engine kernel
extern "C" PROX_SERIAL_PLUGIN_API void registerPlugin(DvcKernel &K, const std::string &
     pluginName) {

   Prox_Serial *ptr = new Prox_Serial();
   ptr->SetPluginName(pluginName);
  K.getCP_SolverServer().AddCP_Solver(ptr);

}
```

B.2.3 ProxCUDA

B.2.3.1 proxCUDAInterface.h

```
#pragma once

//Datastructures (prototypes, fully defined in proxCUDAKernels.cuh)
#define SINGLE_MODE
//#define DOUBLE_MODE

#ifdef SINGLE_MODE
typedef float FULL;
//#define ALIGNMENT_SIZE 16
#else
typedef double FULL;
#endif

// Structures for each collision point
struct proxG_t {  // For Gn and Gf
//Arrays [4] for allignment
  FULL G1[4];
  FULL G2[4];
};

struct proxData_t {
  FULL r;
  FULL p;
  int b1index;
  int b2index;
  FULL gap;
};

// Structures for each body
struct proxNu_t { // For nu and nu_lp1
  FULL nu[4];
};

struct proxBodyExternal_t {
  FULL pExt[4];
};

struct proxBodyConsts_t {
  FULL mass;
  FULL mInertia;
};

struct results_t {
  int numIters;
  bool overvalue;
  int error;
  bool globalConverge;
};

struct bodyPointers {
  proxNu_t *proxNu;
  proxNu_t *proxNu_lp1;
  proxBodyExternal_t *proxBodyExternal;
  proxBodyConsts_t *proxBodyConsts;
};

struct contactPointers {
  proxG_t *proxGn;
  proxG_t *proxGf;
  proxData_t *proxContactN;
  proxData_t *proxContactF;
  bool *convergedN;
  bool *convergedF;
};
//struct numbers_t;
extern "C" void runTestCuda();

extern "C" bool InitCUDA(void);

//delete mem found on device(GPU)
extern "C" bool deleteCUDAMem(bodyPointers *proxBody_d, contactPointers *proxContact_d);
//Delete memory found on host (CPU)
extern "C" bool deleteHostMem(bodyPointers *proxBody_h, contactPointers *proxContact_h);

//delete result struct
extern "C" bool deleteResult(results_t *&results_h, results_t *&results_d);
//allocate strusts for basic comm with kernel
extern "C" bool allocateResult(results_t *&results_h, results_t *&results_d);

//Allocate memory relevant to bodies (host and device)
extern "C" bool allocateBodiesMem(int maxBodies,
          bodyPointers *proxBody_h,
          bodyPointers *proxBody_d
         );

//Allocate memory relevant to contacts (host and device)
extern "C" bool allocateContactsMem(int maxContacts,
            contactPointers *proxContact_h,
            contactPointers *proxContact_d
           );


//for populateProxDataStructure:

// Copy populated body structs to device
extern "C" bool populateProxBodiesCUDA(int numConstrainedBodies,
               bodyPointers *proxBody_h,
               bodyPointers *proxBody_d
              );
// Copy populated contact structs to device
extern "C" bool populateProxContactsCUDA(int numContacts,
           FULL coeffFriction,
           contactPointers *proxContact_h,
           contactPointers *proxContact_d
          );

//for solveProx:
```

```
// do all solving in nvcc (better control over GPU)
extern "C" bool solveLCPProx(int solveMode,
          FULL relaxCoeff,
          FULL coeffFriction,
          int maxIters,
          int numContacts,
          int numConstrainedBodies,
          contactPointers *proxContact_d,
          contactPointers *proxContact_h,
          bodyPointers *proxBody_d,
          bodyPointers *proxBody_h,
          results_t *results_h,
          results_t *results_d
         );
```

B.2.3.2 proxCUDAInterface.cu

```
#include "proxCUDAInterface.h"
#include "proxCUDAKernels.cuh"
//###
#include <stdio.h>

extern "C" bool deleteCUDAMem(bodyPointers *proxBody_d, contactPointers *proxContact_d){
  cudaFree(proxBody_d->proxNu);
  cudaFree(proxBody_d->proxNu_lp1);
  cudaFree(proxBody_d->proxBodyExternal);
  cudaFree(proxBody_d->proxBodyConsts);
  cudaFree(proxContact_d->convergedF);
  cudaFree(proxContact_d->convergedN);
  cudaFree(proxContact_d->proxContactF);
  cudaFree(proxContact_d->proxContactN);
  cudaFree(proxContact_d->proxGf);
  cudaFree(proxContact_d->proxGn);
  return true;
}

extern "C" bool deleteHostMem(bodyPointers *proxBody_h, contactPointers *proxContact_h){
  cudaFreeHost(proxBody_h->proxNu);
  cudaFreeHost(proxBody_h->proxNu_lp1);
  cudaFreeHost(proxBody_h->proxBodyExternal);
  cudaFreeHost(proxBody_h->proxBodyConsts);
  cudaFreeHost(proxContact_h->convergedF);
  cudaFreeHost(proxContact_h->convergedN);
  cudaFreeHost(proxContact_h->proxContactF);
  cudaFreeHost(proxContact_h->proxContactN);
  cudaFreeHost(proxContact_h->proxGf);
  cudaFreeHost(proxContact_h->proxGn);
  return true;

}

extern "C" bool deleteResult(results_t *&results_h, results_t *&results_d){
  cudaFreeHost(results_h);
  cudaFree(results_d);
  return true;
}

extern "C" bool allocateResult(results_t *&results_h, results_t *&results_d){
  cudaHostAlloc(&results_h,sizeof(results_t),cudaHostAllocDefault);
  cudaMalloc(&results_d,sizeof(results_t));
  return true;
}
```

```
extern "C" bool allocateBodiesMem(int maxBodies,
         bodyPointers *proxBody_h,
         bodyPointers *proxBody_d
        )
{
  //host (WC when proper)
  cudaHostAlloc(&proxBody_h->proxNu,sizeof(proxNu_t)*maxBodies,cudaHostAllocWriteCombined);
  cudaHostAlloc(&proxBody_h->proxNu_lp1,sizeof(proxNu_t)*maxBodies,cudaHostAllocDefault); //
      read from later
  cudaHostAlloc(&proxBody_h->proxBodyExternal,sizeof(proxBodyExternal_t)*maxBodies,
      cudaHostAllocWriteCombined);
  cudaHostAlloc(&proxBody_h->proxBodyConsts,sizeof(proxBodyConsts_t)*maxBodies,
      cudaHostAllocWriteCombined);
  //device
  cudaMalloc(&proxBody_d->proxNu,sizeof(proxNu_t)*maxBodies);
  cudaMalloc(&proxBody_d->proxNu_lp1,sizeof(proxNu_t)*maxBodies);
  cudaMalloc(&proxBody_d->proxBodyExternal,sizeof(proxBodyExternal_t)*maxBodies);
  cudaMalloc(&proxBody_d->proxBodyConsts,sizeof(proxBodyConsts_t)*maxBodies);
  printf("bodies allocated!!!\n");
  return true;
}

extern "C" bool allocateContactsMem(int maxContacts,
           contactPointers *proxContact_h,
           contactPointers *proxContact_d
          )
{
  //host (WC when proper)
  cudaHostAlloc(&proxContact_h->convergedF,sizeof(bool)*maxContacts,0);
  cudaHostAlloc(&proxContact_h->convergedN,sizeof(bool)*maxContacts,0);
  cudaHostAlloc(&proxContact_h->proxContactF,sizeof(proxData_t)*maxContacts,
      cudaHostAllocWriteCombined);
  cudaHostAlloc(&proxContact_h->proxContactN,sizeof(proxData_t)*maxContacts,
      cudaHostAllocWriteCombined);
  cudaHostAlloc(&proxContact_h->proxGf,sizeof(proxG_t)*maxContacts,cudaHostAllocWriteCombined);
  cudaHostAlloc(&proxContact_h->proxGn,sizeof(proxG_t)*maxContacts,cudaHostAllocWriteCombined);
  //device
  cudaMalloc(&proxContact_d->convergedF,sizeof(bool)*maxContacts);
  cudaMalloc(&proxContact_d->convergedN,sizeof(bool)*maxContacts);
  cudaMalloc(&proxContact_d->proxContactF,sizeof(proxData_t)*maxContacts);
  cudaMalloc(&proxContact_d->proxContactN,sizeof(proxData_t)*maxContacts);
  cudaMalloc(&proxContact_d->proxGf,sizeof(proxG_t)*maxContacts);
  cudaMalloc(&proxContact_d->proxGn,sizeof(proxG_t)*maxContacts);
    printf("contacts allocated!!!\n");
  return true;
}

// Copy populated body structs to device
extern "C" bool populateProxBodiesCUDA(int numConstrainedBodies,
             bodyPointers *proxBody_h,
             bodyPointers *proxBody_d
            )
{
  cudaMemcpyAsync(proxBody_d->proxBodyConsts,proxBody_h->proxBodyConsts,sizeof(proxBodyConsts_t)
      *numConstrainedBodies,cudaMemcpyHostToDevice);
  cudaMemcpyAsync(proxBody_d->proxBodyExternal,proxBody_h->proxBodyExternal,sizeof(
      proxBodyExternal_t)*numConstrainedBodies,cudaMemcpyHostToDevice);
  cudaMemcpyAsync(proxBody_d->proxNu,proxBody_h->proxNu,sizeof(proxNu_t)*numConstrainedBodies,
      cudaMemcpyHostToDevice);
  //nu_lp1 contains no valid data, no need to copy
  return true;
}
```

```
// Copy populated contact structs to device
extern "C" bool populateProxContactsCUDA(int numContacts,
           FULL coeffFriction,
           contactPointers *proxContact_h,
           contactPointers *proxContact_d
          )
{
  //convergedN has no useful data
  //convergedF has no useful data
  if (coeffFriction != 0.0){ // only copy friction elements if there is friction
    cudaMemcpyAsync(proxContact_d->proxContactF, proxContact_h->proxContactF, sizeof(proxData_t)*
        numContacts, cudaMemcpyHostToDevice);
    cudaMemcpyAsync(proxContact_d->proxGf, proxContact_h->proxGf, sizeof(proxG_t)*numContacts,
        cudaMemcpyHostToDevice);
  }
  cudaMemcpyAsync(proxContact_d->proxContactN, proxContact_h->proxContactN, sizeof(proxData_t)*
      numContacts, cudaMemcpyHostToDevice);
  cudaMemcpyAsync(proxContact_d->proxGn, proxContact_h->proxGn, sizeof(proxG_t)*numContacts,
      cudaMemcpyHostToDevice);
  return true;
}
```

B.2.3.3 proxCUDA.h

```
  /** @file plugins/proxSolver
 *
 * This file contains
 * @author: Jeremy Betz
 * @ingroup: proxSolver
 */

#ifndef _PROX_CUDA_PLUGIN_H
#define _PROX_CUDA_PLUGIN_H

#include <string>
#include "DvcKernel.h"
#include "proxCUDAInterface.h"

#ifdef WIN32
#ifdef PROX_CUDA_PLUGIN_EXPORTS
  #define PROX_CUDA_PLUGIN_API __declspec(dllexport)
#else
  #define PROX_CUDA_PLUGIN_API __declspec(dllimport)
#endif
//Linux
#else
  #define PROX_CUDA_PLUGIN_API
#endif

using namespace DVC;

/**
 * class Prox_CUDA
 * @ingroup: Prox_CUDA
 */
class Prox_CUDA : public Prox_Solver {
public:

  Prox_CUDA();
  virtual ~Prox_CUDA();
  const std::string &getName() const; // (Should figure out a way to check if a plugin is prox
      based using name)
  virtual void refresh(const DVC::PrefMap& prefs); // Data structures built here (so they don't
      need to be rebuilt every timestep)
  // FROM Prox_Solver
  bool populateProxDataStructure(const std::list <DynamicalBody *> *m_dynamicalBodies,
  const std::list<DvcCollisionResultPtr> *m_collisions,
  size_t m_numConstrainedBodies, size_t m_numContacts);
  bool solveProx(); // Interface from Prox_Solver

  bool getProxBodyNu(int index, DVC::Vector<REAL> *nu);



private:
  //for populateProxDataStructure:
  virtual void initProxBodies(const std::list <DynamicalBody *> *m_dynamicalBodies);
  virtual void formNormalProx(const std::list<DvcCollisionResultPtr> *m_collisions);
  virtual void formFrictionProx(const std::list<DvcCollisionResultPtr> *m_collisions);

  bodyPointers proxBody_h;
  bodyPointers proxBody_d;
  contactPointers proxContact_h;
  contactPointers proxContact_d;
  results_t *results_h;
  results_t *results_d;

  // Values from setup (refresh function/prefmap)
  int maxBodies;
  int maxContacts;
  int solveMode;
  double relaxCoeff; // also known as omega
  int debug;
  int maxIters;

};

#endif // _PROX_CUDA_PLUGIN_H
```

B.2.3.4 proxCUDA.cpp

```
/** @defgroup *** solver plugin
 *  @ingroup cpsolver
 *  Group *** is a subgroup of cpsolver
 */

/** @file plugins/.cpp
 * The *** solver plugin
 * This file contains the *** solver plugin implementation
 * @author: Jeremy Betz
 * @ingroup:
 */

#include "proxCUDA.h"
#include <string>
#include <fstream>
#include <iostream>
#include <ForceController.h>

/**
 * Default constructor
 */
Prox_CUDA::Prox_CUDA()
{
  // Set values to defaults that will cause predictable breaking if not given proper values (in
       refresh() )
  maxBodies = 0;
  maxContacts = 0;
  solveMode = 0;
  relaxCoeff = 0;
  debug = 0;
  //Host memory
  proxBody_h.proxNu = NULL;
  proxBody_h.proxNu_lp1 = NULL;
  proxBody_h.proxBodyConsts = NULL;
  proxBody_h.proxBodyExternal = NULL;
  proxContact_h.convergedF = NULL;
  proxContact_h.convergedN = NULL;
  proxContact_h.proxContactF = NULL;
  proxContact_h.proxContactN = NULL;
  proxContact_h.proxGf = NULL;
  proxContact_h.proxGn = NULL;
  results_h = NULL;
  //Device memory
  proxBody_d.proxNu = NULL;
  proxBody_d.proxNu_lp1 = NULL;
  proxBody_d.proxBodyConsts = NULL;
  proxBody_d.proxBodyExternal = NULL;
  proxContact_d.convergedF = NULL;
  proxContact_d.convergedN = NULL;
  proxContact_d.proxContactF = NULL;
  proxContact_d.proxContactN = NULL;
  proxContact_d.proxGf = NULL;
  proxContact_d.proxGn = NULL;
  results_d = NULL;
}

Prox_CUDA::~Prox_CUDA()
{
  // Delete host memory allocated by cuda (pinned memory)
  deleteHostMem(&proxBody_h,&proxContact_h);
  // Let cuda code delete all allocated cuda variables
  deleteCUDAMem(&proxBody_d,&proxContact_d);
  //delete result struct
  deleteResult(results_h, results_d);
}

void Prox_CUDA::refresh( const PrefMap & prefs ) {


  InitCUDA();

  //allocate struct for kernel comm
  if (results_h == NULL){
    allocateResult(results_h, results_d);
  }

  //printf("debug qq %d\n",sizeof(normalProx_t));
  std::string pluginPath = GetPluginPrefPath();

  maxBodies = prefs.get_int(pluginPath + "/maxBodies");
  if (maxBodies < 1){ maxBodies = 10; printf("Invalid number of maxBodies, setting to 10...\n")
      ;}
  if (proxBody_h.proxNu == NULL) // if value is not null, assume already allocated (should allow
       adjusting size)
    allocateBodiesMem(maxBodies,&proxBody_h,&proxBody_d);

  maxContacts = prefs.get_int(pluginPath + "/maxContacts");
  if (maxContacts < 1){ maxContacts = 100; printf("Invalid number of maxContacts, setting to 
      100...\n");}
  if (proxContact_h.proxContactN == NULL) // if value is not null, assume already allocated (
       should allow adjusting size)
    allocateContactsMem(maxContacts,&proxContact_h,&proxContact_d);

  // should be changed to use enum
  std::string solveStr = prefs.get_string(pluginPath + "/solveMode");
  if (solveStr == "richardson")
    solveMode = 1;
  else if (solveStr == "jacobi")
    solveMode = 2;
  else if (solveStr == "gauss_seidel")
    solveMode = 3;
  else {
    solveMode = 1;
    printf("Invalid_solveMode,_using_richardson...\n");
  }

  maxIters = prefs.get_int(pluginPath + "/maxIters");
  if (maxIters < 1){ maxIters = 500; printf("Invalid_number_of_maxIters,_setting_to_500...\n");}

  debug = prefs.get_int(pluginPath + "/debug");
  relaxCoeff = prefs.get_double(pluginPath + "/relaxCoeff"); // also known as omega
  if (relaxCoeff == 0) {printf("Invalid_relaxCoeff,_setting_to_1...\n"); relaxCoeff = 1; } //
       relaxCoeff can't be 0, so set to a default of 1

}

bool Prox_CUDA::populateProxDataStructure(const std::list< DynamicalBody* >* m_dynamicalBodies,
              const std::list< DvcCollisionResultPtr >* m_collisions,
              size_t m_numConstrainedBodies, size_t m_numContacts)
{
  numConstrainedBodies = m_numConstrainedBodies;
  numContacts = m_numContacts;

  // Populate the datastructure for the contacts
  initProxBodies(m_dynamicalBodies);
  // Copy datastructure to device
  populateProxBodiesCUDA(numConstrainedBodies,&proxBody_h,&proxBody_d);
  // Populate the datastructure for the bodies
  formNormalProx(m_collisions);
  if (coeffFriction != 0.0){ // if there is friction, form datastructure for friction contacts
    formFrictionProx(m_collisions);
  }
  // Copy datastructure to device
  populateProxContactsCUDA(numContacts,coeffFriction,&proxContact_h,&proxContact_d);
  return true;
}

void Prox_CUDA::initProxBodies(const std::list< DynamicalBody* >* m_dynamicalBodies)
{
  DVC::Vector<REAL> newForce(3);
  DVC::Vector<REAL> newForceSum(3);
  std::list <DynamicalBody *>::const_iterator bodyItr = m_dynamicalBodies->begin();
  for ( ; bodyItr != m_dynamicalBodies->end(); ++bodyItr )
  {
      DynamicalBody *b = *bodyItr;
      if( !(b->IsConstrained()) ) // Body in freefall
      { // free fall body, not formulated in ST LCP
    continue;
      }

      newForceSum.clear();
      unsigned int rowIndex = b->GetWrenchRowIndex()/3;
      // Init default values
```

```
        proxBody_h.proxBodyExternal[rowIndex].pExt[0] = 0;
        proxBody_h.proxBodyExternal[rowIndex].pExt[1] = 0;
        proxBody_h.proxBodyExternal[rowIndex].pExt[2] = 0;
        if (b->IsForceControlled())
        { // sum up the contributing forces from each force controller
      const std::list <const ForceController *>& forceCtrls = b->GetForceControllers();
      std::list <const ForceController *>::const_iterator itr = forceCtrls.begin();
      for( ; itr != forceCtrls.end(); ++itr){
        newForce.clear();
        (*itr)->GetForce(simTime, newForce, b->GetQ(), b->GetNu() );
        newForceSum += newForce;
      }
      // add the impulses from the controllers
      proxBody_h.proxBodyExternal[rowIndex].pExt[0] = stepSize*newForceSum[0];
      proxBody_h.proxBodyExternal[rowIndex].pExt[1] = stepSize*newForceSum[1];
      proxBody_h.proxBodyExternal[rowIndex].pExt[2] = stepSize*newForceSum[2];
        }

        // add the gravitational force
        proxBody_h.proxBodyExternal[rowIndex].pExt[1] -= gravity * stepSize * b->GetMass() ;
        // Get other info for prox solver
        proxBody_h.proxBodyConsts[rowIndex].mass = b->GetMass();
        proxBody_h.proxBodyConsts[rowIndex].mInertia = b->GetMomentInertia();
        proxBody_h.proxNu[rowIndex].nu[0] = b->GetNu()(0);
        proxBody_h.proxNu[rowIndex].nu[1] = b->GetNu()(1);
        proxBody_h.proxNu[rowIndex].nu[2] = b->GetNu()(2);
  }
}

void Prox_CUDA::formNormalProx(const std::list< DvcCollisionResultPtr >* m_collisions)
{
  int i = 0;

  DVC::Vector<REAL> n(2), r(2), Gn_vec(3);

  std::list<DvcCollisionResultPtr>::const_iterator contactItr;
  for (contactItr = m_collisions->begin(); contactItr != m_collisions->end(); ++contactItr )
  {
      const DvcCollisionResultPtr &colRes = *contactItr;

      // Get the two bodies in contact.
      Body * m1 = colRes->b1;
      Body * m2 = colRes->b2;

      // Sanity Check: Should never be collision checking between 2 static objects
      assert( ! ( m1->GetBodyType()==BODY_OBSTACLE && m2->GetBodyType()==BODY_OBSTACLE ) );
      // Get the normal and tangential information
      n(0) = colRes->normalB1toB2[0];
      n(1) = colRes->normalB1toB2[1];
      // Init p_n to 0
      proxContact_h.proxContactN[i].p = 0;
      // Get each body's row location in the wrench and each body's position
      if ( m1->GetBodyType()==BODY_DYNAMICAL )
      { // Only in the wrench if it is not an obstacle
    //M1RowIndex = static_cast< const DynamicalBody * > ( m1 )->GetWrenchRowIndex();
    const DVC::Vector<REAL> &M1Pos = static_cast< const DynamicalBody * > ( m1 )->GetQ();
    //const DVC::Vector<REAL> &M1Nu = static_cast< const DynamicalBody * > ( m1 )->GetNu();

    r(0) = colRes->b1ContactLoc[0] - M1Pos[0];
    r(1) = colRes->b1ContactLoc[1] - M1Pos[1];
    assert(r(0) == r(0) && r(1) == r(1)  );
    proxContact_h.proxGn[i].G1[0] = n(0);
    proxContact_h.proxGn[i].G1[1] = n(1);
    proxContact_h.proxGn[i].G1[2] = r.cross2D(n);
    proxContact_h.proxContactN[i].gap = colRes->distance;
    proxContact_h.proxContactN[i].b1index = static_cast< const DynamicalBody * > ( m1 )->
        GetWrenchRowIndex()/3;
      }
      else proxContact_h.proxContactN[i].b1index = -1;

      if ( m2->GetBodyType()==BODY_DYNAMICAL )
      { // Only in the wrench if it is not an obstacle
    //M2RowIndex = static_cast< const DynamicalBody * > ( m2 )->GetWrenchRowIndex();
    const DVC::Vector<REAL> &M2Pos = static_cast< const DynamicalBody * > ( m2 )->GetQ();
    //const DVC::Vector<REAL> &M2Nu = static_cast< const DynamicalBody * > ( m2 )->GetNu();

    r(0) = colRes->b2ContactLoc[0] - M2Pos[0];
    r(1) = colRes->b2ContactLoc[1] - M2Pos[1];
    assert(r(0) == r(0) && r(1) == r(1)  );
    n(0) = -n(0);
    n(1) = -n(1);
    proxContact_h.proxGn[i].G2[0] = n(0);
    proxContact_h.proxGn[i].G2[1] = n(1);
    proxContact_h.proxGn[i].G2[2] = r.cross2D(n);
    proxContact_h.proxContactN[i].gap = colRes->distance;
    proxContact_h.proxContactN[i].b2index = static_cast< const DynamicalBody * > ( m2 )->
        GetWrenchRowIndex()/3;
      }
      else proxContact_h.proxContactN[i].b2index = -1;
      //++ colIndex; // Next column



      i++; //Position in array
  }
  return;
}

void Prox_CUDA::formFrictionProx(const std::list< DvcCollisionResultPtr >* m_collisions)
{
  int i = 0;

  DVC::Vector<REAL> n(2), r(2), Gf_vec(3);

  std::list<DvcCollisionResultPtr>::const_iterator contactItr;
  for (contactItr = m_collisions->begin(); contactItr != m_collisions->end(); ++contactItr )
  {
      const DvcCollisionResultPtr &colRes = *contactItr;

      // Get the two bodies in contact.
      Body * m1 = colRes->b1;
      Body * m2 = colRes->b2;

      // Sanity Check: Should never be collision checking between 2 static objects
      assert( ! ( m1->GetBodyType()==BODY_OBSTACLE && m2->GetBodyType()==BODY_OBSTACLE ) );

      // Get the normal and tangential information
      //(x' = y; y' = -x)
      n(0) = colRes->normalB1toB2[1];
      n(1) = -(colRes->normalB1toB2[0]);
      proxContact_h.proxContactF[i].p = 0;
      // Get each body's row location in the wrench and each body's position
      if ( m1->GetBodyType()==BODY_DYNAMICAL )
      { // Only in the wrench if it is not an obstacle
    //M1RowIndex = static_cast< const DynamicalBody * > ( m1 )->GetWrenchRowIndex();
    const DVC::Vector<REAL> &M1Pos = static_cast< const DynamicalBody * > ( m1 )->GetQ();
    //const DVC::Vector<REAL> &M1Nu = static_cast< const DynamicalBody * > ( m1 )->GetNu();
```

```
      r(0) = colRes->b1ContactLoc[0] - M1Pos[0];
      r(1) = colRes->b1ContactLoc[1] - M1Pos[1];
      assert(r(0) == r(0) && r(1) == r(1)   );
      proxContact_h.proxGf[i].G1[0] = n(0);
      proxContact_h.proxGf[i].G1[1] = n(1);
      proxContact_h.proxGf[i].G1[2] = r.cross2D(n);
      proxContact_h.proxContactF[i].b1index = static_cast< const DynamicalBody * > ( m1 )->
          GetWrenchRowIndex()/3;
        }
        else proxContact_h.proxContactF[i].b1index = -1;

        if ( m2->GetBodyType()==BODY_DYNAMICAL )
        { // Only in the wrench if it is not an obstacle
      //M2RowIndex = static_cast< const DynamicalBody * > ( m2 )->GetWrenchRowIndex();
      const DVC::Vector<REAL> &M2Pos = static_cast< const DynamicalBody * > ( m2 )->GetQ();
      //const DVC::Vector<REAL> &M2Nu = static_cast< const DynamicalBody * > ( m2 )->GetNu();

      r(0) = colRes->b2ContactLoc[0] - M2Pos[0];
      r(1) = colRes->b2ContactLoc[1] - M2Pos[1];
      assert(r(0) == r(0) && r(1) == r(1)   );
      n(0) = -n(0);
      n(1) = -n(1);
      proxContact_h.proxGf[i].G2[0] = n(0);
      proxContact_h.proxGf[i].G2[1] = n(1);
      proxContact_h.proxGf[i].G2[2] = r.cross2D(n);
      proxContact_h.proxContactF[i].b2index = static_cast< const DynamicalBody * > ( m2 )->
          GetWrenchRowIndex()/3;
        }
        else proxContact_h.proxContactF[i].b2index = -1;
        //++colIndex; // Next column



        i++; //Position in array
  }
  return;
}

bool Prox_CUDA::solveProx()
{
  solveLCPProx(solveMode,relaxCoeff,coeffFriction,maxIters,numContacts,numConstrainedBodies,&
       proxContact_d,&proxContact_h,&proxBody_d,&proxBody_h,results_h,results_d);

  return true;
}

bool Prox_CUDA::getProxBodyNu(int index, DVC::Vector<REAL> *nu)
{
  (*nu)[0] = (REAL)proxBody_h.proxNu_lp1[index].nu[0];
  (*nu)[1] = (REAL)proxBody_h.proxNu_lp1[index].nu[1];
  (*nu)[2] = (REAL)proxBody_h.proxNu_lp1[index].nu[2];
  return true;
}

const std::string & Prox_CUDA::getName() const {
      static std::string sName("Prox_CUDA_Solver");
      return sName;
}

///// The following methods are required for all plugins /////////
/// Retrieve the engine version we're going to expect
extern "C" PROX_CUDA_PLUGIN_API int getEngineVersion() {
  return DvcEngineVersion;
}
```

```
/// Tells us to register our functionality to an engine kernel
extern "C" PROX_CUDA_PLUGIN_API void registerPlugin(DvcKernel &K, const std::string & pluginName
    ) {

  Prox_CUDA *ptr = new Prox_CUDA();
  ptr->SetPluginName(pluginName);
  K.getCP_SolverServer().AddCP_Solver(ptr);

}
```

B.2.3.5 proxCUDAKernels.cuh

```
#pragma once
// Set CUDA kernel to use double or single precision
#include "proxCUDAInterface.h"

//if defined, then launch kernels all from host, if not defined, then will use kernel based
    solver
#define DEBUG_KERNEL_HOST

bool calculateR(int solveMode, FULL relaxCoeff, bool friction, int numContacts, contactPointers
    *proxContact_d, bodyPointers *proxBody_d, cudaStream_t *stream);
//for calculateR:
#ifdef DEBUG_KERNEL_HOST
__global__ void calculateRichardson(FULL relaxCoeff, bool friction, int numContacts,
    contactPointers proxContact_d);
#else
__device__ void calculateRichardson(FULL relaxCoeff, bool friction, int numContacts,
    contactPointers proxContact_d);
#endif

#ifdef DEBUG_KERNEL_HOST
__global__ void calculateJacobi(FULL relaxCoeff, bool friction, int numContacts, contactPointers
     proxContact_d, bodyPointers proxBody_d);
#else
__device__ void calculateJacobi(FULL relaxCoeff, bool friction, int numContacts, contactPointers
     proxContact_d, bodyPointers proxBody_d);
#endif

//for checkConverge:
bool checkConverge(int numContacts, bool friction, contactPointers *proxContact_h);

#ifndef DEBUG_KERNEL_HOST
__global__ void solveLCPProxKernel(int solveMode, int maxIters, int numContacts, int
    numConstrainedBodies, bool friction, FULL coeffFriction, FULL relaxCoeff, contactPointers
    proxContact_d, bodyPointers proxBody_d, results_t *results_d);
#endif

//for updateBodyNu:
#ifdef DEBUG_KERNEL_HOST
__global__ void updateBodyNu(bool friction, int numContacts, int numConstrainedBodies,
    contactPointers proxContact_d, bodyPointers proxBody_d);
#else
__device__ void updateBodyNu(bool friction, int numContacts, int numConstrainedBodies,
    contactPointers proxContact_d, bodyPointers proxBody_d);
#endif

//for solveLCPProx
#ifdef DEBUG_KERNEL_HOST
__global__ void solveLCPProxNormal(int numContacts, contactPointers proxContact_d, bodyPointers
    proxBody_d);
#else
```

```
__device__ void solveLCPProxNormal(int numContacts, contactPointers proxContact_d, bodyPointers
     proxBody_d);
#endif

#ifdef DEBUG_KERNEL_HOST
__global__ void solveLCPProxFriction(int numContacts, FULL coeffFriction, contactPointers
     proxContact_d, bodyPointers proxBody_d);
#else
__device__ void solveLCPProxFriction(int numContacts, FULL coeffFriction, contactPointers
     proxContact_d, bodyPointers proxBody_d);
#endif
```

B.2.3.6 proxCUDAKernels.cu

```
#include "proxCUDAKernels.cuh"

#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include "cuPrintf.cu"
#include <math.h>

//#define PRINT_KERNEL_ERRORS

#define convergeNormal -1e-6
#define convergeFriction 1e-6


/*********************************************************************/
/* Init CUDA                                                          */
/*********************************************************************/
#if __DEVICE_EMULATION__

bool InitCUDA(void){return true;}

#else
extern "C" bool InitCUDA(void)
{
        int count = 0;
        int i = 0;

        cudaGetDeviceCount(&count);
        if(count == 0) {
                fprintf(stderr, "There is no device.\n");
                return false;
        }

        for(i = 0; i < count; i++) {
                cudaDeviceProp prop;
                if(cudaGetDeviceProperties(&prop, i) == cudaSuccess) {
                        if(prop.major >= 1) {
                                break;
                        }
                }
        }
        if(i == count) {
                fprintf(stderr, "There is no device supporting CUDA.\n");
                return false;
        }
        cudaSetDevice(i);
   cudaPrintfInit();
   //printf("FULL size = %d\n",(int)sizeof(FULL));
        printf("CUDA initialized.\n");
        return true;
```

```
}

#endif


/**********************************************************************/
/* KERNELS                                                            */
/**********************************************************************/

// Threads are 0 indexed
__device__ int getId(){
  //2D block rows, 2D thread blocks
  return (gridDim.x*blockIdx.y+blockIdx.x)*(blockDim.x*blockDim.y) + (blockDim.x*threadIdx.y) +
      threadIdx.x;
}

//returns if a thread is an overflow thread (will just return [exit] if true)
__device__ bool amIDead(int maxId){
  return (getId() > maxId); // maxId will be numBlank-1
}

#ifdef DEBUG_KERNEL_HOST
__global__ void calculateRichardson(FULL relaxCoeff, bool friction, int numContacts,
    contactPointers proxContact_d){
#else
__device__ void calculateRichardson(FULL relaxCoeff, bool friction, int numContacts,
    contactPointers proxContact_d){
#endif
  if (amIDead(numContacts-1)) return; //kill useless threads at beginning
  proxContact_d.proxContactN[getId()].r = relaxCoeff;
  if (friction){
    proxContact_d.proxContactF[getId()].r = relaxCoeff;
  }
  return;
}

#ifdef DEBUG_KERNEL_HOST
__global__ void calculateJacobi(FULL relaxCoeff, bool friction, int numContacts, contactPointers
     proxContact_d, bodyPointers proxBody_d){
#else
__device__ void calculateJacobi(FULL relaxCoeff, bool friction, int numContacts, contactPointers
     proxContact_d, bodyPointers proxBody_d){
#endif
  if (amIDead(numContacts-1)) return; //kill useless threads at beginning
  FULL delassus = 0;
  int id = getId();
  if (proxContact_d.proxContactN[id].b1index != -1){
    delassus += (proxContact_d.proxGn[id].G1[0]*proxContact_d.proxGn[id].G1[0])/proxBody_d.
        proxBodyConsts[proxContact_d.proxContactN[id].b1index].mass +
    (proxContact_d.proxGn[id].G1[1]*proxContact_d.proxGn[id].G1[1])/proxBody_d.proxBodyConsts[
        proxContact_d.proxContactN[id].b1index].mass +
    (proxContact_d.proxGn[id].G1[2]*proxContact_d.proxGn[id].G1[2])/proxBody_d.proxBodyConsts[
        proxContact_d.proxContactN[id].b1index].mInertia;
  }
  if (proxContact_d.proxContactN[id].b2index != -1){
    delassus += (proxContact_d.proxGn[id].G2[0]*proxContact_d.proxGn[id].G2[0])/proxBody_d.
        proxBodyConsts[proxContact_d.proxContactN[id].b2index].mass +
    (proxContact_d.proxGn[id].G2[1]*proxContact_d.proxGn[id].G2[1])/proxBody_d.proxBodyConsts[
        proxContact_d.proxContactN[id].b2index].mass +
    (proxContact_d.proxGn[id].G2[2]*proxContact_d.proxGn[id].G2[2])/proxBody_d.proxBodyConsts[
        proxContact_d.proxContactN[id].b2index].mInertia;
  }
  proxContact_d.proxContactN[getId()].r = relaxCoeff/delassus;
  if (friction){
      delassus = 0;
      if (proxContact_d.proxContactN[id].b1index != -1){ // NOTE proxContactN has already been
          looked up (so is probably cached) and will have the same b1 and b2index
        delassus += (proxContact_d.proxGf[id].G1[0]*proxContact_d.proxGf[id].G1[0])/proxBody_d.
             proxBodyConsts[proxContact_d.proxContactN[id].b1index].mass +
        (proxContact_d.proxGf[id].G1[1]*proxContact_d.proxGf[id].G1[1])/proxBody_d.proxBodyConsts[
             proxContact_d.proxContactN[id].b1index].mass +
        (proxContact_d.proxGf[id].G1[2]*proxContact_d.proxGf[id].G1[2])/proxBody_d.proxBodyConsts[
             proxContact_d.proxContactN[id].b1index].mInertia;
      }
      if (proxContact_d.proxContactN[id].b2index != -1){
        delassus += (proxContact_d.proxGf[id].G2[0]*proxContact_d.proxGf[id].G2[0])/proxBody_d.
             proxBodyConsts[proxContact_d.proxContactN[id].b2index].mass +
        (proxContact_d.proxGf[id].G2[1]*proxContact_d.proxGf[id].G2[1])/proxBody_d.proxBodyConsts[
             proxContact_d.proxContactN[id].b2index].mass +
        (proxContact_d.proxGf[id].G2[2]*proxContact_d.proxGf[id].G2[2])/proxBody_d.proxBodyConsts[
             proxContact_d.proxContactN[id].b2index].mInertia;
      }
      proxContact_d.proxContactF[getId()].r = relaxCoeff/delassus;

    }

    return;
}

#ifdef DEBUG_KERNEL_HOST
__global__ void updateBodyNu(bool friction, int numContacts, int numConstrainedBodies,
      contactPointers proxContact_d, bodyPointers proxBody_d){
#else
__device__ void updateBodyNu(bool friction, int numContacts, int numConstrainedBodies,
      contactPointers proxContact_d, bodyPointers proxBody_d){
#endif
    int id = getId();
    FULL localNu[3];
    //cuPrintf("I am thread %d!\n", id);
    if (amIDead(numConstrainedBodies-1)) return; //kill useless threads at beginning
    localNu[0] = 0;
    localNu[1] = 0;
    localNu[2] = 0;

    for (int i = 0; i < numContacts; i++){ // if body1 is me
      if (proxContact_d.proxContactN[i].b1index == id){
        localNu[0] += proxContact_d.proxGn[i].G1[0]*proxContact_d.proxContactN[i].p;
        localNu[1] += proxContact_d.proxGn[i].G1[1]*proxContact_d.proxContactN[i].p;
        localNu[2] += proxContact_d.proxGn[i].G1[2]*proxContact_d.proxContactN[i].p;
        if (friction){
    localNu[0] += proxContact_d.proxGf[i].G1[0]*proxContact_d.proxContactF[i].p;
    localNu[1] += proxContact_d.proxGf[i].G1[1]*proxContact_d.proxContactF[i].p;
    localNu[2] += proxContact_d.proxGf[i].G1[2]*proxContact_d.proxContactF[i].p;
        }
      }
      if (proxContact_d.proxContactN[i].b2index == id){
        localNu[0] += proxContact_d.proxGn[i].G2[0]*proxContact_d.proxContactN[i].p;
        localNu[1] += proxContact_d.proxGn[i].G2[1]*proxContact_d.proxContactN[i].p;
        localNu[2] += proxContact_d.proxGn[i].G2[2]*proxContact_d.proxContactN[i].p;
        if (friction){
    localNu[0] += proxContact_d.proxGf[i].G2[0]*proxContact_d.proxContactF[i].p;
    localNu[1] += proxContact_d.proxGf[i].G2[1]*proxContact_d.proxContactF[i].p;
    localNu[2] += proxContact_d.proxGf[i].G2[2]*proxContact_d.proxContactF[i].p;
        }
      }
    }
```

```
  proxBody_d.proxNu_lp1[id].nu[0] = proxBody_d.proxNu[id].nu[0] + ( (proxBody_d.proxBodyExternal
      [id].pExt[0] + localNu[0])/proxBody_d.proxBodyConsts[id].mass);
  proxBody_d.proxNu_lp1[id].nu[1] = proxBody_d.proxNu[id].nu[1] + ( (proxBody_d.proxBodyExternal
      [id].pExt[1] + localNu[1])/proxBody_d.proxBodyConsts[id].mass);;
  proxBody_d.proxNu_lp1[id].nu[2] = proxBody_d.proxNu[id].nu[2] + ( (proxBody_d.proxBodyExternal
      [id].pExt[2] + localNu[2])/proxBody_d.proxBodyConsts[id].mInertia);;
  //cuPrintf("I am thread %d! Nu_lp1[0]= %d\n", id,proxBody_d.proxNu_lp1[id].nu[0]);
  return;
}

#ifdef DEBUG_KERNEL_HOST
__global__ void solveLCPProxNormal(int numContacts, contactPointers proxContact_d, bodyPointers
    proxBody_d){
#else
__device__ void solveLCPProxNormal(int numContacts, contactPointers proxContact_d, bodyPointers
    proxBody_d){
#endif
  int id = getId();
  if (amIDead(numContacts-1)) return; //kill useless threads at beginning
  FULL p_n_star = 0;
  FULL rho = 0;

  if (proxContact_d.proxContactN[id].b1index != -1){ // add velocity of body 1
    rho += proxContact_d.proxGn[id].G1[0]*proxBody_d.proxNu_lp1[proxContact_d.proxContactN[id].
        b1index].nu[0]
  +  proxContact_d.proxGn[id].G1[1]*proxBody_d.proxNu_lp1[proxContact_d.proxContactN[id].b1index
      ].nu[1]
  +  proxContact_d.proxGn[id].G1[2]*proxBody_d.proxNu_lp1[proxContact_d.proxContactN[id].b1index
      ].nu[2];
  }
  if (proxContact_d.proxContactN[id].b2index != -1){ // add velocity of body 2
    rho += proxContact_d.proxGn[id].G2[0]*proxBody_d.proxNu_lp1[proxContact_d.proxContactN[id].
        b2index].nu[0]
  +  proxContact_d.proxGn[id].G2[1]*proxBody_d.proxNu_lp1[proxContact_d.proxContactN[id].b2index
      ].nu[1]
  +  proxContact_d.proxGn[id].G2[2]*proxBody_d.proxNu_lp1[proxContact_d.proxContactN[id].b2index
      ].nu[2];
  }

  rho += proxContact_d.proxContactN[id].gap;


  // RHO FINISHED CALCULATING
  // CALCULATE P_N_STAR
  p_n_star = proxContact_d.proxContactN[id].p - proxContact_d.proxContactN[id].r*rho;

  //enforce prox condition (p_n > 0)
  if (p_n_star < 0){
    p_n_star = 0;
  }

  //save final value of prox
  proxContact_d.proxContactN[id].p = p_n_star;
  //cuPrintf("I am thread %d! p = %f\n",id,proxContact_d.proxContactN[id].p);
  //###
  //cuPrintf("I am thread %d! converged = %d\n", id, proxContact_d.convergedN[id]);

  if (rho > convergeNormal){
    proxContact_d.convergedN[id] = true;
  }
  else {
    proxContact_d.convergedN[id] = false;
  }
  /*if (proxContact_d.convergedN[id] != true){
    cuPrintf("I am normal thread %d! p = %f\n",id,rho);
  }*/
  return;
}

#ifdef DEBUG_KERNEL_HOST
__global__ void solveLCPProxFriction(int numContacts, FULL coeffFriction, contactPointers
    proxContact_d, bodyPointers proxBody_d){
#else
__device__ void solveLCPProxFriction(int numContacts, FULL coeffFriction, contactPointers
    proxContact_d, bodyPointers proxBody_d){
#endif
  int id = getId();
  if (amIDead(numContacts-1)) return; //kill useless threads at beginning
  FULL p_f_star = 0;
  FULL rho = 0;

  if (proxContact_d.proxContactN[id].p == 0.0){// if no normal force, then friction = 0
    proxContact_d.proxContactF[id].p = 0;
    proxContact_d.convergedF[id] = true;
    return;
  }
  // use proxContactN for b1 and b2 index, (possibly cached for faster access)
  if (proxContact_d.proxContactN[id].b1index != -1){ // add velocity of body 1
    rho += proxContact_d.proxGf[id].G1[0]*proxBody_d.proxNu_lp1[proxContact_d.proxContactN[id].
        b1index].nu[0]
  +  proxContact_d.proxGf[id].G1[1]*proxBody_d.proxNu_lp1[proxContact_d.proxContactN[id].b1index
      ].nu[1]
  +  proxContact_d.proxGf[id].G1[2]*proxBody_d.proxNu_lp1[proxContact_d.proxContactN[id].b1index
      ].nu[2];
  }
  if (proxContact_d.proxContactN[id].b2index != -1){ // add velocity of body 2
    rho += proxContact_d.proxGf[id].G2[0]*proxBody_d.proxNu_lp1[proxContact_d.proxContactN[id].
        b2index].nu[0]
  +  proxContact_d.proxGf[id].G2[1]*proxBody_d.proxNu_lp1[proxContact_d.proxContactN[id].b2index
      ].nu[1]
  +  proxContact_d.proxGf[id].G2[2]*proxBody_d.proxNu_lp1[proxContact_d.proxContactN[id].b2index
      ].nu[2];
  }
  /*
  if ( ((FULL)fabs(rho) < convergeFriction) && (proxContact_d.convergedF[id] == true) ){
    return;
  }
  else*/ if ((FULL)fabs(rho) < convergeFriction){
    proxContact_d.convergedF[id] = true;
  }
  else {
    proxContact_d.convergedF[id] = false;
  }

  // RHO FINISHED CALCULATING
  // CALCULATE P_N_STAR
  p_f_star = proxContact_d.proxContactF[id].p - proxContact_d.proxContactF[id].r*rho;

  //enforce prox condition
  if (p_f_star < -(coeffFriction*proxContact_d.proxContactN[id].p)){
    p_f_star = -(coeffFriction*proxContact_d.proxContactN[id].p);
    proxContact_d.convergedF[id] = true;
  }
  else if (p_f_star > (coeffFriction*proxContact_d.proxContactN[id].p)) {
    p_f_star = (coeffFriction*proxContact_d.proxContactN[id].p);
    proxContact_d.convergedF[id] = true;
  }
  //save final value of prox
  proxContact_d.proxContactF[id].p = p_f_star;
  return;
}

#ifndef DEBUG_KERNEL_HOST
__global__ void solveLCPProxKernel(int solveMode,
           int maxIters,
           int numContacts,
           int numConstrainedBodies,
           bool friction,
           FULL coeffFriction,
           FULL relaxCoeff,
           contactPointers proxContact_d,
           bodyPointers proxBody_d,
           results_t *results_d)
{
  int id = getId();
  //calculateR
  if(solveMode == 1)
    calculateRichardson(relaxCoeff,friction,numContacts,proxContact_d);
  else if (solveMode == 2)
    calculateJacobi(relaxCoeff,friction,numContacts,proxContact_d,proxBody_d);
  else {
    results_d->error=1;
    return;
  }
  // Update the bodies velocity to initialize nu_lp1
  updateBodyNu(friction, numContacts, numConstrainedBodies, proxContact_d,proxBody_d);
  int k = 0;

  do{
    k++;
    if (id == 1){
      results_d->globalConverge = true; //set to default
    }
    __threadfence();
    solveLCPProxNormal(numContacts,proxContact_d,proxBody_d);
    if (proxContact_d.convergedN[id] == false && !amIDead(numContacts-1))
  results_d->globalConverge = false; // if any contact hasn't converged, ensure iterations
     continue
    if (friction){
      solveLCPProxFriction(numContacts,friction,proxContact_d,proxBody_d);
      if (proxContact_d.convergedF[id] == false && !amIDead(numContacts-1))
  results_d->globalConverge = false; // if any contact hasn't converged, ensure iterations
     continue
    }
    //sync before dynamics update
    __threadfence();

    updateBodyNu(friction, numContacts, numConstrainedBodies, proxContact_d,proxBody_d);

  }while(!(results_d->globalConverge) && (k<maxIters)); //while not converged

  if (id == 1){
    results_d->numIters = k;
    results_d->error = 0;
    results_d->overvalue = false;
  }
  return;

}
#endif
```

```
/**********************************************************************/
/* HOST                                                               */
/**********************************************************************/

int numGrids(int size){

  return (int)ceil((float)size/32);
}

#ifdef DEBUG_KERNEL_HOST
bool calculateR(int solveMode, FULL relaxCoeff, bool friction, int numContacts, contactPointers
    *proxContact_d, bodyPointers *proxBody_d, cudaStream_t *stream){
  int numGrid = numGrids(numContacts);

  if (solveMode == 1) { // Richardson
    calculateRichardson<<<numGrid,32,0,*stream>>>(relaxCoeff,friction,numContacts,*proxContact_d
        );
  }
  else if (solveMode == 2) { // Jacobi
    calculateJacobi<<<numGrid,32,0,*stream>>>(relaxCoeff,friction,numContacts,*proxContact_d,*
        proxBody_d);

  }
  else if (solveMode == 3) { // Gauss-Seidel
    printf("Gauss-Seidel_not_yet_implemented,_probably_going_to_crash...\n");
    return false;
  }

  return true;
}
#endif

extern "C" bool solveLCPProx(int solveMode,
           FULL relaxCoeff,
           FULL coeffFriction,
           int maxIters,
           int numContacts,
           int numConstrainedBodies,
           contactPointers *proxContact_d,
           contactPointers *proxContact_h,
           bodyPointers *proxBody_d,
           bodyPointers *proxBody_h,
           results_t *results_h,
           results_t *results_d
          )
#ifdef DEBUG_KERNEL_HOST
// kernels launched from host
{
  //create streams
  cudaStream_t streams[2];
  for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&streams[i]);

  // DEBUG
  cudaError_t error;
  bool friction = (coeffFriction != 0.0);
  int contactGrid = numGrids(numContacts);
  int bodyGrid = numGrids(numConstrainedBodies);
  /*
  //[debug] ### ###
        size_t free_mem,total_mem, used_mem;
        cuMemGetInfo( &free_mem, &total_mem );
        used_mem = total_mem-free_mem;
        printf("total mem: %0.3f MB, free: %0.3f MB, used : %0.3f MB\n",
                  ((double)total_mem)/1024.0/1024.0,
                  ((double)free_mem )/1024.0/1024.0,
                  ((double)used_mem )/1024.0/1024.0 );
//[end debug]*/
#ifdef PRINT_KERNEL_ERRORS
// DEBUG
  cudaThreadSynchronize();
  error = cudaGetLastError();
  if (error != cudaSuccess){
    printf("PreLCP Simulation Error: %s\n", cudaGetErrorString(error));
  }
#endif

  // calculate R value (keep on gpu)
  calculateR(solveMode,relaxCoeff,friction,numContacts,proxContact_d,proxBody_d,&streams[0]);
#ifdef PRINT_KERNEL_ERRORS
  // DEBUG
  cudaThreadSynchronize();
  error = cudaGetLastError();
  if (error != cudaSuccess){
    printf("calcR Simulation Error: %s\n", cudaGetErrorString(error));
  }
#endif
  // Update the bodies velocity to initialize nu_lp1
  updateBodyNu<<<bodyGrid,32,0,streams[1]>>>(friction, numContacts, numConstrainedBodies, *
       proxContact_d,*proxBody_d);
#ifdef PRINT_KERNEL_ERRORS
  // DEBUG
  cudaThreadSynchronize();
  error = cudaGetLastError();
  if (error != cudaSuccess){
    printf("Update1 Simulation Error: %s\n", cudaGetErrorString(error));
  }
#endif

  //init loop variables
  bool converge = false;
  bool oldconverge = false;
  int k = 0;

  cudaThreadSynchronize(); // need nu_lp1 and R to continue
  solveLCPProxNormal<<<contactGrid,32,0,streams[1]>>>(numContacts,*proxContact_d,*proxBody_d);
  if (friction){
    solveLCPProxFriction<<<contactGrid,32,0,streams[1]>>>(numContacts,friction,*proxContact_d,*
         proxBody_d);
  }
  cudaThreadSynchronize(); // get a set of results to start
#ifdef PRINT_KERNEL_ERRORS
  // DEBUG
  cudaThreadSynchronize();
  error = cudaGetLastError();
  if (error != cudaSuccess){
    printf("Prox1 Simulation Error: %s\n", cudaGetErrorString(error));
  }
#endif
  do{
    //get converge results from last iteration
    cudaMemcpyAsync(proxContact_h->convergedN,proxContact_d->convergedN,sizeof(bool)*numContacts
         ,cudaMemcpyDeviceToHost,streams[0]);
    if (friction){
      cudaMemcpyAsync(proxContact_h->convergedF,proxContact_d->convergedF,sizeof(bool)*
           numContacts,cudaMemcpyDeviceToHost,streams[0]);
    }
#ifdef PRINT_KERNEL_ERRORS
```

```
    // DEBUG
    cudaThreadSynchronize();
    error = cudaGetLastError();
    if (error != cudaSuccess){
      printf("mem1 Simulation Error: %s\n", cudaGetErrorString(error));
    }
#endif
    //update bodies while copy is in progress
    updateBodyNu<<<bodyGrid,32,0,streams[1]>>>(friction, numContacts, numConstrainedBodies, *
        proxContact_d,*proxBody_d);
    k++;
    //debugging code ###
    //if (!(k%5)){printf("%d th loop\n",k);}
#ifdef PRINT_KERNEL_ERRORS
    // DEBUG
    cudaThreadSynchronize();
    error = cudaGetLastError();
    if (error != cudaSuccess){
      printf("update2 Simulation Error: %s\n", cudaGetErrorString(error));
    }
#endif
    //queue up next solve after bodies update
    solveLCPProxNormal<<<contactGrid,32,0,streams[1]>>>(numContacts,*proxContact_d,*proxBody_d);
    if (friction){
      solveLCPProxFriction<<<contactGrid,32,0,streams[1]>>>(numContacts,coeffFriction,*
          proxContact_d,*proxBody_d);
    }
        cudaPrintfDisplay(stdout, true);
#ifdef PRINT_KERNEL_ERRORS
    // DEBUG
    cudaThreadSynchronize();
    cudaError_t error = cudaGetLastError();
    if (error != cudaSuccess){
      printf("Prox2 Simulation Error: %s\n", cudaGetErrorString(error));
    }
#endif
    //check results of last iteration for convergence
    oldconverge = converge; // used to make sure 2 iterations are stable
    converge = checkConverge(numContacts,friction,proxContact_h);

    cudaThreadSynchronize();

    //printf("R and Nu_lp1 calculated?\n");
    cudaPrintfDisplay(stdout, true);
  }while((!converge) && (k<maxIters));
  // }while((!oldconverge) && (!converge) && (k<maxIters));
  //(!converge) && (k<maxIters) ###
  printf("finished at %d iters\n",k);
  //clean up streams (waits till stream finished processing before destroy)
  for (int i = 0; i < 2; ++i)
    cudaStreamDestroy(streams[i]);
  //copy results back to host, wait till done
  cudaMemcpy(proxBody_h->proxNu_lp1,proxBody_d->proxNu_lp1,sizeof(proxNu_t)*numConstrainedBodies
      ,cudaMemcpyDeviceToHost);
  cudaThreadSynchronize(); //probably not needed
#ifdef PRINT_KERNEL_ERRORS
  // DEBUG
  cudaThreadSynchronize();
  error = cudaGetLastError();
  if (error != cudaSuccess){
    printf("mem2 Simulation Error: %s\n", cudaGetErrorString(error));
  }
#endif
```

```
  return true;
}
#else
//extern "C" bool solveLCPProx defined above
//single kernel launch per timestep
{
  bool friction = (coeffFriction != 0.0);
  int contactGrid = numGrids(numContacts);

  solveLCPProxKernel<<<contactGrid,32,0>>>(solveMode,maxIters,numContacts,numConstrainedBodies,
      friction,coeffFriction,relaxCoeff,*proxContact_d,*proxBody_d,results_d);
  cudaThreadSynchronize();
  cudaMemcpy(results_h,results_d,sizeof(results_t),cudaMemcpyDeviceToHost);

  printf("finished_at_%d_iters\n",results_h->numIters);
  cudaMemcpy(proxBody_h->proxNu_lp1,proxBody_d->proxNu_lp1,sizeof(proxNu_t)*numConstrainedBodies
      ,cudaMemcpyDeviceToHost);
  return true;
}

#endif

bool checkConverge(int numContacts, bool friction, contactPointers *proxContact_h){

  for(unsigned  int i=0;i<numContacts;i++){ // Check for total convergence
    if ( (proxContact_h->convergedN[i] == false) || ((proxContact_h->convergedF[i] == false) &&
        friction) ){
  //printf("not converged at %d\n", i); ###
  return false;
    }
  }
  return true; // If didn't return false yet, then must be converged
}








__global__ void printTest(){
  int id = getId();

  cuPrintf("I_am_thread_%d!\n", id);
}



extern "C" void runTestCuda() {

  InitCUDA();
  dim3 dimBlock(10,10);//under 512 total (1024 for 2.0)
  dim3 dimGrid(2,2); //under 65535 total
  cudaPrintfInit();

  printTest<<<dimGrid,dimBlock>>>();
  cudaPrintfDisplay(stdout, true);


  cudaPrintfEnd();


};
```