A Parallel Active-Set Method for Solving Frictional Contact Problems

by

Joshua Alexander Litven

B.Sc., The University of Waterloo, 2010

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

The Faculty of Graduate Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

February 18, 2013

Abstract

Simulating frictional contact is a challenging computational task and there exist a variety of techniques to do so. One such technique, the staggered projections algorithm, requires the solution of two convex quadratic program (QP) subproblems at each iteration. We introduce a method, SCHURPA, which employs a primal-dual active-set strategy to efficiently solve these QPs based on a Schur-complement method. A single factorization of the initial saddle point system and a smaller dense Schur-complement is maintained to solve subsequent saddle point systems. Exploiting the parallelizability and warm-starting capabilities of the active-set method as well as the problem structure of the QPs yields a novel approach to the problem of frictional contact. Numerical results of a parallel GPU implementation using NVIDIA's CUDA applied to a physical simulator of highly deformable bodies are presented.

Table of Contents

Appendices

List of Tables

List of Figures

List of Algorithms

Acknowledgments

First and foremost, I would like to thank both of my supervisors, Chen Greif and Dinesh K. Pai, for their outstanding guidance. Their curiosity, insights, and passion are truly inspiring. I greatly appreciate the thoughtfulness and understanding that they have shown me.

I would like to thank David I.W. Levin for teaching me many important things about life. Special thanks go to Ye Fan, who coded the Eulerian solids simulator. I am also grateful to Uri Ascher for reading this thesis and providing excellent feedback.

Finally, I would like to thank my family for their unconditional love and support throughout the years.

Chapter 1

Introduction

1.1 Problem Statement

Physical simulation, the computational process of simulating physical objects that we observe in reality, is a profoundly important and all-encompassing area of modern research. As such, efficient and scalable methods for simulating physical phenomena of ever increasing size and complexity are highly desirable. In particular, simulating realistic frictional contact has applications ranging from computer graphics to biomechanics.

An Eulerian solids simulator, presented in (Levin et al., 2011), simulates highly deformable solid bodies undergoing frictionless contact. The simulator utilizes the finite volume method permitting a parallel implementation on the GPU using CUDA (NVIDIA, 2012b). However, the simulator is currently incapable of simulating frictional contact. Lacking the ability to simulate friction restricts the simulator from capturing important phenomena such as hands grasping objects, fingers sliding across a surface etc. Incorporating frictional contact into the simulator is therefore highly desirable.

Resolving frictional contact dynamics is a challenging, formidable task, and several techniques have been proposed to do so (Jean, 1999; Stewart and Trinkle, 1996; Kaufman et al., 2008). The staggered projections (SP) algorithm, introduced in (Kaufman et al., 2008), is an attractive method because it is robust and can often converge rapidly. As the simulator implementation is parallel and

scalable, the implementation of the frictional contact solution procedure should share these attractive properties.

The work presented in this thesis details an efficient parallel solution framework which incorporates the staggered projections algorithm to resolve frictional contact dynamics in the Eulerian solids simulator. We demonstrate the effectiveness and scalability of our method with a GPU implementation.

1.2 Solution Methodology

The staggered projections algorithm resolves frictional contact by iteratively solving a pair of convex quadratic program (QP) subproblems. A general QP may be given as

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2}x^T H x - c^T x \\ \text{subject to} \quad & A^T x \leq b \quad E^T x = d, \end{aligned} \tag{1.1}$$

where $c, d, x \in \mathbb{R}^n$, $A \in \mathbb{R}^{n\times m}$, $E \in \mathbb{R}^{n\times p}$ and $H \in \mathbb{R}^{n\times n}$. The QP (1.1) is convex if H is symmetric and positive semi-definite.

QPs are ubiquitous in a diverse array of fields, and arise as subproblems of general purpose optimization schemes such as the sequential quadratic programming method (Nocedal and Wright, 1999, Chapter 18). As such, solving these problems efficiently has been the subject of decades of research and remains an active area to this day. The fruits of these laborious efforts is evidenced by a plethora of algorithmic methodologies, each with strengths and weaknesses. These include their ability to handle sparse data, scalability, robustness, accuracy etc.

One such methodology is the class of algorithms known as active-set methods (ASMs). A particularly attractive trait of these methods is their ability to *warm start*: A good initial guess of the solution to the QP being solved significantly

improves the convergence rate of these algorithms. A good guess of an optimal solution in physical simulation arises naturally due to temporal coherence. More precisely, at each time step of the simulation loop, the induced QPs are in some sense a small perturbation of the QP solved at the previous time step, inducing techniques to provide initial estimates. This perturbation depends on the specific implementation, time step size and other factors. Thus ASMs are a natural approach to solve QPs arising in physical simulation. However, there are two fundamental limitations to the classical ASMs:

- they cannot handle sparse data, which limits their applicability to large-scale problems, and
- they cannot be parallelized easily, often requiring many cheap but sequential iterations.

In contrast, the class of interior point methods (IPs) can handle sparse data, and have since become the more popular method in solving large problems. However, IPs lack the warm-starting capabilities ASMs enjoy (Wright, 1987, Chapter 11).

To overcome the limitations of the classical ASM approaches, we leverage the primal-dual active-set method (PDASM) (Ito and Kunisch, 2008) to retain the benefits of ASMs whilst being able to solve large, sparse QPs efficiently. PDASM is central to the work of this thesis.

1.2.1 The Primal-Dual Active-Set Method

Typically, each iteration of an ASM selects a *working set* which consists of a subset of the constraints given in (1.1). This working set serves as a guess to the optimal active set and yields an equality-constrained subproblem to be solved, or equivalently, a saddle point linear system. Unlike classical ASMs which ensure primal or dual feasibility over iterates, PDASM can violate both primal and

dual feasibility. Instead, PDASM induces the subproblem by enforcing complementarity to hold. One can show that PDASM is equivalent to a semismooth Newton method applied to the KKT conditions of (1.1) (Hintermüller et al., 2002) (see Section 2.4). Two fundamental differences between classical ASMs and PDASM are:

- PDASM allows *multiple* changes to the working set per iteration, being able to quickly identify the optimal active set, and
- being a semismooth Newton method, PDASM super-linearly converges. Typically very few iterations are required.

These properties imply that PDASM overcomes the previously mentioned limitations of classical ASMs. Super-linear convergence allows the method to scale well with problem size. In Chapter 3, we utilize multiple changes to the working set in our parallel implementation.

Recently, (PDASM) has been successfully applied to constrained optimal control (Bergounioux et al., 1999), (Kunisch and Rösch, 2002), contact and friction (Brunssen et al., 2007), (Hüeber and Wohlmuth, 2005), and others (Hintermuller, 2004). The convergence properties of PDASM are an active area of research. Kunisch and Ito showed global convergence for certain classes of Hessians (Ito and Kunisch, 2008). In (Kunisch and Rendl, 2003), it was observed that PDASM quickly eliminates the inactive constraints from the current estimate of the active set at a given iteration. This served as a motivation for our work.

1.2.2 SCHURPA

Efficiency of ASMs is determined by the method of solving the saddle point systems. In classical methods, a factorization of the initial saddle point system K_0

is produced, and the factors are then updated, e.g., by orthogonal factorizations. This updating process is what can potentially destroy sparsity.

Another approach uses a factorization of K_0 and, rather than updating factors, subsequent saddle point systems can be solved using an expanded form, given as

$$\begin{pmatrix} K_0 & U \\ U^T & V \end{pmatrix} \begin{pmatrix} y \\ \pi \end{pmatrix} = \begin{pmatrix} c \\ w \end{pmatrix}. \tag{1.2}$$

Equation (1.2) is solved via the factorization of K_0 and the Schur-complement matrix

$$C = V - U^T K_0^{-1} U.$$

This Schur-complement method was used in (Gill et al., 1990) and (Bartlett and Biegler, 2006), which describe primal and dual methods, respectively. Assuming a sufficient closeness of the initial working set to the optimal active set, the Schur-complement matrix C remains small. This assumption depends on the ability of the warm-starting strategy to accurately reflect the changes of the active in time. The motivation of the previously mentioned work was to solve the quadratic programming subproblems arising in the sequential quadratic programming method (SQP) for nonlinear optimization. In SQP, the nonlinear inequality constraints of the original problem are linearized, and as the solution converges the linearized constraints' active sets also converge. Thus, employing warm-starts of the QPs results in rapid convergence of the ASMs. The primal Schur-complement approach was successfully implemented in (Betts, 1994) to solve sparse nonlinear problems.

In classical ASMs, a single change of the working set occurs per iteration. This corresponds to a solve of the form $K_0^{-1}u$ to form the Schur-complement matrix C. In contrast, PDASM makes multiple changes to the working set per iteration, modifying the Schur-complement method to solves of the form $K_0^{-1}U$,

where U is a matrix encoding the changes of the working set. If warm-starting is used, one expects that the number of columns of U will not be large. Computing $K_0^{-1}U$ amounts to l solves of the form

$$K_0 x_i = u_i \quad i = 1, ..., l,$$

which direct methods can efficiently solve upon factorization of K_0. Furthermore, a parallel implementation of the solve with K_0 can significantly improve the performance of the Schur-complement method. In an ideal sense, if the solves were fully parallelized, the duration of an iteration of PDASM and the classical ASMs would be roughly equal. Since PDASM requires far fewer iterations than classical ASMs to make the equivalent changes to an initial working set, PDASM is expected to perform much better. We demonstrate this improvement via a parallel implementation, called SCHURPA (for SCHUR-complement method in PArallel), on the GPU using NVIDIA's CUDA.

1.3 Contributions of Thesis

Our contribution in this thesis is a solution framework which utilizes SCHURPA to solve the QP subproblems of SP. To further elucidate, we:

- introduce SCHURPA: a Schur-complement approach to solving the saddle point systems arising in PDASM;
- integrate SCHURPA into SP. We reformulate the problem of friction to be amenable to SCHURPA, and show that in the case of the simulator the saddle point systems may be phrased as banded SPD systems;
- demonstrate the effectiveness of SCHURPA with a GPU implementation.

The thesis is organized as follows. Chapter 2 overviews the general class of active-set methods and introduces the primal-dual active-set method, showing its equivalence to a semismooth Newton method. Chapter 3 introduces SCHURPA, our method for solving the saddle point systems in PDASM, which is conducive to a parallel implementation. We then describe the Eulerian solids simulation framework in Chapter 4. The problem of adding friction to the simulator is discussed, along with the staggered projections algorithm. Chapter 5 describes how we exploit problem structure of the QP subproblems in SP to solve them efficiently. Our parallel GPU implementation is discussed in Chapter 6. Results of SCHURPA on synthetic data and in the simulator are presented in Chapter 7. Conclusions and future work are given in Chapter 8.

1.3.1 Notation

For notation of quantities, we use upper case for matrices, lower case for vectors, and Greek letters generally refer to scalars. We denote the standard basis of $\mathbb{R}^n$ by $\{e_1, ..., e_n\}$ i.e. $e_i = [\delta_{ij}]_j \in \mathbb{R}^n$. The notation $V = \{v_i\}_{i \in S}$ defines the matrix V to have column vectors v_i indexed by the set S, and $V = \left\{v_i^T\right\}_{i \in S}$ is defined analogously to have row vectors v_i^T. Quantities with bars denote values at the next iteration (e.g. $\bar{x}$), while initial quantities will be subscripted by 0 (e.g. x_0). The active set of a primal variable x, denoted by $\mathcal{A}(x)$, is the subset of constraints which hold with equality at x. We denote an optimal solution to an optimization problem by x^*.

Chapter 2

Active-Set Methods for Quadratic Programming

Active-set methods (ASMs) are a class of algorithmic methodologies for solving QPs. A particularly attractive trait of ASMs is their ability to *warm-start* with an estimate of the solution. In many applications an estimate of $\mathcal{A}(x^*)$, the active set at the optimal solution, is known. As discussed in Section 1.2, ASMs can effectively utilize this estimate to drastically reduce the number of iterations required for convergence.

In this chapter we outline ASMs in the context of solving convex QPs, which are described in Section 2.1. Section 2.2 outlines the background and derivation of ASMs, and discusses the Karush-Kuhn-Tucker (KKT) Conditions. These conditions are necessary and sufficient conditions for obtaining optimality of a convex optimization problem. We then briefly review the classical ASMs in Section 2.3, the *primal* ASM and the *dual* ASM, and describe the method of updating factorizations common to ASMs. Unsatisfied with the limitations of these approaches, we turn our eyes towards greener pastures in Section 2.4 to the primal-dual ASM. Details of the algorithm can be found in (Ito and Kunisch, 2008).

2.1 Quadratic Programming

Convex quadratic programs (QPs) are an important class of optimization problems which may be stated as

$$\begin{aligned} &\text{minimize} && \frac{1}{2}x^T H x - c^T x \\ &\text{subject to} && a_i^T x \leq b_i, \quad i \in \mathcal{I}, \end{aligned} \tag{P}$$

where $c \in \mathbb{R}^n$, $x \in \mathbb{R}^n$, $\mathcal{I} = \{1, 2, ..., m\}$, $b = \{b_i\}_{i \in \mathcal{I}} \in \mathbb{R}^m$, $A = \{a_i\}_{i \in \mathcal{I}} \in \mathbb{R}^{n \times m}$ is the constraint Jacobian and $H \in \mathbb{R}^{n \times n}$, the Hessian, is a symmetric positive semidefinite matrix so that the problem is convex. The problem could also include equality constraints, but for the purposes of this thesis and without loss of generality they are left out. In this work we assume strict convexity (i.e. H is positive definite) and feasibility of (P) so that the optimal solution x^* exists and is unique.

2.2 Overview of Active-Set Methods

The class of ASMs (as well as most optimization algorithms) attempt to satisfy the Karush-Kuhn-Tucker conditions (Nocedal and Wright, 1999), which for (P) are given by

$$Hx^* - c + \sum_{i \in \mathcal{I}} \lambda_i^* a_i = 0 \tag{2.1a}$$

$$a_i^T x^* \leq b_i, \qquad i \in \mathcal{I} \tag{2.1b}$$

$$\lambda_i^* \geq 0, \qquad i \in \mathcal{I} \tag{2.1c}$$

$$(a_i^T x^* - b_i)\lambda_i^* = 0, \qquad i \in \mathcal{I} \tag{2.1d}$$

For convex problems these are necessary and sufficient conditions for primal

and dual variables x^* and λ^*, respectively, to be an optimal primal-dual pair, assuming some constraint qualification is satisfied. For example, the linear independence constraint qualification (LICQ) assumes the set $\{a_i\}_{i\in\mathcal{A}^*}$ is linearly independent. When the inequality and equality constraint functions are affine (as in (P)), no constraint qualification is needed. For the KKT conditions we can equivalently write

$$Hx^* - c + \sum_{i\in\mathcal{A}^*} \lambda_i^* a_i = 0 \tag{2.2a}$$

$$a_i^T x^* = b_i, \qquad i \in \mathcal{A}^* \tag{2.2b}$$

$$a_i^T x^* < b_i, \qquad i \in \mathcal{I}\backslash\mathcal{A}^* \tag{2.2c}$$

$$\lambda_i^* \geq 0, \qquad i \in \mathcal{A}^* \tag{2.2d}$$

$$\lambda_i^* = 0, \qquad i \in \mathcal{I}\backslash\mathcal{A}^*. \tag{2.2e}$$

Here we have used the fact that $\lambda_i^* = 0$ for $i \in \mathcal{I}\backslash\mathcal{A}^*$, which follows from Equation (2.1b), Equation (2.1d). Consider the equality-constrained subproblem

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2}x^T H x - c^T x \\ \text{subject to} \quad & a_i^T x = b_i, \qquad i \in \mathcal{A}^*. \end{aligned} \tag{2.3}$$

The KKT conditions of (2.3) are

$$H\hat{x} - c + \sum_{i\in\mathcal{A}^*} \hat{\lambda}_i a_i = 0 \tag{2.4a}$$

$$a_i^T \hat{x} = b_i, \qquad i \in \mathcal{A}^*. \tag{2.4b}$$

By strict convexity $\hat{x}$ is unique. Since Equation (2.4) is a subset of the KKT

conditions Equation (2.2), the optimal solution of (P) is also the optimal solution to (2.3), and since the solutions are unique, solving the subproblem also solves the full QP. Thus if $\mathcal{A}^*$ were known a priori, one could simply solve (2.3) to solve the original problem. Active set methods employ strategies for finding $\mathcal{A}^*$.

Beginning with a primal-dual pair (x^0, λ^0), Each iteration k of an ASM updates a working set $\mathcal{A}^k \subseteq \mathcal{I}$ of constraints which represents a guess of $\mathcal{A}^*$. The subproblem

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2}x^T H x - c^T x \\ \text{subject to} \quad & a_i^T x = b_i, \qquad i \in \mathcal{A}^k \end{aligned} \tag{2.5}$$

is solved, resulting in the optimal primal-dual pair (x^*, λ^*). The step directions

$$p = x^* - x^k$$

$$\mu = \lambda^* - \lambda^k$$

are then used to update the primal and dual iterates. A generic ASM is given in Algorithm 2.1.

Algorithm 2.1 Generic Active-set Method

Input: x^0, λ^0, $\mathcal{A}^0$
for $k = 0, 1, 2, ...$ **do**
  Solve (2.5) Defined by $\mathcal{A}^k$ in terms of step directions p and μ
  Step $x^{k+1} := x^k + \alpha p, \qquad \alpha \in [0, 1]$
    $\lambda^{k+1} := \lambda^k + \beta \mu, \qquad \beta \in [0, 1]$
  if the KKT conditions to (P) are satisfied **then**
    DONE; $x^* = x^{k+1}$ and $\lambda^* = \lambda^{k+1}$
  else
    Choose $\mathcal{A}^{k+1}$
  end if
end for
Output: x^*, λ^*

The computational "meat" of Algorithm 2.1 lies in solving (2.5), which may

be rewritten as the symmetric indefinite linear system (called the saddle point system)

$$\begin{pmatrix} H & A_k \\ A_k^T & \end{pmatrix} \begin{pmatrix} x \\ \lambda \end{pmatrix} = \begin{pmatrix} c \\ b_k \end{pmatrix}, \tag{2.6}$$

where $A_k = \{a_i\}_{i \in \mathcal{A}^k}$ and $b_k = \{b_i\}_{i \in \mathcal{A}^k}$. The saddle point matrix

$$K^k = \begin{pmatrix} H & A_k \\ A_k^T & \end{pmatrix}$$

is a special case of the symmetric indefinite matrix

$$\mathcal{M} = \begin{pmatrix} A & B \\ B^T & \end{pmatrix}.$$

The following theorem is stated without proof (see, e.g., (Nocedal and Wright, 1999)).

Theorem 2.1. *$\mathcal{M}$ is nonsingular iff A is positive definite on the nullspace of B^T.*

Due to the assumption that H is positive definite we get the following corollary.

Corollary 2.1. *K^k is nonsingular iff A_k^T is full row rank.*

From Corollary 2.1 it follows that $\mathcal{A}^0$ must prescribe a linearly independent set of constraints to ensure Equation (2.6) is solvable.

2.3 Classical ASMs

The two classical variants of ASMs are the primal and dual algorithms. The primal ASM, described in (Nocedal and Wright, 1999), chooses the primal step length α such that x^{k+1} is always feasible. The dual ASM, described in (Goldfarb and Idnani, 1983), chooses the dual step length β ensuring λ^{k+1} is feasible to the dual problem.

To ensure convergence, the primal ASM assumes the active constraints are linearly independent at each feasible vertex; the dual requires that H be positive definite. Convergence proofs for these ASMs are based on showing that $f(x^{k+1}) < f(x^k)$ in the primal version and $f(x^{k+1}) > f(x^k)$ in the dual version; thus cycling can never occur. Since there are a finite number of active-set subsets, the algorithms converge (Nocedal and Wright, 1999). While this theoretical argument allows for a possibly exponential number of steps in the size of the input data, in practice this is rarely observed. ASMs typically add correct and remove incorrect constraints to and from the working set; thus their running time is proportional to $|\mathcal{A}^0 - \mathcal{A}^*|$. This is why ASMs benefit from warm-starting with a good estimate of $\mathcal{A}^*$.

In the primal and dual algorithms one constraint is added or removed to the working set $\mathcal{A}^k$. Each saddle point system is not solved from scratch; rather, factorizations are updated as constraints are added and removed. In typical implementations these factors are dense, as maintaining sparse factorizations is difficult (Gill et al., 1987). We now give an example demonstrating the factorization update process. Consider the QP

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2} x^T H x - c^T x \\ \text{subject to} \quad & x \geq 0, \end{aligned} \tag{2.7}$$

which is a special case of (P) where $A = -I$ and $b = 0$.

At iteration k of Algorithm 2.1 $\mathcal{A}^k$ defines the set of bound and free components of x, which we denote by x_B and x_F, respectively, i.e. $B = \mathcal{A}^k$ and $F = \mathcal{I} \backslash \mathcal{A}^k$. The resulting saddle point system is then

$$\begin{bmatrix} H_{F,F} & H_{B,F}^T \\ H_{B,F} & H_{B,B} \end{bmatrix} \begin{bmatrix} x_F \\ x_B \end{bmatrix} - \begin{bmatrix} \lambda_F \\ \lambda_B \end{bmatrix} = \begin{bmatrix} c_F \\ c_B \end{bmatrix}. \tag{2.8}$$

Since $x_B = 0$ and $\lambda_F = 0$, we get

$$H_{F,F} x_F = c_F, \tag{2.9}$$

which can then be used to solve for λ_B via

$$\lambda_B = -c_B + H_{B,F} x_F. \tag{2.10}$$

Solving the saddle point system this way reduces the problem to a symmetric positive definite system of size $|F|$, the number of free variables.

Suppose iteration $k+1$ binds the i^{th} variable of x_F. Dropping the F subscript for simplicity, we can write the current saddle point system Equation (2.9) as

$$\begin{bmatrix} H_{11} & h_{21} & H_{31}^T \\ h_{21}^T & h_{22} & h_{32}^T \\ H_{31} & h_{32} & H_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_i \\ x_3 \end{bmatrix} = \begin{bmatrix} c_1 \\ c_i \\ c_3 \end{bmatrix}.$$

Then we can solve the next saddle point system by solving

$$\bar{H}\bar{x} = \bar{c},$$

where

$$\bar{H} = \begin{bmatrix} H_{11} & 0 & H_{31}^T \\ 0^T & 1 & 0^T \\ H_{31} & 0 & H_{33} \end{bmatrix}, \quad \bar{c} = \begin{bmatrix} c_1 \\ 0 \\ c_3 \end{bmatrix}.$$

Suppose we have a Cholesky factorization of $H_{F,F}$ which we write as

$$LL^T = \begin{bmatrix} L_{11} & & \\ l_{21}^T & \alpha & \\ L_{31} & l_{32} & L_{33} \end{bmatrix} \begin{bmatrix} L_{11}^T & l_{21} & L_{31}^T \\ & \alpha & l_{32}^T \\ & & L_{33}^T \end{bmatrix} = \begin{bmatrix} H_{11} & h_{21} & H_{31}^T \\ h_{21}^T & h_{22} & h_{32}^T \\ H_{31} & h_{32} & H_{33} \end{bmatrix}. \tag{2.11}$$

We now wish to determine the Cholesky factorization of $\bar{H}$:

$$\bar{L}\bar{L}^T = \begin{bmatrix} \bar{L}_{11} & & \\ \bar{l}_{21}^T & \bar{\alpha} & \\ \bar{L}_{31} & \bar{l}_{32} & \bar{L}_{33} \end{bmatrix} \begin{bmatrix} \bar{L}_{11}^T & \bar{l}_{21} & \bar{L}_{31}^T \\ & \bar{\alpha} & \bar{l}_{32}^T \\ & & \bar{L}_{33}^T \end{bmatrix} = \begin{bmatrix} H_{11} & 0 & H_{31}^T \\ 0^T & 1 & 0^T \\ H_{31} & 0 & H_{33} \end{bmatrix}.$$

Writing out the relevant equations of Equation (2.11) give

$$\begin{aligned} L_{11}L_{11}^T &= H_{11} \\ L_{31}L_{31}^T + l_{32}l_{32}^T + L_{33}L_{33}^T &= H_{33}. \end{aligned}$$

We can now solve for the components of the updated Cholesky factors:

$$
\begin{aligned}
\bar{L}_{11}\bar{L}_{11}^T &= H_{11} = L_{11}L_{11}^T \Rightarrow \bar{L}_{11} = L_{11} \\
\bar{L}_{11}\bar{L}_{31}^T &= H_{31} = L_{11}L_{31}^T \Rightarrow \bar{L}_{31} = L_{31} \\
\bar{L}_{11}\bar{l}_{21} &= 0 \Rightarrow \bar{l}_{21} = 0 \\
\bar{l}_{21}^T\bar{l}_{21} + \bar{\alpha}^2 &= 1 \Rightarrow \bar{\alpha} = 1 \\
\bar{L}_{31}\bar{l}_{21} + \bar{\alpha}\bar{l}_{32} &= 0 \Rightarrow \bar{l}_{32} = 0 \\
\bar{L}_{31}\bar{L}_{31}^T + \bar{l}_{32}\bar{l}_{32}^T + \bar{L}_{33}\bar{L}_{33}^T &= H_{33} = L_{31}L_{31}^T + l_{32}l_{32}^T + L_{33}L_{33}^T \\
\Rightarrow \bar{L}_{33}\bar{L}_{33}^T &= L_{33}L_{33}^T + l_{32}l_{32}^T.
\end{aligned}
$$

Thus, as can be seen from the last equation above, binding a variable amounts to a rank-one update of a Cholesky factorization. It can be shown that freeing a variable corresponds to a rank-one *down-date* (Davis and Hager, 2006). Updating Cholesky factorizations from rank-one changes are detailed in the seminal work (Gill et al., 1974). All updating procedures require $O(n^2)$ floating point operations as opposed to $O(n^3)$ required for a full factorization.

In summary, we see that classical ASMs admit several advantages over competing methods such as the interior point method (IP). As mentioned, in practice ASMs can take advantage of warm-starting, whereas IPs have difficulty doing so (Wright, 1987, Chapter 11). Also, updating factors is a cheap operation compared to full factorizations required in the iterates of IPs. The tradeoff comes from the large number of iterations in ASMs if the working set requires many changes.

We are therefore limited by classical ASMs because

- They do not exploit sparsity, which limits their applicability to large-scale problems, and

- The iterations are inherently sequential, and therefore they cannot be parallelized.

As we will show in Chapter 3 the Schur-complement approach ameliorates the first limitation by solving Equation (2.6) in a different way; rather than updating factors, an initial saddle point system is factorized, and future saddle point systems are expressed as solutions of the original saddle point matrix and a smaller Schur-complement matrix. We now address the second issue.

2.4 The Primal-Dual ASM

In an attempt to utilize the advantages of ASMs, we turn to a more recent approach: the primal-dual active-set method (PDASM). A key divergence from its classical counterparts is the ability to make multiple changes to the active set per iteration. We will see that combining this with a linear algebra technique to solve the saddle point systems results in a parallel ASM we have been searching for (see Chapter 3).

2.4.1 Derivation via the Semismooth Newton Method

We follow (Hintermüller et al., 2002) in deriving PDASM. In a nutshell, PDASM is defined by Algorithm 2.1 by choosing $\alpha = \beta = 1$ and selecting the next working set as

$$\mathcal{A}^{k+1} = \left\{ i \in \mathcal{I} \mid b_i - a_i^T x^{k+1} - \lambda_i^{k+1} < 0 \right\}. \tag{2.12}$$

PDASM can be derived from a generalized Newton method applied to a set of semismooth equations. A comprehensive overview of semismooth equations and algorithms can be found in (Ulbrich, 2011).

We define a function to be *semismooth* if it is differentiable almost everywhere, mathematically speaking (that is, everywhere except for a set of measure zero). Consider the function

$$\phi(a, b) = \min(a, b),$$

which is differentiable everywhere except along the line $a = b$ and thus is a semismooth function. Other examples of semismooth functions include convex functions and piecewise differentiable functions. We have the following useful property.

Lemma 2.4.1. $\phi(a, b) = 0 \Leftrightarrow ab = 0, \quad a \geq 0, \quad b \geq 0.$

Proof. Suppose $\phi(a, b) = 0$.

$$\begin{aligned} \text{Then} \quad & \min(a, b) = 0 \\ \Leftrightarrow \quad & a = 0 \text{ or } b = 0, \quad a \geq 0, \quad b \geq 0 \\ \Leftrightarrow \quad & ab = 0, \quad a \geq 0, \quad b \geq 0, \end{aligned}$$

as required. □

Corollary 2.4.2. *The KKT conditions to* (P) *may be reformulated as the solution to the equations*

$$F(x, \lambda) = \begin{cases} Hx + A\lambda - c \\ \Phi(x, \lambda) \end{cases} = 0, \tag{2.13}$$

where

$$\Phi_i(x, \lambda) = \phi(b_i - a_i^T x, \lambda_i). \tag{2.14}$$

Applying a generalized Newton method to semismooth equations was introduced in (Qi and Sun, 1993) and we briefly discuss the approach here. Let $F : \mathbb{R}^n \to \mathbb{R}^n$ and $D_F \subseteq \mathbb{R}^n$ be the set of points where F is differentiable. The set

$$\partial_B F(x) \equiv \left\{ \lim_{x_k \to x} \nabla F(x_k) \mid \{x_k\} \subseteq D_F \right\}$$

is called the B-subdifferential of F at x, where ∇F denotes the Jacobian of F. Intuitively this is the set of Jacobians nearing the point x in the limit. If F is differentiable at x it is clear that $\partial_B F(x) = \nabla F(x)$. We may now formulate a generalized Newton iteration as

$$x_{k+1} = x_k - V_k^{-1} F(x_k), \tag{2.15}$$

where $V_k \in \partial_B F(x_k)$. If F is semismooth at a solution x^* of $F(x) = 0$ and all $V \in \partial_B F(x^*)$ are nonsingular, local super-linear convergence can be proved (Ito and Kunisch, 2008). Recall that in the differentiable case, quadratic convergence occurs if $\nabla F(x^*)$ is nonsingular.

We now apply the generalized Newton method to equations Equation (2.13). The subdifferential of Φ is given by

$$\begin{aligned} \partial_B \Phi_i(x, \lambda) &= \partial_B \phi(b_i - a_i^T x, \lambda_i) \\ &= \begin{cases} (-a_i, 0)^T & \text{if } b_i - a_i^T x < \lambda_i \\ (0, e_i)^T & \text{if } b_i - a_i^T x > \lambda_i \\ \left\{(-a_i, 0)^T, (0, e_i)^T\right\} & \text{if } b_i - a_i^T x = \lambda_i \end{cases} \end{aligned}$$

We arbitrarily choose $V_i = (0, e_i)^T \in \partial_B \Phi_i(x, \lambda)$ in the final case. Let

$$\mathcal{A} = \left\{ i \in \mathcal{I} \mid b_i - a_i^T x^k - \lambda_i^k < 0 \right\}$$
$$\bar{\mathcal{A}} = \left\{ i \in \mathcal{I} \mid b_i - a_i^T x^k - \lambda_i^k \geq 0 \right\}.$$

Assume without loss of generality that the equations are ordered first by $\mathcal{A}$ and subsequently by $\bar{\mathcal{A}}$. Then, by taking the partial subdifferentials of F, the generalized Newton iteration Equation (2.15) applied to Equation (2.13) gives

$$\begin{pmatrix} H & A \\ -A_{\mathcal{A}}^T & 0 \\ 0 & I_{\bar{\mathcal{A}}} \end{pmatrix} \begin{pmatrix} x^{k+1} - x^k \\ \lambda^{k+1} - \lambda^k \end{pmatrix} = - \begin{pmatrix} Hx^k + A\lambda^k - c \\ b_{\mathcal{A}} - A_{\mathcal{A}}^T x^k \\ \lambda_{\bar{\mathcal{A}}}^k \end{pmatrix},$$

where $A_{\mathcal{A}} = \{a_i\}_{i \in \mathcal{A}}$, $I_{\bar{\mathcal{A}}} = \left\{e_i^T\right\}_{i \in \bar{\mathcal{A}}}$ and $b_{\mathcal{A}} = \{b_i\}_{i \in \mathcal{A}}$. Equivalently

$$Hx^{k+1} + A\lambda^{k+1} = c; \tag{2.16a}$$

$$A_{\mathcal{A}}^T x^{k+1} = b_{\mathcal{A}}; \tag{2.16b}$$

$$\lambda_{\bar{\mathcal{A}}}^{k+1} = 0, \tag{2.16c}$$

and after substituting Equation (2.16c) into Equation (2.16a) we get

$$Hx^{k+1} - c + \sum_{i \in \mathcal{A}} \lambda_i^{k+1} a_i = 0$$
$$a_i^T x^{k+1} = b_i, \qquad i \in \mathcal{A}.$$

Notice that these are exactly the KKT conditions for solving the equality-constrained QP

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2}x^T Hx - c^T x \\ \text{subject to} \quad & a_i^T x = b_i, \qquad i \in \mathcal{A}. \end{aligned} \tag{2.18}$$

Thus, we get an active-set procedure with the working set determined by $\mathcal{A}$, which is how we defined the update in Equation (2.12). Intuitively, rather than satisfying primal or dual feasibility, PDASM satisfies the complementarity condition

$$(a_i^T x^{k+1} - b_i)\lambda_i^{k+1} = 0 \qquad i \in \mathcal{I},$$

which follows from Equation (2.16b) and Equation (2.16c). Since either $b_i - a_i^T x^{k+1} = 0$ or $\lambda_i^{k+1} = 0$, the working set $\mathcal{A}$ chooses the former to hold if $b_i - a_i^T x^k < \lambda_i^k$; otherwise it ensures the latter holds. The super-linear convergence of the generalized Newton method applies to PDASM, and in practice very few iterations are required to obtain convergence (see Chapter 7). In combination with being an active-set method and therefore amenable to warm-starting, these make PDASM a very attractive approach for solving QPs. Algorithm 2.2 describes this stunningly simple yet powerful algorithm.

Algorithm 2.2 Primal-dual Active-set Method

Input: x^0, λ^0, $\mathcal{A}^0$
for $k = 0, 1, 2, ...$ **do**
  Solve (2.5) defined by $\mathcal{A}^k$ to compute the solution pair $\left(x^{k+1}, \lambda^{k+1}\right)$
  if the KKT conditions to (P) are satisfied **then**
    DONE; $x^* = x^{k+1}$ and $\lambda^* = \lambda^{k+1}$
  else
    $\mathcal{A}^{k+1} = \left\{i \in \mathcal{I} \mid b_i - a_i^T x^{k+1} - \lambda_i^{k+1} < 0\right\}$
  end if
end for
Output: x^*,λ^*

Global convergence of PDASM in certain cases has been proven for several classes of problems in (Ito and Kunisch, 2008), e.g., problems with M-matrix

Hessians. The two pitfalls in the successful convergence of PDASM are cycling and a potentially singular saddle point system.

Assuming that all saddle point systems are uniquely solvable (which is true, for example, if the constraint Jacobian has full rank), the algorithm will converge if and only if it is free of cycles. Typically global convergence proofs rely on a merit function which measures the progress of the algorithm to the solution and monotonically decreases, ensuring cycling cannot occur (Nocedal and Wright, 1999). The quantity $||F(x^k, \lambda^k)||$ is a natural choice, but unfortunately is inadequate without additional assumptions on the input data. In practice, convergence occurs for well-conditioned problems. We observe unconditional global convergence of PDASM for our application (see Chapter 7).

2.4.2 Comparison to Classical ASMs

Here we recap the similarities and differences of PDASM with the classical ASMs discussed in Section 2.3.

Similarities:

- being an ASM, PDASM yields the benefits of warm-starting and accurate solutions,
- the standard techniques for computing and updating factorizations may be applied to PDASM, and
- under certain conditions global convergence can be proved. PDASM is observed to converge as long as the saddle point systems are nonsingular.

Differences:

- there is no longer any guarantee of primal or dual feasibility,
- multiple changes to the working set are made each iteration, and

- due to super-linear convergence, few iterations are required and thus PDASM scales well to large problems.

In the next section, we exploit these unique features of PDASM to develop a parallel algorithm for solving QPs.

Chapter 3

SCHURPA : a Method for Solving Saddle Point Systems Arising from PDASM

The central ingredient for our parallel solver is the solution method to the saddle point system Equation (2.6). In this chapter we first show that the standard approach of updating factorizations for saddle point systems in PDASM cannot be parallelized in Section 3.1. We then introduce the Schur-complement method in Section 3.2, originally developed in (Gill et al., 1990). Their work used the method in the primal ASM to solve large, sparse problems. We show that using PDASM, the Schur-complement method not only permits the use of sparse data, but also produces an efficient parallel solver, which we call SCHURPA. Accuracy and complexity of SCHURPA are discussed in Section 3.3.

3.1 Limitations of Updating Factorizations

Our goal is to utilize the information PDASM provides each iteration, a set of l changes to the working set, to develop a parallel algorithm. Recall in the classical regime of ASMs a single change to the working set corresponds to an update-factorization procedure. In principle then, one could apply l such updates sequentially, but this would squander the use of parallelism and revert the algorithm's behaviour to that of the classical methods, i.e. each update would be cheap, but they could be numerous and inherently sequential.

Let us return to the example discussed in Section 2.3, where Cholesky factors of the reduced Hessian were updated via a rank-one change, corresponding to a single change of the working set. A natural approach in the case of multiple changes would be to update the Cholesky factors in parallel, accounting for all changes to the working set simultaneously. For example, suppose we bind two variables so that

$$LL^T = \begin{bmatrix} L_{11} & & & & \\ l_{21}^T & \alpha & & & \\ L_{31} & l_{32} & L_{33} & & \\ l_{41}^T & a & l_{43}^T & \beta & \\ L_{51} & l_{52} & L_{53} & l_{54} & L_{55} \end{bmatrix} \begin{bmatrix} L_{11}^T & l_{21} & L_{31}^T & l_{41} & L_{51}^T \\ & \alpha & l_{32}^T & a & l_{52}^T \\ & & L_{33}^T & l_{43} & L_{53}^T \\ & & & \beta & l_{54}^T \\ & & & & L_{55}^T \end{bmatrix} = \begin{bmatrix} H_{11} & h_{21} & H_{31}^T & H_{41}^T & H_{51}^T \\ h_{21}^T & h_{22} & h_{32}^T & h_{42}^T & h_{52}^T \\ H_{31} & h_{32} & H_{33} & H_{43} & H_{53}^T \\ h_{41}^T & h_{42} & h_{43}^T & h_{44} & h_{54}^T \\ H_{51} & h_{52} & H_{53} & h_{54} & H_{55} \end{bmatrix}.$$

The updated factors for the next iteration are

$$\bar{L}\bar{L}^T = \begin{bmatrix} \bar{L}_{11} & & & & \\ 0^T & 1 & & & \\ \bar{L}_{31} & 0 & \bar{L}_{33} & & \\ 0^T & 0 & 0^T & 1 & \\ \bar{L}_{51} & 0 & \bar{L}_{53} & 0 & \bar{L}_{55} \end{bmatrix} \begin{bmatrix} \bar{L}_{11}^T & 0 & \bar{L}_{31}^T & 0 & \bar{L}_{51}^T \\ & 1 & 0^T & 0 & 0^T \\ & & \bar{L}_{33}^T & 0 & \bar{L}_{53}^T \\ & & & 1 & 0^T \\ & & & & \bar{L}_{55}^T \end{bmatrix} = \begin{bmatrix} H_{11} & 0 & H_{31}^T & 0 & H_{51}^T \\ 0^T & 1 & 0 & 0^T & 0 \\ H_{31} & 0 & H_{33} & 0 & H_{53}^T \\ 0^T & 0 & 0^T & 1 & 0^T \\ H_{51} & 0 & H_{53} & 0 & H_{55} \end{bmatrix}.$$

Working out the equations as in the previous example yields the same rank-one update

$$\bar{L}_{33}\bar{L}_{33}^T = L_{33}L_{33}^T + l_{32}l_{32}^T,$$

as well as

$$\begin{aligned} \bar{L}_{53}^T &= \bar{L}_{33}^{-1}\left(l_{32}l_{52}^T + L_{33}L_{53}^T\right) \\ \bar{L}_{55}\bar{L}_{55}^T &= L_{55}L_{55}^T + l_{52}l_{52}^T + l_{54}l_{54}^T + L_{53}L_{53}^T - \bar{L}_{53}\bar{L}_{53}^T. \end{aligned}$$

The last equation requires two rank-one updates as well as two the size of L_{53}. If the variables being bound are close together or the second variable is near the end, this is a low rank update, but these are unknown a priori. As a second attempt, one could attempt to re-order the factors to produce a single column-row block change to induce a rank-l update. For example, suppose we bind l variables so that

$$\begin{aligned} PHP &= \begin{bmatrix} \bar{H}_{11} & \bar{H}_{21}^T \\ \bar{H}_{21} & \bar{H}_{22} \end{bmatrix} \\ P\bar{H}P^T &= \begin{bmatrix} \bar{H}_{11} & \\ & I_l \end{bmatrix} \end{aligned} \tag{3.1}$$

for some permutation matrix

$$P = \begin{bmatrix} P_{11} & P_{12} \\ P_{21} & P_{22} \end{bmatrix}.$$

We wish to find the cholesky factors

$$\bar{H}_{11} = \bar{L}_{11}\bar{L}_{11}^T$$

via updating the Cholesky factors

$$H = \begin{bmatrix} L_{11} & \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} L_{11}^T & L_{21}^T \\ & L_{22}^T \end{bmatrix}. \tag{3.2}$$

Substituting Equation (3.2) into Equation (3.1) yields

$$P_{11}^T \bar{L}_{11}\bar{L}_{11}^T P_{11} = L_{11}L_{11}^T - P_{11}^T \bar{H}_{21}^T P_{21} - P_{21}^T \bar{H}_{21}^T P_{11} - P_{21}^T \bar{H}_{22} P_{21}. \tag{3.3}$$

Unfortunately by construction P_{11} is not invertible unless $P = I$. Therefore Equation (3.3) cannot be used as an factorization update equation. We abandon the dream of updating factors and instead turn to another idea: the Schur-complement method.

3.2 The Schur-Complement Method

Recall that each iteration k of an ASM computes the solution to a saddle point system of the form

$$\begin{pmatrix} H & A_k \\ A_k^T & \end{pmatrix} \begin{pmatrix} x \\ \lambda \end{pmatrix} = \begin{pmatrix} c \\ b_k \end{pmatrix}.$$

A Schur-complement approach, developed in (Gill et al., 1990) for solving saddle point systems arising in a primal active-set method, computes an initial (sparse) factorization of the saddle point system

$$K_0 = \begin{pmatrix} H & A_0 \\ A_0^T & \end{pmatrix} \in \mathbb{R}^{(n+p)\times(n+p)}, \tag{3.4}$$

where $p = |\mathcal{A}^0|$ and $A_0 = \{a_i\}_{i\in\mathcal{A}^0}$ is given by the initial working set $\mathcal{A}^0$. The method relies on the following key observation:

Adding or removing a constraint to the working set is equivalent to symmetrically appending a column and a row to the current saddle point system.

To see this, consider the first iteration after the initial factorization of K_0. Suppose we add a constraint i. Then the new saddle point system is

$$\begin{pmatrix} H & A_0 & a_i \\ A_0^T & & \\ a_i^T & & \end{pmatrix} \begin{pmatrix} x \\ \lambda \\ \mu \end{pmatrix} = \begin{pmatrix} c \\ b_0 \\ b_i \end{pmatrix},$$

where μ is the Lagrange multiplier corresponding to the added constraint. Now suppose an initially active constraint i is removed from the working set. The new saddle point system can be written as

$$\begin{pmatrix} H & A_0 & \\ A_0^T & & e_i \\ & e_i^T & \end{pmatrix} \begin{pmatrix} x \\ \lambda \\ \gamma \end{pmatrix} = \begin{pmatrix} c \\ b_0 \\ 0 \end{pmatrix}.$$

This adds the final constraint $\lambda_i = 0$, which is required for a non-active constraint, as well as modifies the equation $a_i^T x = b_i$ to $a_i^T x + \gamma = b_i$, which can be satisfied for any x. Therefore, the constraint is "ignored" and the dummy

variable γ may be discarded.

In general, for a given iteration k suppose there are l changes made between $\mathcal{A}^{k-1}$ and $\mathcal{A}^k$. The system to be solved is given by the form

$$\begin{pmatrix} K_0 & U \\ U^T & \end{pmatrix} \begin{pmatrix} y \\ \pi \end{pmatrix} = \begin{pmatrix} c \\ w \end{pmatrix}, \tag{3.5}$$

where $U \in \mathbb{R}^{(n+p)\times l}$ and $w \in \mathbb{R}^l$. The variables x^k and λ^k can be obtained by "unscrambling" y^k and π^k. More precisely,

$$x_i^k = y_i^k, \qquad i = 1, ..., n$$

and

$$\lambda_i^k = \begin{cases} \pi_s^k & \text{if constraint } i\text{was the } s^{th}\text{constraint added} \\ 0 & \text{if constraint } i\text{was the } s^{th}\text{constraint removed} \\ y_{n+i}^k & \text{otherwise} \end{cases}$$

Although the system grows in size, by using the Schur-complement matrix defined as

$$C \equiv -U^T K_0^{-1} U,$$

solving Equation (3.5) is equivalent to solving the following systems:

$$\begin{aligned} K_0 v &= c \\ C\pi &= w - U^T v \\ K_0 y &= c - U\pi. \end{aligned} \tag{3.6a}$$

Thus, the work required at each iteration involves two solves with K_0 (and updating/solving with C). However, by solving for $R = K_0^{-1}U$ and noticing that $v = K_0^{-1}c$ is fixed throughout the algorithm we can write Equation (3.6) as

$$\begin{aligned} C &= -U^T R \\ C\pi &= w - U^T v \\ y &= v - R\pi \end{aligned}$$

and we see that l solves with K_0 is required to form C and solve for y and π.

As mentioned, an advantage of the Schur-complement approach is its abstraction of the solve with K_0 in Equation (3.5). More precisely, we black-box the solution of K_0 via a function `K0_solve` so that

$$R = \texttt{K0\_solve}\,(\texttt{U})\,.$$

The function `K0_solve` is application-dependent and implementations can take advantage of specific hardware such as the GPU. In general computing the function requires a matrix factorization and solving requires solving with the computed factors (see Section 3.3). We emphasize that, in contrast to updating factors, the solves

$$r_i = \texttt{K0\_solve}\,(u_i) \qquad i = 1, ..., l$$

are naturally parallelizable.

There is one final ingredient needed to make the algorithm above more efficient and ensure non-singularity of the Schur-complement matrix. Suppose a constraint $i \in \mathcal{A}_0$ is removed from $\mathcal{A}^k$ at some iteration k, adding the l^{th} col-

umn to U as described above. If at some later iteration, the constraint is then added back to $\mathcal{A}^k$, adding the l^{th} column to U will yield a rank deficient matrix and thus C will be singular. Instead of adding another column to U, we may simply *remove* the l^{th} column from U, as well as the l^{th} column and l^{th} row in C. Thus each modification to the working set of this form decreases the number of solves required by one. A similar argument holds for a constraint $i \notin \mathcal{A}_0$ which is added and subsequently removed from $\mathcal{A}^k$. In addition to avoiding a solve, removing a column also reduces the memory cost of the algorithm. In our GPU implementation, U is stored as a dense matrix, and minimizing the memory footprint on the GPU is crucial.

We are now ready to combine PDASM with the Schur-complement method, which we call SCHURPA (SCHUR in PArallel), given in Algorithm 3.2. An initialization step, given in Algorithm 3.1, computes the function `K0_solve` and solves the initial saddle point system. We describe application-specific implementations of `K0_solve` for the solids simulator in Chapter 5. The properties of Algorithm 3.2 are the topic of the next section.

Algorithm 3.1 SCHURPA-INIT

Input: $\mathcal{A}^0$
Factorize K_0 given by Equation (3.4) to produce the function `K0_solve`
Solve the initial saddle point system: $\begin{pmatrix} x^* \\ \lambda^* \end{pmatrix} = \texttt{K0\_solve}\left(\begin{pmatrix} c \\ b_0 \end{pmatrix}\right)$
Output: x^*, λ^*, `K0_solve`

3.3 Details of the SCHURPA Algorithm

By solving the saddle point systems via the Schur-complement method, SCHURPA avoids the need for updating factorizations. The approach works well when the cost of computing `K0_solve` is significantly more than the cost of solving with said function. This was the case in (Gill et al., 1990; Bartlett and

Algorithm 3.2 SCHURPA

Input: $\mathcal{A}^0, x^0, \lambda^0, \texttt{K}_0\texttt{\_\_solve}$
for $k = 0, 1, 2, ...$ **do**
  if the KKT conditions to (P) are satisfied **then**
    DONE; $x^* = x^k$ and $\lambda^* = \lambda^k$
  else
    $\mathcal{A}^{k+1} = \left\{ i \in \mathcal{I} \mid b_i - a_i^T x^k - \lambda_i^k < 0 \right\}$
  end if
  Compute U based on the changes between $\mathcal{A}^k$ and $\mathcal{A}^{k+1}$
  Compute $C = -U^T R$, where $R = \texttt{K}_0\texttt{\_\_solve}(\texttt{U})$
  Solve Equation (3.5) via Equation (3.7) to obtain (y^{k+1}, π^{k+1})
  Use y^{k+1} and π^{k+1} to obtain x^{k+1},λ^{k+1}
end for
Output: x^*, λ^*

Biegler, 2006), which employed the primal and dual methods, respectively, to solve large sparse QPs arising in the sequential quadratic programming (SQP) method. The Schur-complement method computes an initial sparse factorization, and updating factors are avoided. As the QPs in SQP converge, few active set changes are required to find the optimal solution. Our motivation, in contrast, is in physical simulation, where due to the temporal coherence the change in active-sets between time steps may be small but never converging. Thus we expect each iteration a nontrivial number of changes in the active-set, which suits the parallel linear solve of Algorithm 3.2. We also emphasize that their motivation was for solving sparse problems sequentially, while our motivation is solving structured dense problems in parallel (see Chapter 4). This exemplifies the utility of the block-box abstraction of $\texttt{K}_0\texttt{\_\_solve}$.

The computational complexity of SCHURPA depends on the number of iterations, the cost of computing $\texttt{K}_0\texttt{\_\_solve}$, and the cost of solving with $\texttt{K}_0\texttt{\_\_solve}$. Due to the super-linear convergence of any PDASM, Algorithm 3.2 typically converges in $O(1)$ iterations; in practice typically less than 10 iterations are required. For a generic implementation where a sparse matrix $K_0 \in \mathbb{R}^{n \times n}$ is factorized, Algorithm 3.1 has a complexity of $O\left(n^3\right)$ and solving with the result-

ing factors has complexity $O\left(n^2\right)$. The advantage is that only one factorization is required throughout the entire algorithm.

As the number of changes from $\mathcal{A}^0$ from $\mathcal{A}^k$ grows, so does the size of the Schur-complement matrix C. If C gets too large, the Schur-complement method could be inefficient and a re-factorization of the current saddle point system should be computed to create a new `K0_solve`. Additionally, since U is stored as a dense matrix, memory limitations must be considered in a GPU implementation. Therefore if either C gets sufficiently large or U cannot be stored in memory, we compute a new `K0_solve` from the current iteration. In our implementation, C is re-computed and factored each iteration, as in (Bartlett and Biegler, 2006).

Finally, we note that the Schur-complement method is less robust than methods updating factors via orthogonal transformations. If the initial saddle point system is ill-conditioned, all subsequent saddle point systems will compromise accuracy due to the Schur-complement approach. One possible amelioration to this is fixed-precision iterative refinement applied when constraint drift occurs. However, as our QPs are found to be reasonably well-conditioned and non-degenerate in practice, iterative refinement was not required to identify the optimal active set (see Chapter 7).

Chapter 4

The Eulerian Solids Simulator

Equipped with SCHURPA, a parallel PDASM for solving QPs, we describe our solution procedure for the frictional contact problem arising in an Eulerian deformable solid bodies simulator, the subject of this chapter.

Deformable body simulations typically work by keeping track of material points associated with the object and updating their positions in space. This approach is called *Lagrangian* simulation. In contrast, *Eulerian* simulation fixes points in space and simulates objects moving through this space by advecting them through the object's velocity field.

Introduced in (Levin et al., 2011), the Eulerian solids simulator handles complex, highly deformable solid objects undergoing frictionless contact. By discretizing the spatial domain into a regular grid, the simulator utilizes the finite volume method (Versteeg and Malalasekera, 2007). A parallel implementation of the simulator was developed on the GPU using CUDA (NVIDIA, 2012b). However, the contact was assumed to be frictionless.

Simulating friction is crucial for many interesting and complex phenomena. Consider hands grasping an object, fingers sliding across a surface, hair blowing in the wind, or cloth sliding off of a surface. Our goal is to imbue the simulator with the ability to simulate friction.

Section 4.1 provides a description of the simulator and a derivation of the contact QP solved at each time step of the simulation. We then introduce the frictional contact problem, and the staggered projections algorithm of (Kaufman et al., 2008) as our solution methodology, in Section 4.2. By appropriately reformulating the linearized friction cone in Section 4.3, we show that SCHURPA can be applied to the QPs that result from SP. The final SCHURPA-SP algorithm is outlined in Section 4.4.

4.1 Overview

4.1.1 Simulating Objects Without Contact

Here we briefly describe the method of simulating a single solid body in the simulator, the details of which can be found in (Levin et al., 2011).

Suppose we are interested in simulating an object starting at time t_0. The object's configuration at time t_0 is said to be in its *reference* configuration. A mapping between the object's reference configuration and its configuration at time t is given by

$$x : (\mathbb{R}^3, \mathbb{R}_+) \rightarrow (\mathbb{R}^3, \mathbb{R}_+)$$

$$x(X, t) = X + u(X, t).$$

The reference coordinate X is the coordinate of a point on the object in its reference configuration, and $u(X, t)$ is the change, or deformation, of that point at time t. In typical Lagrangian solids simulations, the object is discretized into particles. The reference coordinates X of these particles act as degrees of freedom (DOFs) of the system, completely determining the configuration of the object. The spatial positions of these particles are computed by integrating the

momentum equation

$$\rho \frac{dv(X)}{dt} = \nabla \cdot \sigma + \rho \mathrm{b},$$

where v is the velocity of the particle, ρ is the density, σ is the Cauchy stress, and b are the body forces acting on the particle. However, in the Eulerian framework, degrees of freedom are no longer reference coordinates, but rather spatial coordinates fixed on a grid in space. More precisely, we discretize space by a uniform grid $G = (\Delta x, \Delta y, \Delta z)$ of dimensions $(L_x \Delta x) \times (L_y \Delta y) \times (L_z \Delta z)$ and define our degrees of freedom by

$$x_{ijk} \quad = \quad (x_0 + i\Delta x, y_0 + j\Delta y, z_0 + k\Delta z),$$

where

$$i \in 0, ..., L_x, \quad j \in 0, ..., L_y, \quad k \in 0, ..., L_z.$$

By writing the velocity v as a function of x, the material derivative begets the modified momentum equation

$$\rho \left(\frac{\partial v}{\partial t} + v \cdot \nabla v \right) = \nabla \cdot \sigma + \rho b. \tag{4.1}$$

Integrating Equation (4.1) yields a velocity field with which the object may be advected through. For an integration region Ω centered around a node $N = (i, j, k)$ we assume a constant density ρ and integrate Equation (4.1) to obtain

$$m_N \left(\frac{\partial v_N}{\partial t} + v_N \nabla v_N \right) = f_N,$$

where m_N is the integrated mass and f_N are the integrated stress and body forces inside Ω. To compute the velocities at time $t + 1$, we use v_N^t in the

advection term $v_N \nabla v_N$ and apply a first order discretization to $\frac{\partial v_N}{\partial t}$, yielding

$$m_N v_N^{t+1} = m_N v_N^t + \Delta t(f_N - m_N D(v_N^t) v_N^t),$$

where the matrix D is the discrete Jacobian matrix using the first-order upwind differencing scheme. Assembling equations over the entire spatial grid for all objects yields the equation

$$M v^{t+1} = f^*, \tag{4.2}$$

where $M \in \mathbb{R}^{n \times n}$ is the globally assembled diagonal mass matrix, $v^{t+1} \in \mathbb{R}^n$ is the global velocity vector and $f^* \in \mathbb{R}^n$ is the global impulse vector. Once Equation (4.2) is solved, the reference coordinates are advected through the velocity field via the material derivative:

$$\begin{aligned} \frac{DX}{Dt} &= 0 \\ \Rightarrow \frac{\partial X}{\partial t} + v^{t+1} \cdot \nabla X &= 0, \end{aligned}$$

yielding the update equation

$$X^{t+1} = X^t - \Delta t D(v^{t+1}) X^t.$$

By tracking the reference coordinates, strain and stress can be properly computed and an elastic object will always return to its undeformed state.

4.1.2 Simulating Objects With Contact

Contact between bodies in the simulation is resolved by invoking Gauss' principle of least constraint (Lanczos, 1986), a variational formulation of classical mechanics, which states that the velocity is the solution to the following optimization problem:

$$\begin{aligned} v^{t+1} = \underset{v}{\text{argmin}} \quad & \frac{1}{2} v^T M v - v^T f^* \\ \text{subject to} \quad & v \in \mathcal{V}, \end{aligned} \tag{4.3}$$

where $\mathcal{V}$ is the constraint set. In the case of the simulator, constraints enforce non-interpenetration between objects.

Consider a colliding pair of objects A and B with contact surface Γ_{AB}. Non-interpenetration constraints are formulated on the velocity level so that the relative velocity along the unit normal is always nonnegative along the contact surface:

$$v^{rel} \cdot n \geq 0, \tag{4.4}$$

where $v^{rel} = v^A - v^B$ is the relative velocity and n is the unit normal to the surface. As in typical FEM fashion, we integrate Equation (4.4) in each grid cell Ω_c where contact occurs to obtain

$$\begin{aligned} & \int_{\Gamma_c} v^{rel} \cdot n d\Gamma \\ = & \int_{\Gamma_c} \left(v^A - v^B \right) \cdot n d\Gamma \geq 0, \end{aligned} \tag{4.5}$$

where $\Gamma_c = \Omega_c \cap \Gamma_{AB}$. By expressing velocities in terms of nodal shape functions we get

$$v(x) = \sum_{N=1}^{L} \phi_N(x) v_N, \tag{4.6}$$

where $v_N \in \mathbb{R}^3$, $L = L_x L_y L_z$ and ϕ_N is a scalar shape function. In the implementation of the simulator we use trilinear shape functions as in (Levin

et al., 2011). Substituting in Equation (4.6) to Equation (4.5) yields

$$\begin{aligned} & \sum_{N=1}^{L} \left(\int_{\Gamma_c} \phi_N n d\Gamma \right) \cdot v_N^A - \sum_{N=1}^{L} \left(\int_{\Gamma_c} \phi_N n d\Gamma \right) \cdot v_N^B \\ = \quad & j_c^T v, \end{aligned}$$

where

$$j_c \quad = \quad \left\{ \left(\int_{\Gamma_c} \phi_N n_x d\Gamma, \int_{\Gamma_c} \phi_N n_y d\Gamma, \int_{\Gamma_c} \phi_N n_z d\Gamma \right)^T \right\}_{N=1,\ldots,L} \in \mathbb{R}^n$$

The constraints are assembled into a global constraint Jacobian matrix

$$J = \{j_c\}_{c=1,\ldots,m} \in \mathbb{R}^{n \times m}$$

and the resulting QP solved at each time step is given by

$$\begin{aligned} v^{t+1} = \underset{v}{\text{argmin}} \quad & \frac{1}{2} v^T M v - v^T f^* \\ \text{subject to} \quad & J^T v \geq 0. \end{aligned} \tag{4.7}$$

Because contact only occurs on surfaces of objects, the number of constraints is much smaller than the total number of degrees of freedom, i.e., $m \ll n$. In such cases it is useful to transform (4.7) into its *dual* formulation, given by

$$\begin{aligned} \alpha^{t+1} = \underset{\alpha}{\text{argmin}} \quad & \frac{1}{2} \alpha^T J^T M^{-1} J \alpha + \alpha^T (J^T M^{-1} f^*) \\ \text{subject to} \quad & \alpha \geq 0, \end{aligned} \tag{4.8}$$

where $\alpha \in \mathbb{R}^m$. The velocities can then be recovered via

$$v^{t+1} = M^{-1}(f^* + c^{t+1}), \tag{4.9}$$

where

$$c^{t+1} \equiv J\alpha^{t+1}$$

can be interpreted as the generalized contact impulse with α giving the magnitudes of those impulses. Reducing to the dual results in a problem of size m. Due to the diagonal structure of M, formulating (4.8) is computationally tractable. We now turn to the more challenging case of simulating frictional contact.

4.2 Adding Friction Using Staggered Projections

4.2.1 Friction Model

We begin our derivation of frictional contact with Coulomb's law of friction. Consider a contact point p for two objects A and B in contact. Coulomb's law states that the frictional impulse f_p must directly oppose the relative velocity v^{rel} at the contact point, and must lie in the feasible set

$$\{f_p \in \mathcal{T}_p \ : \ ||f_p|| \leq \mu\alpha\},$$

where $\mathcal{T}_p$ is the tangent plane to the contact point, α is the magnitude of the normal impulse, and $0 \leq \mu \leq 1$ is the coefficient of friction dependent on the material properties. We model friction using a linearized, isotropic Coulomb friction law (Stewart, 2000). We define a set of $2l$ symmetric unit length vectors

$\{\mathcal{T}_i\}_{i=1,...,2l} \in \mathcal{T}_p$ so that

$$\mathcal{T}_{i+1} = -\mathcal{T}_i \qquad i = 1, 3..., 2l-1.$$

The frictional impulse is then expressed as

$$f_p = \mathcal{T}\beta_p, \tag{4.10}$$

where $\mathcal{T} = \{\mathcal{T}_i\}_{i=1,...,2l} \in \mathbb{R}^{3\times 2l}$ and $\beta_p \in \mathbb{R}^{2l}$ gives the magnitude of each tangent direction. Note that, due to the symmetry of the tangent vectors $\mathcal{T}_i$, $\mathcal{T}$ is necessarily rank deficient. We address the algorithmic issues this implies in Section 4.3. We discretize β_p using piecewise constants so that for each contact cell c the frictional impulse is

$$f_p = \mathcal{T}\beta_c,$$

where $\beta_c \in \mathbb{R}^{2l}$ is a constant within the contact cell. The linearized Coulomb law may then be described as the set of feasible points

$$F_c = \left\{ f_p = \mathcal{T}\beta_c \ : \ e^T\beta_c \leq \mu_c\alpha_c, \ \beta_c \geq 0 \right\}, \tag{4.11}$$

where $e = (1, ..., 1)^T$, $\mu_c \geq 0$ is the coefficient of friction and $\alpha_c \geq 0$ is the contact magnitude. F_c represents the convex hull of the columns of $\mathcal{T}$. The feasible set over all contacts can now be written in terms of the tangent magnitudes as

$$\left\{ \beta \mid E^T\beta \leq \mathrm{diag}(\mu)\alpha, \ \beta \geq 0 \right\}, \tag{4.12}$$

where $\beta \in \mathbb{R}^{2lm}$ is the global tangent magnitude vector, $E \in \mathbb{R}^{2lm\times m}$ is globally assembled from e, and $\mathrm{diag}(\mu)$ constructs the diagonal matrix of the glob-

ally assembled vector μ. Turning from feasibility to optimality, we rephrase Coloumb's law as a variational principle, given by the Maximal Dissipation Principle (Moreau, 1973):

For a contact point p, the frictional force f maximizes the dissipation among all feasible forces.

The dissipation is equal to the rate of negative work and defined as $-f_p^T v^{rel}$. We integrate over the contact surface Γ_c of the contact cell c to obtain the dissipation for a given contact:

$$\begin{aligned}
&\int_{\Gamma_c} -f_p^T v^{rel} d\Gamma \\
= \; & -\left(\int_{\Gamma_c} f_p^T v^A d\Gamma - \int_{\Gamma_c} f_p^T v^B d\Gamma \right) \\
= \; & -\beta_c^T \left(\sum_{N=1}^{L} \int_{\Gamma_c} \mathcal{T}^T \phi_N v_N^A - \sum_{N=1}^{L} \int_{\Gamma_c} \mathcal{T}^T \phi_N v_N^B \right) \qquad (4.13) \\
= \; & -\beta_c^T T_c^T v, \qquad (4.14)
\end{aligned}$$

where

$$T_c = \left\{ \left(\int_{\Gamma_c} \phi_N \mathcal{T}_x^T d\Gamma, \int_{\Gamma_c} \phi_N \mathcal{T}_y^T d\Gamma, \int_{\Gamma_c} \phi_N \mathcal{T}_z^T d\Gamma \right)^T \right\}_{N=1,\ldots,L} \in \mathbb{R}^{n \times l}.$$

Equation (4.13) follows from expressing the velocities as a linear combination of the shape functions defined by Equation (4.6). Summing over all contacts yields the total dissipation:

$$\begin{aligned}
& -\sum_{c=1}^{m} \beta_c^T T_c^T v \\
= \; & -\beta^T T^T v,
\end{aligned}$$

where

$$T = \{T_c\}_{c=1,..,m} \in \mathbb{R}^{n \times 2lm} \tag{4.15}$$

is the globally assembled subspace of generalized friction impulses and

$$f := T\beta$$

is the generalized friction impulse. Maximizing the dissipation over the feasible set Equation (4.12) yields the optimization problem

$$\begin{aligned} \beta^{t+1} = \underset{\beta}{\text{argmax}} \quad & -\frac{1}{2}\beta^T T^T v^{t+1} \\ \text{subject to } \beta \geq 0, \quad & E^T\beta \leq \text{diag}(\mu)\alpha. \end{aligned} \tag{4.16}$$

We can transform Equation (4.16) into a QP in β by adding the generalized frictional impulse to the momentum Equation (4.9):

$$\begin{aligned} v^{t+1} &= M^{-1}(f^* + c^{t+1} + f^{t+1}) \\ &= M^{-1}(f^* + c^{t+1} + T\beta^{t+1}), \end{aligned}$$

yielding the friction QP

$$\begin{aligned} \beta^{t+1} = \underset{\beta}{\text{argmin}} \quad & \frac{1}{2}\beta^T T^T M^{-1} T\beta + \beta^T (T^T M^{-1}(c^{t+1} + f^*)) \\ \text{subject to } \beta \geq 0, \quad & E^T\beta \leq \text{diag}(\mu)\alpha^{t+1}. \end{aligned} \tag{4.17}$$

Notice that (4.17) is dependent on the solution of the contact QP (4.8). Conversely, taking into account the generalized frictional impulse f^{t+1} modifies

(4.8) to

$$\begin{aligned} \alpha^{t+1} = \underset{\alpha}{\text{argmin}} \quad & \frac{1}{2}\alpha^T J^T M^{-1} J\alpha + \alpha^T (J^T M^{-1}(f^{t+1} + f^*)) \\ \text{subject to} \quad & \alpha \geq 0. \end{aligned} \tag{4.18}$$

We now have two coupled convex QPs: (4.17), yielding the optimal friction impulse magnitudes β^{t+1} given the contact impulse c^{t+1}, and (4.18), yielding the optimal contact impulse magnitudes α^{t+1} given the friction impulse f^{t+1}. This intrinsic coupling is what makes frictional contact a challenging problem, as finding the solution of the coupled problem is equivalent to solving a global minimization of a *non*-convex QP, which is in general NP-hard (Kaufman et al., 2008). We adopt the method of staggered projections to solve the frictional contact problem.

4.2.2 The Staggered Projections Algorithm

Staggered projections, introduced in (Kaufman et al., 2008), is a robust, computationally efficient method to simulate discrete frictional contact response for a class of systems with finite degrees of freedom described in generalized coordinates. We briefly outline the algorithm now.

Recall the coupled friction and contact QPs (4.17) and (4.18), respectively. Beginning with an initial estimate $f^0 = T\beta^0$ of the friction impulse, SP solves the following set of QPs at iteration s:

$$\begin{aligned} \alpha^{s+1} = \underset{\alpha}{\text{argmin}} \quad & \frac{1}{2}\alpha^T J^T M^{-1} J\alpha + \alpha^T (J^T M^{-1}(f^s + f^*)) \\ \text{subject to} \quad & \alpha \geq 0, \end{aligned} \tag{C}$$

$$\begin{aligned} \beta^{s+1} = \underset{\beta}{\text{argmin}} \quad & \frac{1}{2}\beta^T T^T M^{-1} T\beta + \beta^T (T^T M^{-1}(c^{s+1} + f^*)) \\ \text{subject to} \quad & \beta \geq 0, \quad E^T\beta \leq \text{diag}(\mu)\alpha^{s+1}. \end{aligned} \tag{F}$$

Note that if $\beta^0 = 0$ this reduces to the case of frictionless contact discussed in Section 4.1.2. The staggered projections algorithm is given in Algorithm 4.1. The convergence criterion is given by

$$\texttt{rel\_error} = \frac{(f_c^{s+1} - f_c^s)^T M^{-1} (f_c^{s+1} - f_c^s)}{f^{sT} M^{-1} f^s} \leq \texttt{tol} \tag{4.19}$$

where `rel_error` specifies the relative kinetic metric error and `tol` is a user-specified tolerance. Similar to ASMs, SP benefits greatly from warm starts. Up to two orders of magnitude speed-ups were observed in Kaufman (2009). Since each QP subproblem is strictly convex and feasible, the algorithm will never fail on a given iteration. While global convergence is not guaranteed, in practice few iterations are required, especially if warm-starting is employed.

SP can be interpreted as a fixed point predictor correction scheme (Kaufman, 2009). The predicted impulse f^* is corrected by the coupled projections

$$\begin{aligned} f^{s+1} &= \mathrm{P}_{F(\alpha^s)}(f^* - c^s) \\ c^{s+1} &= \mathrm{P}_C(f^* - f^{s+1}), \end{aligned}$$

where P is the projection operator and

$$\begin{aligned} F(\alpha) &:= \left\{ T\beta \mid E^T\beta \leq \mathrm{diag}(\mu)\alpha,\ \beta \geq 0 \right\}, \\ C &:= \{ J\alpha \ :\ \alpha \geq 0 \} \end{aligned}$$

represent the set of admissible friction and contact impulses, respectively.

We are now in a position to incorporate SCHURPA into SP. Assuming for the moment that SCHURPA can solve (C) and (F), notice that only the linear terms of the objective function and right-hand-sides of the contact and friction QPs change each iteration of Algorithm 4.1. In other words, the Hessians and

constraint matrices remain constant. Thus the initial factorization computed in SCHURPA can be used not only within a single QP solve, but *throughout the entire staggered projections algorithm!* This idea is formalized in Section 4.4. However, we must first investigate the applicability of SCHURPA to the contact and friction QPs. In particular, the friction QP (F) must be reformulated to ensure the resulting saddle point systems occurring in SCHURPA are solvable, which we do in the next section.

Algorithm 4.1 Staggered Projections

Given $\beta^0, \mathcal{A}_c^0, \mathcal{A}_f^0, tol$
`rel_error` $= \infty$
for $s = 0, 1, 2, ...$ **do**
  Solve the contact QP (C) for α^{s+1}
  Solve the friction QP (F) for β^{s+1}
  Compute `rel_error` given by Equation (4.19)
  if `rel_error` $<$`tol` **then**
    BREAK; $\alpha^{t+1} = \alpha^{s+1}$ and $\beta^{t+1} = \beta^{s+1}$
  end if
end for

$f^{t+1} = T\beta^{t+1}$
$c^{t+1} = J\alpha^{t+1}$
$v^{t+1} = M^{-1}(f^* + f^{t+1} + c^{t+1})$

4.3 Applying SCHURPA to Frictional Contact

Recall from Section 2.4 local convergence of PDASM is guaranteed under the assumption that the saddle point systems are all nonsingular. In the case of the contact QP (C), this assumption holds, which we prove in Theorem 4.1 using the following lemma.

Lemma 4.1. *The Jacobian matrix J of* (4.7) *is full rank.*

Proof. We prove the lemma on a 2D grid using bilinear shape functions by assuming for a contradiction the Jacobian has linearly dependent columns; the

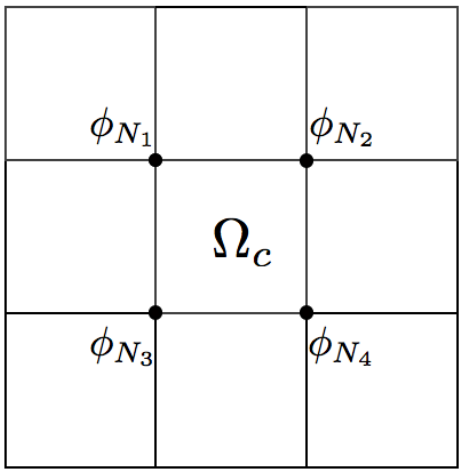


Figure 4.1: Nodal shape functions for a contact cell Ω_c on a 2D grid.

extension to 3D using trilinear shape functions as in the simulator is straightforward. Consider the four shape functions ϕ_N for $N \in \mathcal{N}^c := \{N_1, N_2, N_3, N_4\}$ providing local support to a contact cell Ω_c; see Figure 4.1. Assume the associated contact surface Γ_c has a point strictly in the interior so that

$$j_N = \left(\int_{\Gamma_c} \phi_N n_x d\Gamma, \int_{\Gamma_c} \phi_N n_y d\Gamma, \int_{\Gamma_c} \phi_N n_z d\Gamma \right) \neq (0,0,0)^T , \tag{4.20}$$

where $j = j_c$ is the contact constraint and $N \in \mathcal{N}^c$. Let $C \subseteq \{1, ..., m\}$ be a subset of the constraints containing a linearly dependent set so that

$$\sum_{c \in C} \alpha_c j_c = 0, \tag{4.21}$$

for scalars $\alpha_c \neq 0$, $c \in C$. We define (x_c, y_c) to be the grid center of the grid cell Ω_c. The idea now is to find a constraint $\bar{c} \in C$ for which $\bar{j} = j_{\bar{c}}$ has unique positions which contain nonzero elements, thus contradicting Equation (4.21). One can always find such a constraint $\bar{c}$ by choosing the grid cell $\Omega_{\bar{c}} \in \{\Omega_c\}_{c \in C}$

such that

$$\begin{aligned} x_{\bar{c}} &\geq x_c \quad c \in C, \\ y_{\bar{c}} &\geq y_c \quad c \in K \end{aligned}$$

where $K = \{c \in C \mid x_c = x_{\bar{c}}\}$. In other words, $\bar{c}$ selects the constraint cell with maximum x coordinates and maximal y coordinates. Notice that this cell is the only cell with support from the shape function ϕ_{N_2} where $N_2 \in \mathcal{N}^{\bar{c}}$ (see Figure 4.1). This implies the nonzero entry in $\bar{j}$ corresponding to $\bar{j}_{N_2} \neq 0$ (which is nonzero due to Equation (4.20)) is the only constraint with a nonzero entry in that position. Thus

$$\bar{j} \neq \frac{1}{\alpha_{\bar{c}}} \sum_{c \in C \setminus \bar{c}} \alpha_c j_c,$$

contradicting Equation (4.21). We therefore conclude the columns of J must be linearly independent, and J is full rank. □

Theorem 4.1. *The saddle point systems resulting from PDASM applied to* (C) *will always be nonsingular and PDASM will locally converge super-linearly.*

Proof. From Lemma 4.1 J is full rank, and thus the Hessian of (C) is nonsingular. Since the constraint matrix of (C) is simply the identity, any subset of constraints are linearly independent. Therefore by Theorem 2.1 the saddle point systems will be nonsingular, as required. □

Because SCHURPA is a PDASM, the following corollary immediately follows from the above theorem.

Corollary 4.1. *SCHURPA applied to the contact QP* (C) *locally converges super-linearly.*

We now tackle the friction QP. Unfortunately, in the form given by (F) the saddle point systems can become singular because of the Hessian matrix. Recall the tangent matrix is given by

$$T = \{T_c\}_{c=1,..,m},$$

where

$$T_c = \left\{ \left(\int_{\Gamma_c} \phi_N \mathcal{T}_x^T d\Gamma, \int_{\Gamma_c} \phi_N \mathcal{T}_y^T d\Gamma, \int_{\Gamma_c} \phi_N \mathcal{T}_z^T d\Gamma \right)^T \right\}_{N=1,...,L} \in \mathbb{R}^{n \times l}.$$

As previously mentioned, the symmetry of the unit length vectors $\mathcal{T}_i$ used to construct the linearized friction cone results in the matrices T_c, and hence T, to become rank deficient. This implies (F) is a semidefinite QP with a singular Hessian. The infinite solutions arise due to the nullspace of T which induces infinite representations of a single frictional impulse. More precisely, the symmetry implies

$$T_{i+1} = -T_i \qquad i = 1, 3..., 2ml - 1 \tag{4.22}$$

and thus

$$\begin{aligned} f = T\beta &= \sum_{i=1,...,2ml} \beta_i T_i \\ &= \sum_{i=1,3...2ml-1} (\beta_{i+1} - \beta_i) T_i. \end{aligned} \tag{4.23}$$

Even if we assume the set $\{T_i\}_{i=1,3...2ml-1}$ is linearly independent, only the differences $\beta_{i+1} - \beta_i$ uniquely define a friction impulse, and the Hessian $T^T M^{-1} T \in \mathbb{R}^{2lm \times 2lm}$ of (F) has rank lm. To make matters worse, we have $2ml + m$ constraints vs $2ml$ unknowns. Both the rank deficiency of the Hessian

and possible linear dependence of working set constraints can cause a singular saddle point system in the course of PDASM, prohibiting the direct application to (F). These deficiencies are not just theoretical. In (Daryina and Izmailov, 2009), it was observed that if the number of constraints is greater than the number of variables, PDASM fails a majority of the time. Their experiments applied to generic QPs also showed that if the Hessian is fairly rank deficient PDASM becomes an unreliable method. Therefore, we must reformulate (F) in order to apply PDASM.

We can reformulate the friction QP to an equivalent convex QP with a positive definite Hessian for the case $l = 2$ (we shall henceforth assume $l = 2$). Let $\hat{\mathcal{T}}$ be an orthonormal basis for the tangent surface $\mathcal{T}_p$ of a contact point p. We can then rewrite the tangent impulse as

$$f_p = \hat{\mathcal{T}}\beta_c, \tag{4.24}$$

where $\beta_c \in \mathbb{R}^2$ is taken as constant in a contact cell c as before. In the case of $l = 2$, the linearized friction cone is simply expressed by the l_1-norm

$$F_c = \left\{ \hat{\mathcal{T}}\beta_c \mid ||\hat{\mathcal{T}}\beta_c||_1 \leq \mu_c \alpha_c \right\}, \tag{4.25}$$

which can be rewritten as

$$F_c = \left\{ \hat{\mathcal{T}}\beta_c \mid \hat{E}_c^T \beta_c \leq \mathrm{diag}(\hat{\mu}_c)\hat{\alpha}_c) \right\}, \tag{4.26}$$

where

$$\begin{aligned}\hat{E}_c &= \begin{pmatrix} 1 & -1 & 1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix}, \\ \hat{\mu}_c &= (\mu_c, \mu_c, \mu_c, \mu_c)^T, \\ \hat{\alpha}_c &= (\alpha_c, \alpha_c, \alpha_c, \alpha_c)^T.\end{aligned} \tag{4.27}$$

As before, we assemble the global quantities $\hat{E} \in \mathbb{R}^{2m\times 4m}$, $\hat{\mu} \in \mathbb{R}^{4m}$, and $\hat{\alpha} \in \mathbb{R}^{4m}$. The global feasible set is given as

$$\left\{\beta \mid \hat{E}^T\beta \leq \operatorname{diag}(\hat{\mu})\hat{\alpha},\ \beta \geq 0\right\}, \tag{4.28}$$

and the generalized tangent matrix is

$$\hat{T} = \{\hat{T}_c\}_{c=1,..,m} \in \mathbb{R}^{n\times 2lm} \tag{4.29}$$

where

$$\hat{T}_c \quad = \quad \left\{ \left(\int_{\Gamma_c} \phi_N \hat{\mathcal{T}}_x^T d\Gamma, \int_{\Gamma_c} \phi_N \hat{\mathcal{T}}_y^T d\Gamma, \int_{\Gamma_c} \phi_N \hat{\mathcal{T}}_z^T d\Gamma \right)^T \right\}_{N=1,\ldots,L} \in \mathbb{R}^{n\times l}.$$

The modified friction problem is now

$$\begin{aligned}\beta^{s+1} = \underset{\beta}{\operatorname{argmin}} \quad & \frac{1}{2}\beta^T\hat{T}^T M^{-1}\hat{T}\beta + \beta^T(\hat{T}^T M^{-1}(c^{s+1} + f^*)) \\ \text{subject to} \quad & \hat{E}^T\beta \leq \operatorname{diag}(\hat{\mu})\hat{\alpha}^{s+1}.\end{aligned} \tag{F2}$$

The advantage of the formulation (F2) is that $\hat{T}$ has full rank.

Lemma 4.2. *The tangent matrix $\hat{T}$ of* (F2) *is full rank.*

Proof. The proof is analogous to the proof of Lemma 4.1 where quantities corresponding to constraint j_c are replaced with quantities corresponding to $\hat{T}_c$. □

Since $\hat{T}$ is full rank, the Hessian $\hat{T}^T M^{-1}\hat{T}$ of (F2) is positive definite, and the problem size has reduced by half from $4m$ in (F) to $2m$. Although we have $4m$ constraints, the following theorem guarantees that PDASM never selects a linearly dependent working set, and therefore the saddle point systems will be nonsingular.

Theorem 4.2. *Assuming the initial working set $\mathcal{A}^0$ results in a linearly independent set of constraints, The saddle point systems resulting from PDASM applied to* (F2) *will always be nonsingular and PDASM will locally converge super-linearly.*

Proof. We will show by induction that each step of Algorithm 2.2 selects a set of linearly independent constraints, which by Theorem 2.1 implies that the resulting saddle point system is nonsingular.

Base Case: By assumption $\mathcal{A}^0$ is a linearly independent set of constraints, so the initial saddle point system is nonsingular.

Induction Step: Assume at step k of Algorithm 2.2 that $\mathcal{A}^k$ defines a linearly independent set of constraints from (F2), and for a contradiction assume $\mathcal{A}^{k+1}$ selects a linearly *dependent* set. The form of $\hat{E}_c$ implies that $\mathcal{A}^{k+1}$ must contain a pair of constraints of the form

$$\beta_i + \beta_{i+1} = \gamma_i \quad \text{and} \quad -\beta_i - \beta_{i+1} = \gamma_i \tag{4.30}$$

or

$$\beta_i - \beta_{i+1} = \gamma_i \quad \text{and} \quad -\beta_i + \beta_{i+1} = \gamma_i \tag{4.31}$$

for some $i = 1, 3, ..., 2m-1$ and $\gamma_i := \mu_{\frac{i+1}{2}} \alpha_{\frac{i+1}{2}} > 0$. Assume WLOG the pair Equation (4.30) is chosen, and denote these constraints by j and $j+1$. Then by definition of Algorithm 2.2

$$\lambda_j^{k+1} > \gamma_i - (\beta_i^{k+1} + \beta_{i+1}^{k+1}) \tag{4.32}$$

$$\lambda_{j+1}^{k+1} > \gamma_i + \beta_i^{k+1} + \beta_{i+1}^{k+1}. \tag{4.33}$$

Case 1: $\lambda_j^{k+1} \neq 0$ or $\lambda_{j+1}^{k+1} \neq 0$.

Assume WLOG $\lambda_j^{k+1} \neq 0$. By our assumption $\mathcal{A}^k$ defined a linearly independent set. Thus $j \in \mathcal{A}^k$, $j+1 \notin \mathcal{A}^k$ and

$$\begin{aligned} \lambda_{j+1}^{k+1} &= 0 && \text{(Since } j+1\text{was not active)} \\ \gamma_i &= \beta_i^{k+1} + \beta_{i+1}^{k+1} && \text{(Since } j\text{was active)} \\ \Rightarrow \lambda_{j+1}^{k+1} &= 0 > \gamma_i + \beta_i^{k+1} + \beta_{i+1}^{k+1} = 2\gamma_i > 0, \end{aligned} \tag{4.34}$$

a contradiction.

Case 2: $\lambda_j^{k+1} = \lambda_{j+1}^{k+1} = 0$.

Combining Equation (4.32) and Equation (4.33) yields

$$\lambda_j^{k+1} + \lambda_{j+1}^{k+1} = 0 > 2\gamma_i > 0, \tag{4.35}$$

again yielding a contradiction. Thus $\mathcal{A}^{k+1}$ cannot select a linearly dependent set of constraints.

Therefore, by induction Algorithm 2.2 induces a nonsingular saddle point system each iteration, and will locally converge super-linearly. □

Again, as in the case of contact, the following corollary is an immediate consequence to Theorem 4.2.

Corollary 4.2. *SCHURPA applied to the friction QP* (F2) *locally converges super-linearly.*

Corollary 4.1 and Corollary 4.2 give us the theoretical confidence to integrate SCHURPA into SP, solving the contact and friction QPs arising each iteration.

4.4 The SCHURPA-SP Algorithm

We now summarize the entire frictional contact algorithm, which employs SCHURPA to solve the QP subproblems induced by SP. Recall that SP can benefit from warm-starting by providing an initial guess of the working sets $\mathcal{A}_c^0$ and $\mathcal{A}_f^0$ close to the optimal active sets

$$\begin{aligned} \mathcal{A}_c^{t+1} &:= \{i \mid \alpha_i^{t+1} = 0\} \\ \mathcal{A}_f^{t+1} &:= \{i \mid \hat{E}_i^T \beta^{t+1} = \hat{\mu}_i \hat{\alpha_i}^{t+1}\}. \end{aligned}$$

In such a case $\mathcal{A}_c^0$ and $\mathcal{A}_f^0$ will be close to the optimal active sets *of all subproblem QPs arising in SP.* We are therefore motivated to generalize SCHURPA's initial factorization K_0 across all SP iterations so that only one factorization for contact and one factorization for friction is needed.

As discussed in Section 3.2, SCHURPA only performs one factorization of the initial saddle point system K_0, and subsequent iterations use this factorization along with the Schur-complement method to solve subsequent saddle point systems. As mentioned in Section 4.2, the Hessians and constraint matrices of the contact and friction QPs do not change within the SP Algorithm. Therefore, we only need to call SCHURPA-INIT during the first SP iteration to compute the initial solutions via the functions `C0_solve` and `F0_solve` for contact and friction, respectively. Subsequent saddle point systems across all SP iterations are solved using these functions along with the Schur-complement method, as in the SCHURPA algorithm. The combined algorithms of SCHURPA and SP are given in Algorithm 4.2. Notice that each SP iteration, initial active sets must

be given to SCHURPA. These are determined by applying exactly the same prediction scheme Equation (2.12) of PDASM to the current SP iterates, i.e.

$$\mathcal{A}_c^{s+1} = \{i \mid -\alpha_i^{s+1} - \lambda_i^{s+1} < 0\} \tag{4.36}$$

$$\mathcal{A}_f^{s+1} = \{i \mid \hat{\mu}_i \hat{\alpha_i}^{s+1} - \hat{E}_i^T \beta^{s+1} - \nu_i^{s+1} < 0\}, \tag{4.37}$$

where λ and ν are the Lagrange multipliers for contact and friction QPs, respectively. SCHURPA-SP defines our solution procedure for solving frictional contact. Now that the high level description has been given, we move to the efficient implementation of the `C0_solve` and `F0_solve` functions in the next chapter.

Algorithm 4.2 SCHURPA-SP

Input: $\mathcal{A}_c^0, \mathcal{A}_f^0, tol$
`rel_error` $= \infty$
Run SCHURPA-INIT$\left(\mathcal{A}_c^0\right)$ to compute α^0, λ^0 and `C0_solve`
Run SCHURPA-INIT$\left(\mathcal{A}_f^0\right)$ to compute β^0, ν^0 and `F0_solve`
for $s = 0, 1, 2, ...$ **do**
  Solve the contact QP: $\begin{pmatrix}\alpha^{s+1}\\ \lambda^{s+1}\end{pmatrix}$ = SCHURPA$(\mathcal{A}_c^s, \alpha^s, \lambda^s,$ `C0_solve`$)$
  Solve the friction QP: $\begin{pmatrix}\beta^{s+1}\\ \nu^{s+1}\end{pmatrix}$ = SCHURPA$\left(\mathcal{A}_f^s, \beta^s, \nu^s,\right.$ `F0_solve`$\left.\right)$
  Compute *rel_error* given by Equation (4.19)
  if `rel_error` $<$ `tol` **then**
    BREAK; $\alpha^{t+1} = \alpha^{s+1}$ and $\beta^{t+1} = \beta^{s+1}$
  end if
  $\mathcal{A}_c^{s+1} = \{i \mid -\alpha_i^{s+1} - \lambda_i^{s+1} < 0\}$
  $\mathcal{A}_f^{s+1} = \{i \mid \hat{\mu}_i \hat{\alpha_i}^{s+1} - \hat{E}_i^T \beta^{s+1} - \nu_i^{s+1} < 0\}$
end for

$f^{t+1} = T\beta^{t+1}$
$c^{t+1} = J\alpha^{t+1}$
$v^{t+1} = M^{-1}(f^* + f^{t+1} + c^{t+1})$
Output: v^{t+1}

Chapter 5

Solving the Contact and Friction QPs

In this chapter we describe the implementation details of Algorithm 4.2, our solution method for solving frictional contact arising in the simulator described in Chapter 4. Crucial to the efficiency of SCHURPA is the method of solving systems with the initial saddle point matrix, which in the case of staggered projections corresponds to the implementation of the functions that solve the initial contact and friction saddle point systems. We will show that, due to the structure of the contact and friction QPs, both functions require a factorization of a banded SPD matrix, and application of said functions amount to solving with these factors.

Section 5.1 describes the initial saddle point systems for the contact and friction quadratic programs; these may be formulated as SPD systems. We then show in Section 5.2 the afro-mentioned SPD matrices exhibit a banded structure due to the Hessian matrices of the QPs. In particular, we utilize the nullspace method to solve the friction QP which results in a banded reduced Hessian. Since SCHURPA requires multiple simultaneous solves, we are motivated to design an efficient specialized parallel solver for dense SPD banded systems with multiple right hand sides. A parallel solver based on block substitution is outlined in Section 5.3.

5.1 Saddle Point Systems Arising in Contact and Friction

The SCHURPA-SP loop given by Algorithm 4.2 requires the solution of the initial saddle point systems for the contact and friction QPs, which are performed in the functions `C0_solve` and `F0_solve`, respectively. In both cases we can rephrase these systems as the solution to an SPD system. Let us first investigate the case for contact.

5.1.1 Contact

Consider the contact QP solved at each iteration of SCHURPA-SP, which we restate here for convenience:

$$\begin{aligned} \alpha^{s+1} = \underset{\alpha}{\text{argmin}} \quad & \frac{1}{2}\alpha^T H^{\mathcal{C}} \alpha - \alpha^T c \\ \text{subject to} \quad & \alpha \geq 0, \end{aligned} \tag{C}$$

where

$$\begin{aligned} c &= -J^T M^{-1}(\hat{T}\beta^s + f^*) \\ H^{\mathcal{C}} &= J^T M^{-1} J. \end{aligned} \tag{5.1}$$

Recall $\mathcal{A}^0$ defines the initial working set and subsequent initial saddle point system. We define

$$\begin{aligned} F &= \mathcal{I} \backslash \mathcal{A}^0 \\ B &= \mathcal{A}^0, \end{aligned}$$

to be the initial set of free and bound indices of α, respectively, so that the initial saddle point system for (C) is

$$\begin{bmatrix} H^{\mathcal{C}}_{F,F} & H^{\mathcal{C}}_{F,B} & 0 \\ H^{\mathcal{C}}_{B,F} & H^{\mathcal{C}}_{B,B} & -I_B \\ 0 & -I_B & 0 \end{bmatrix} \begin{bmatrix} \alpha_F \\ \alpha_B \\ \lambda \end{bmatrix} = \begin{bmatrix} c_F \\ c_B \\ 0 \end{bmatrix}. \tag{5.2}$$

SCHURPA requires the solution of Equation (5.2) with arbitrary right hand sides, i.e.

$$\begin{bmatrix} H^{\mathcal{C}}_{F,F} & H^{\mathcal{C}}_{F,B} & 0 \\ H^{\mathcal{C}}_{B,F} & H^{\mathcal{C}}_{B,B} & -I_B \\ 0 & -I_B & 0 \end{bmatrix} \begin{bmatrix} \alpha_F \\ \alpha_B \\ \lambda \end{bmatrix} = \begin{bmatrix} c_F \\ c_B \\ b \end{bmatrix}. \tag{5.3}$$

Simplifying Equation (5.3) yields

$$\alpha_B = -b \tag{5.4}$$

$$H^{\mathcal{C}}_{F,F}\alpha_F = c_F - H^{\mathcal{C}}_{F,B}\alpha_B \tag{5.5}$$

$$\lambda = -c_B + H^{\mathcal{C}}_{B,F}\alpha_F + H^{\mathcal{C}}_{B,B}\alpha_B \tag{5.6}$$

Thus the initial saddle point system of the contact QP amounts to solving an SPD system of size $|F| \times |F|$, given in Equation (5.5). Algorithm 5.1 gives `C0_solve`. Note that a factorization of $H^{\mathcal{C}}_{F,F}$ must be computed to solve Equation (5.5); this is performed in SCHURPA-INIT of Algorithm 4.2.

5.1.2 Friction

Due to the simple structure of the reformulated friction constraints, a sparse nullspace can be computed explicitly; this allows us to form the reduced Hessian

Algorithm 5.1 `C0_solve`

Input: $\begin{bmatrix} c \\ b \end{bmatrix}$

Set $\alpha_B = -b$
Solve the SPD system $H^{\mathcal{C}}_{F,F}\alpha_F = c_F - H^{\mathcal{C}}_{F,B}\alpha_B$
Set $\lambda = -c_B + H^{\mathcal{C}}_{B,F}\alpha_F$

Output: $\begin{bmatrix} \alpha \\ \lambda \end{bmatrix}$

without destroying sparsity. We can then reformulate the initial saddle point system of the friction QP as an SPD system via the nullspace method (Nocedal and Wright, 1999). Recall the friction QP of SCHURPA-SP, given by

$$\begin{aligned} \beta^{s+1} = \underset{\beta}{\operatorname{argmin}} \quad & \frac{1}{2}\beta^T H^{\mathcal{F}}\beta - \beta^T d \\ \text{subject to} \quad & \hat{E}\beta \leq \gamma, \end{aligned} \tag{F2}$$

where

$$\begin{aligned} d &= -\hat{T}M^{-1}\left(c^{s+1} + f^*\right) \\ \gamma &= \operatorname{diag}(\hat{\mu})\hat{\alpha}^{s+1} \\ H^{\mathcal{F}} &= \hat{T}^T M^{-1}\hat{T}. \end{aligned} \tag{5.7}$$

Let $p = |\mathcal{A}^0|$ denote the number of initial active constraints and

$$\hat{E}_{\mathcal{A}^0} = \begin{bmatrix} \hat{E}_{1\mathcal{A}^0} & & & \\ & \hat{E}_{2\mathcal{A}^0} & & \\ & & \ddots & \\ & & & \hat{E}_{m\mathcal{A}^0} \end{bmatrix} \in \mathbb{R}^{p\times 2m}, \tag{5.8}$$

where $\hat{E}_{c\mathcal{A}^0}$ is the (possibly empty) sub-matrix of

$$\hat{E}_c = \begin{pmatrix} 1 & -1 & 1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix}^T$$

defined by $\mathcal{A}^0$. Theorem 4.2 ensures that SCHURPA selects at most two linearly independent constraints of $\hat{E}_c$ to be active. If exactly two are selected, this completely determines the unknowns β_{2c-1} and β_{2c}, and they can be removed from the problem. Therefore we may assume

$$\hat{E}_{c\mathcal{A}^0} \in \mathbb{R}^{q\times 2}, \qquad \text{where } 0 \le q \le 1. \tag{5.9}$$

If $q = 1$ we let

$$Z_c = \begin{cases} (1,-1)^T & \text{if } \hat{E}_{c\mathcal{A}^0} = \pm(1,1)^T \\ (1,1)^T & \text{if } \hat{E}_{c\mathcal{A}^0} = \pm(1,-1)^T \end{cases}. \tag{5.10}$$

Notice that $\hat{E}_{c\mathcal{A}^0} Z_c = 0$. If $q = 0$ then there is no reduction, and we let

$$Z_c = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}. \tag{5.11}$$

Proposition 5.1.1. *The matrix*

$$Z = \begin{bmatrix} Z_1 & & & \\ & Z_2 & & \\ & & \ddots & \\ & & & Z_m \end{bmatrix} \in \mathbb{R}^{2m\times(2m-p)} \tag{5.12}$$

forms a basis for the nullspace of $\hat{E}_{\mathcal{A}^0}$.

Proof. By construction Z is full rank since each Z_c is full rank. Computing the

product

$$\begin{aligned}\hat{E}_{\mathcal{A}^0} Z &= \begin{bmatrix} \hat{E}_{1\mathcal{A}^0} Z_1 & & & \\ & \hat{E}_{2\mathcal{A}^0} Z_2 & & \\ & & \ddots & \\ & & & \hat{E}_{m\mathcal{A}^0} Z_m \end{bmatrix} \\ &= \begin{bmatrix} 0 & & & \\ & 0 & & \\ & & \ddots & \\ & & & 0 \end{bmatrix},\end{aligned} \tag{5.13}$$

where we define the matrix product $\hat{E}_{c\mathcal{A}^0} Z_c$ to be the empty matrix if $\hat{E}_{c\mathcal{A}^0}$ is empty. Thus Z is a basis for the nullspace of $\hat{E}_{\mathcal{A}^0}$. □

Equipped with a nullspace of $\hat{E}_{\mathcal{A}^0}$, we can proceed to solve the saddle point system via the null-space method, which we briefly outline here. The initial saddle point system induced by $\mathcal{A}^0$ is

$$\begin{bmatrix} H^{\mathcal{F}} & \hat{E}_{\mathcal{A}^0}^T \\ \hat{E}_{\mathcal{A}^0} & \end{bmatrix} \begin{bmatrix} \beta \\ \nu \end{bmatrix} = \begin{bmatrix} d \\ \gamma_{\mathcal{A}^0} \end{bmatrix}. \tag{5.14}$$

SCHURPA requires the solution of Equation (5.14) with arbitrary right hand sides, i.e.

$$\begin{bmatrix} H^{\mathcal{F}} & \hat{E}_{\mathcal{A}^0}^T \\ \hat{E}_{\mathcal{A}^0} & \end{bmatrix} \begin{bmatrix} \beta \\ \nu \end{bmatrix} = \begin{bmatrix} d \\ b \end{bmatrix}. \tag{5.15}$$

Any solution to $\hat{E}_{\mathcal{A}^0} \beta = b$ can be written as

$$\beta = \beta_p + Z\beta_z, \tag{5.16}$$

where β_p is a particular solution. Substituting Equation (5.16) into Equa-

tion (5.15) and multiplying the first equation by Z^T yields the reduced Hessian equation

$$(Z^T H^{\mathcal{F}} Z)\beta_z = -(Z^T H^{\mathcal{F}} \beta_p + Z^T d). \tag{5.17}$$

Note that we can easily compute β_p by choosing

$$\beta_p = \frac{\hat{E}_{\mathcal{A}}^T b}{2} \tag{5.18}$$

since

$$\hat{E}_{\mathcal{A}} \beta_p = \frac{\hat{E}_{\mathcal{A}} \hat{E}_{\mathcal{A}}^T b}{2} = b. \tag{5.19}$$

Upon solving for β_p and β_z, β can be computed from Equation (5.16) and the corresponding Lagrange multipliers are

$$\nu = \frac{\hat{E}_{\mathcal{A}}^T (d + H^{\mathcal{F}} \beta)}{2}. \tag{5.20}$$

As in the contact QP, we have reduced the problem to an SPD system, given by Equation (5.17), of size $(2m - p) \times (2m - p)$. The function `F0_solve` is given in Algorithm 5.2 and, as in the case of contact, requires a factorization of $Z^T H^{\mathcal{F}} Z$, which is performed in SCHURPA-INIT of Algorithm 4.2.

Algorithm 5.2 `F0_solve`

Input: $\begin{bmatrix} d \\ b \end{bmatrix}$
Set $\beta_p = \frac{\hat{E}_{\mathcal{A}}^T b}{2}$
Solve the SPD system $(Z^T H^{\mathcal{F}} Z)\beta_z = -(Z^T H^{\mathcal{F}} \beta_p + Z^T d)$
Set $\beta = \beta_p + Z\beta_z$
Set $\nu = \frac{\hat{E}_{\mathcal{A}}^T (d + H^{\mathcal{F}} \beta)}{2}$
Output: $\begin{bmatrix} \beta \\ \nu \end{bmatrix}$

While nullspace matrices are typically dense and the reduced Hessian loses the sparsity structure of the original Hessian, this is not the case here. We show in the next section that, due to the structure of $H^{\mathcal{C}}$ and $H^{\mathcal{F}}$, the SPD matrices in Equation (5.5) and Equation (5.17) are banded. This banded structure is critical to the efficient implementation of SCHURPA-SP.

5.2 Banded SPD Systems

The computational "meat" of SCHURPA-SP is the factorization and subsequent solving with the SPD matrices

$$G^{\mathcal{C}} = H^{\mathcal{C}}_{F,F} \tag{5.21}$$

$$G^{\mathcal{F}} = Z^T H^{\mathcal{F}} Z. \tag{5.22}$$

These matrices are in fact banded. We first prove that $H^{\mathcal{C}}$ and $H^{\mathcal{F}}$ exhibit a banded structure.

Proposition 5.1. *Upon discretization using trilinear shape functions $H^{\mathcal{C}}$ is a banded matrix.*

Proof. We prove the proposition on a 2D grid using bilinear shape functions; the extension to 3D using trilinear shape functions as in the simulator is straightforward. Consider the four shape functions ϕ_N for $N \in \mathcal{N}^i := \{N_1, N_2, N_3, N_4\}$ providing local support to a given contact cell Ω_i, as pictured in Figure 4.1. The locality of the shape functions implies

$$J_{iN} \neq (0,0,0)^T \iff N \in \mathcal{N}^i.$$

We define grid cells Ω_i and Ω_j to be *adjacent iff* they share a common shape function with local support, as exemplified in Figure 5.1. Thus if Ω_i and Ω_j are

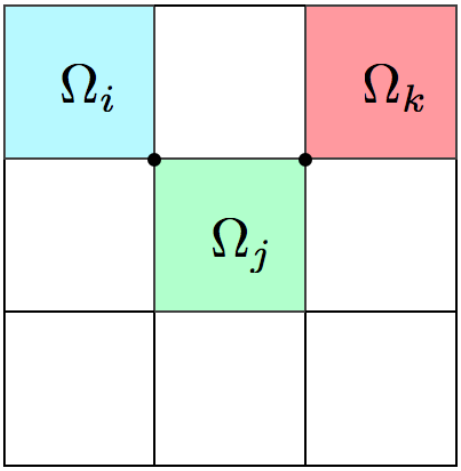


Figure 5.1: Example of cell adjacency on a 2D grid. Grid cells Ω_i and Ω_j are adjacent as they share a common shape function, while Ω_i and Ω_k are not adjacent.

not adjacent

$$\begin{aligned} J_{iN}J_{jN} &= 0 \qquad N = 1, ..., L \\ \Rightarrow H^{\mathcal{C}}_{ij} &= \sum_k \frac{J_{ik}J_{jk}}{M_{kk}} = 0. \end{aligned}$$

In other words, $H^{\mathcal{C}}$ has a bandwidth of at most ρ_{max}, where

$$\rho_{max} = \max_{i,j} \{|i-j| \quad | \quad \text{grid cells } \Omega_i \text{are } \Omega_j \text{ are adjacent}\} \tag{5.23}$$

as required. □

We denote the bandwidth of $H^{\mathcal{C}}$ as ρ. The constraint cells are ordered by the standard triplet ordering (x, y, z) (i.e. first by x, followed by y, followed by z coordinates). Assuming this order roughly groups the adjacent constraint cells together, ρ should be small relative to m. Figure 5.2 exemplifies a typical sparsity pattern of $H^{\mathcal{C}}$, exhibiting a banded structure. A bandwidth minimizing

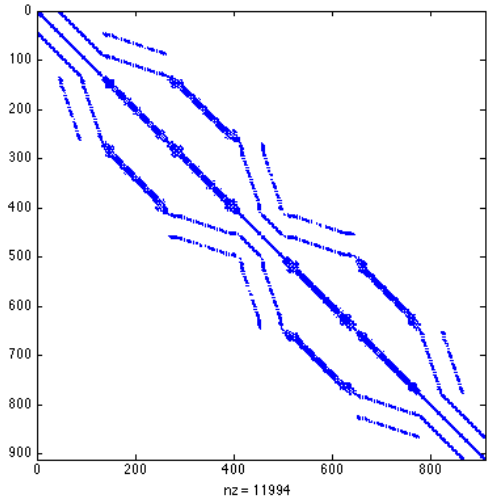


Figure 5.2: Typical banded sparsity pattern of the contact Hessian $H^{\mathcal{C}}$ arising in the simulator.

procedure such as Cuthill-McKee reordering could also be performed to decrease ρ.

Proposition 5.2. *The matrix $H^{\mathcal{F}}$ is banded with a bandwidth of at most 2ρ.*

Proof. The proof is analogous to the proof of Proposition 5.1 where quantities corresponding to J are replaced with quantities corresponding to $\hat{T}$. □

The following corollary immediately follows from Proposition 5.1.

Lemma 5.1. *The matrix $G^{\mathcal{C}}$ is banded and SPD with a bandwidth at most ρ.*

Proof. Since $G^{\mathcal{C}}$ is a sub-matrix of $H^{\mathcal{C}}$, $G^{\mathcal{C}}$ must have a bandwidth at most the bandwidth of $H^{\mathcal{C}}$, which is at most ρ. □

Finally, we show that $G^{\mathcal{F}}$ is also banded.

Lemma 5.2. *The matrix $G^{\mathcal{F}}$ is banded and SPD with a bandwidth at most $2\rho+1$.*

Proof. Recall that

$$Z = \begin{bmatrix} Z_1 & & & \\ & Z_2 & & \\ & & \ddots & \\ & & & Z_m \end{bmatrix},$$

where $Z_c \in \mathbb{R}^{2\times q}$, with $q = 1$ or $q = 2$. We decompose $H^{\mathcal{F}}$ into 2×2 blocks

$$H^{\mathcal{F}} = \begin{bmatrix} H^{\mathcal{F}}_{1,1} & \cdots & H^{\mathcal{F}}_{1,p+1} & & \\ \vdots & H^{\mathcal{F}}_{2,2} & & \ddots & \\ H^{\mathcal{F}}_{p+1,1} & & & & H^{\mathcal{F}}_{m-p,m} \\ & \ddots & & \ddots & \vdots \\ & & H^{\mathcal{F}}_{m,m-p} & \cdots & H^{\mathcal{F}}_{m,m} \end{bmatrix}. \tag{5.24}$$

Equation (5.24) ensures the 2ρ bandwidth of $H^{\mathcal{F}}$ is captured within the block form. Now we simply compute the product

$$\begin{aligned} G^{\mathcal{F}} &= Z^T H^{\mathcal{F}} Z \\ &= \begin{bmatrix} Z_1 & & & \\ & Z_2 & & \\ & & \ddots & \\ & & & Z_m \end{bmatrix}^T \begin{bmatrix} H^{\mathcal{F}}_{1,1} & \cdots & H^{\mathcal{F}}_{1,p+1} & & \\ \vdots & H^{\mathcal{F}}_{2,2} & & \ddots & \\ H^{\mathcal{F}}_{p+1,1} & & & & H^{\mathcal{F}}_{m-p,m} \\ & \ddots & & \ddots & \vdots \\ & & H^{\mathcal{F}}_{m,m-p} & \cdots & H^{\mathcal{F}}_{m,m} \end{bmatrix} \begin{bmatrix} Z_1 & & & \\ & Z_2 & & \\ & & \ddots & \\ & & & Z_m \end{bmatrix} \\ &= \begin{bmatrix} Z_1^T H^{\mathcal{F}}_{1,1} Z_1 & \cdots & Z_1 H^{\mathcal{F}}_{1,p+1} Z_{p+1} & & \\ \vdots & Z_2^T H^{\mathcal{F}}_{2,2} Z_2 & & \ddots & \\ Z_{p+1}^T H^{\mathcal{F}}_{p+1,1} Z_1 & & & & Z_{m-p}^T H^{\mathcal{F}}_{m-p,m} Z_m \\ & \ddots & & \ddots & \vdots \\ & & Z_m^T H^{\mathcal{F}}_{m,m-p} Z_{m-p} & \cdots & Z_m^T H^{\mathcal{F}}_{m,m} Z_m \end{bmatrix}. \end{aligned} \tag{5.25}$$

From Equation (5.25) we see that $G^{\mathcal{F}}$ has a bandwidth of at most $2(\rho+1)-1 = 2\rho+1$, as required. □

We emphasize that while typically the reduced Hessian loses sparsity, this is not the case here due to the particular structure of the nullspace matrix Z. The final section of this chapter describes a parallel solver for the banded SPD systems arising in the contact and friction QPs.

5.3 Block Solver for Banded SPD Systems

Both the contact and friction QP problems require the solution to a banded SPD system. In the case of contact we have

$$G^{\mathcal{C}}\alpha_F = c_F, \tag{5.26}$$

and for friction we have

$$G^{\mathcal{F}}\beta = -(Z^T H^{\mathcal{F}}\beta_p + Z^T d). \tag{5.27}$$

Recall that SCHURPA requires solving systems with multiple right-hand-sides within the functions `C0_solve` and `F0_solve`. Within these functions we must solve Equation (5.26) and Equation (5.27), respectively, implying that c_F, β_p, and d may be matrices. The dimensionality of these matrices depend on the number of changes to the working set. Thus we are interested in developing a parallel solver to systems of the form

$$HX = B, \tag{5.28}$$

where $H \in \mathbb{R}^{n\times n}$ is a banded SPD matrix with bandwidth $\rho \ll n$, and $X, B \in \mathbb{R}^{n\times k}$, where $k \ll n$ is the number of right hand sides. We write the Cholesky

factorization

$$LL^T = H.$$

Note that L is also banded. A solution to Equation (5.28) can now be computed via

$$LY = B,$$

$$L^T X = Y.$$

Unfortunately the backward/forward substitution method is inherently sequential. Recently, approaches to parallelize sparse triangular solves have been investigated (Naumov, 2011) wherein one reorders the matrix by grouping equations which may be solved independently. However, the results only offer a modest speedup (e.g. 2× on average) and don't apply to the dense banded case being discussed here. For large, very narrow banded systems, the SPIKE algorithm Polizzi and Sameh (2006) can work extremely well. However, our bands are not fixed and may be small to medium sized, depending on the simulation data (see Figure 5.2). We opt for a simple yet effective parallel approach based on block substitution.

We decompose the Cholesky factors into blocks of size s as follows:

$$L = \begin{bmatrix} L_{11} & & & & \\ L_{21} & L_{22} & & & \\ & \ddots & \ddots & & \\ & & & L_{p,p-1} & L_{pp} \end{bmatrix}, \tag{5.29}$$

where $L_{ij} \in \mathbb{R}^{s \times s}$ and $p = \lceil \frac{n}{s} \rceil$. Notice that $s \geq \rho$ to ensure the two block-banded structure Equation (5.29) contains all nonzero elements of L. Extracting

the block diagonals into the matrices

$$C = \begin{bmatrix} L_{21} \\ \vdots \\ L_{p,p-1} \end{bmatrix} \in \mathbb{R}^{(p-1)s \times s} \qquad D = \begin{bmatrix} L_{11} \\ \vdots \\ L_{pp} \end{bmatrix} \in \mathbb{R}^{ps \times s},$$

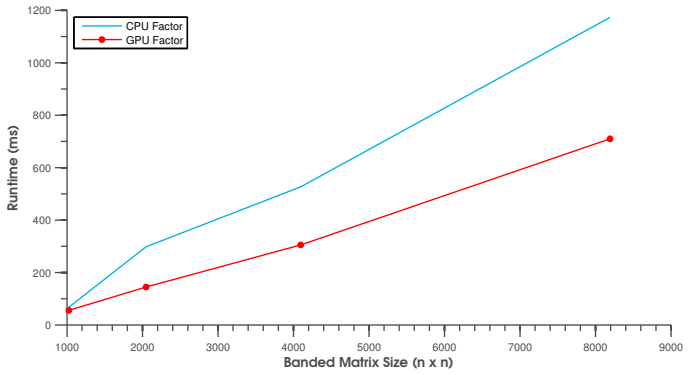
we solve Equation (5.29) via Algorithm 5.3, which is simply a block version of substitution. We implement the algorithm on the GPU using CUDA, described in the next chapter.

Algorithm 5.3 Block Substitution

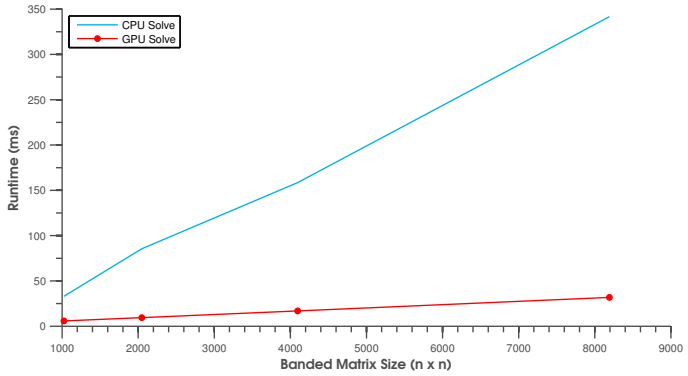
Input: Block matrices C, D from Cholesky factorization $LL^T = H$
$Y_1 = D_1^{-1} B_1$
for $i = 2, ..., p$ **do**
  $Y_i = D_i^{-1}(B_i - C_{i-1} Y_{i-1})$
end for
$X_p = D_p^{-1}$
for $i = p-1, ..., 1$ **do**
  $X_i = D_i^{-1}(Y_i - C_i X_{i+1})$
end for
Output: Solution to $HX = B$

Upon computing D_i^{-1} for $i = 1, ..., p$ in parallel as a pre-processing step, Algorithm 5.3 requires $4p - 2$ sequential matrix-matrix multiplications which can be performed efficiently in parallel. By choosing appropriately sized blocks, the solving phase of SCHURPA can be significantly improved. The only additional cost is the pre-processing step of computing the inverse diagonal blocks D_i^{-1}. This cost is small relative to the Cholesky factorization and need only be performed once during the initialization step of SCHURPA, after the initial factorization phase. Explicit inversion of the diagonal blocks was also done in (Tomov et al., 2010) to perform triangular solvers on the GPU, and performance gains exceeding 50× that of NVIDIA'S CUBLAS triangular solvers were observed. Figure 5.3 shows runtimes for the factorization and solve time of the

block banded solver implemented on the GPU compared against a sequential substitution implemented on the CPU for a fixed bandwidth of $\rho = 1000$ solving for $k = 10$ right-hand-sides as a function of the matrix dimensions. We used MATLAB to perform the banded Cholesky factorization on the CPU. For the GPU version we utilized the CULA library (Humphrey et al., 2010) for CUDA. For matrices of size $n \approx 8000$ we observe the GPU performs the factorization twice as fast as the CPU. Solving a banded system sequentially runs in $O\left(n\rho^2\right)$, whereas our block banded solver (in an ideally parallelized setting) runs in $O\left(\rho s\right) = O\left(n\right)$. Thus, while both the CPU and GPU solving runtimes are linear in n, the GPU solver is independent of the bandwidth size. Figure 5.3 demonstrates the scalability of the GPU block substitution solver. The gains in the solving phase come at the cost of the pre-processing step during factorization which, as mentioned, is small relative to the cost of factorization.



(a)



(b)

Figure 5.3: Comparison of (a) the banded Cholesky factorization and (b) the block banded solver CPU and GPU runtimes, applied to banded matrices as a function of matrix size. The bandwidth $\rho = 1000$ and number of right-hand-sides $k = 10$ are fixed across matrix dimensions.

Chapter 6

GPU Implementation

General purpose computing on Graphics Processing Units (GPUs), or GPGPU, is a relatively recent endeavor taken up by the scientific computing community. Before common API frameworks existed, customizing GPU functionality was arduous and left primarily to specialists. However, two important developments have made GPGPU popularity explode:

- the advent of programming interfaces such as OpenCL (Stone et al., 2010) and NVIDIA's CUDA (NVIDIA, 2012b), and
- double precision capability on GPUs.

No longer limited to computer graphics, GPUs are fast becoming a staple of scientific computing.

This chapter outlines a parallel implementation of SCHURPA applied to the simulator described in Chapter 4. We implement SCHURPA in the GPGPU framework using CUDA. Our motivation for a GPU implementation is two-fold: the simulator is also GPU-based, so CPU to GPU memory overhead is avoided, and SCHURPA's amenability to parallelism, as discussed in Chapter 3. Dense linear algebra algorithms run on GPUs can lead to orders of magnitude acceleration compared to standard CPU implementations(Tomov et al., 2010), and this performance gap will only widen as the many-core paradigm continues to pervade contemporary programming.

Section 6.1 overviews CUDA, describing the C++ extension syntax. In Section 6.2, we give a taste of the custom functionality implemented on the GPU required by SCHURPA by detailing a symmetric-banded general matrix-matrix multiplication operation. Off-the-shelf GPU libraries were also extensively used in SCHURPA, which we outline in Section 6.3.

6.1 CUDA Programming on the GPU

CUDA (Compute Unified Device Architecture), introduced by NVIDIA in 2006, refers to a massively parallel programming model and architecture, along with an API to program GPUs using high level programming languages such as C++ and Fortran. It is freely available (NVIDIA, 2012b) and requires a CUDA-capable NVIDIA GPU (GeForce 8 or greater, Tesla, Quadro etc.). CUDA uses a SIMD (Single Instruction Multiple Data) execution paradigm to run many concurrent threads (individual processing units) from a single function call of a kernel: functions programmed in CUDA which are compiled and run on a GPU. The many cores on a GPU chip allow it to perform thousands of tasks on large sets of data in parallel, whereas CPU programs perform sequentially (multi-core CPUs and hyper-threading attempt to improve this limitation). Due to the increased number of cores on GPUs, theoretical floating point operation throughput (GFLOP/s) is orders of magnitude greater than that of the CPU. Table 6.1 on page 74 summarizes the differences between CPU and GPU programming paradigms. SCHURPA was written in the extended C++ CUDA API.

GPU memory, called *device* memory, is managed by the GPU; CPU, or *host* memory, can be copied to device memory and vice versa via the CUDA API. Host-to-device and device-to-host memory copies are slow and thus aimed to be minimized. As the simulator computes all data on the GPU, and device-to-

	CPU	GPU
Number of Cores	Several	Hundreds
Number of Threads	Several	Thousands
Thread Speed	Fast	Slow
Cache size	Large	Small

Table 6.1: CPU vs. GPU comparison.

device memory transfer is fast, we need not be concerned with any host memory transfer overhead.

There are several types of GPU memory, including: global, shared, registers and local. Global memory is the largest, with the slowest read/write access. Pointers passed to kernel functions are typically arrays stored in global memory. Shared memory is much smaller and shared among threads in a single processing block. The access speed is much faster than global memory, and can be thought of as the cache. Registers store local data for each individual thread and has the fastest access. Local memory is again restricted to a thread and stores any data that can't fit into the register memory. There is also texture and constant memory, which we shan't discuss here.

6.1.1 CUDA Programming Model

Kernel functions are configured to execute threads partitioned into computational blocks of one, two or three-dimensions which exist on a two-dimensional grid (see Figure 6.1). The number of threads per block is limited as they share resources residing on the core executing the block. The blocks cannot communicate or synchronize with one another; this allows them to be run independently in any order. The computational grid is conceptually useful for decomposing a problem domain such as a fluid simulation, or two-dimensional data such as a matrix. Kernels require two additional parameters placed within "<<<>>>"

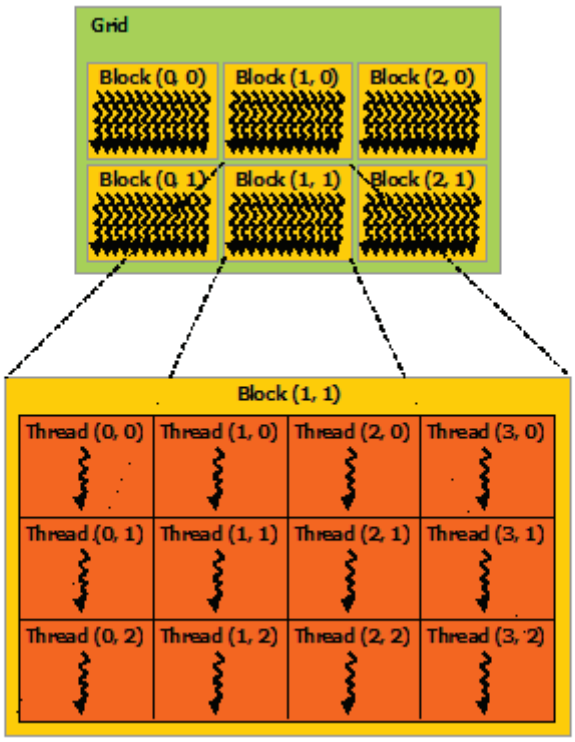


Figure 6.1: Computational grid of threads, partitioned into blocks, executed in parallel on the GPU. From CUDA Programming Guide (NVIDIA, 2012b).

brackets prefacing the regular parameter list: one `int2` specifying the number of blocks on the two-dimensional grid, and an `int3` specifying the number of threads per block. For example, suppose we want to add two matrices $A \in \mathbb{R}^{n \times n}$ and $B \in \mathbb{R}^{n \times n}$. Each thread will compute one element of the resulting matrix $C \in \mathbb{R}^{n \times n}$. We decompose the matrices into 16×16 blocks mapped onto the grid and run the kernel `addMatrices` via the following syntax:

```
// assume N x N matrices A, B, C have been defined
int2 threadsPerBlock(16, 16);
int3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y, 1);

addMatrices<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

Listing 6.1: Calling a CUDA kernel.

In Listing 6.1 the number of blocks is computed to ensure there are n^2 threads. Listing 6.2 shows the function body of the addMatrices kernel. The "___global___" identifier is used to specify a kernel function, which requires a void return type.

```
__global__ void addMatrices(float[N][N] A, float[N][N] B, float[N][N] C)
{

  int i = blockIdx.y * blockDim.y + threadIdx.y; // row index
  int j = blockIdx.x * blockDim.x + threadIdx.x; // column index

  if (i < N && j < N)
    C[i][j] = A[i][j] + B[i][j];
}
```

Listing 6.2: Body of a CUDA kernel.

CUDA provides internal variables to determine the unique thread index and block index, given by `threadIdx` and `blockIdx`, respectively. Notice that in Listing 6.2 these indices are used to compute the global row and column indices. Each thread executes the kernel in parallel on the GPU, writing the solution to the two-dimensional array C. The arrays are all stored in global memory on the GPU.

6.2 Custom Kernels for SCHURPA

Several operations not available in the current versions of the CUDA libraries (see Section 6.3) were required for the implementation of SCHURPA. Here we simply describe one kernel that performs matrix-matrix multiplication with a symmetric banded and dense matrix, respectively, to get a flavor for the kernel programming process in CUDA.

We denote the following operation as `symbgmm`, for symmetric-banded general matrix-matrix multiplication:

$$C = BA, \tag{6.1}$$

where $B \in \mathbb{R}^{n\times n}$ is a symmetric banded matrix, and $A \in \mathbb{R}^{m\times n}, C \in \mathbb{R}^{m\times n}$ are dense matrices. We define B to have at most ρ nonzero sub/super-diagonals, i.e.

$$B_{ik} = 0 \quad \text{for } \min(1, i-\rho) \leq \ k \leq \ \max(1, i+\rho)$$

so that Equation (6.1) can be rewritten as

$$\begin{aligned} C_{ij} &= \sum_{k=1}^{n} B_{ik} A_{kj} \\ &= \sum_{k=\min(1,i-\rho)}^{\max(n,i+\rho)} B_{ik} A_{kj.} \end{aligned}$$

Such an operation is required due to the banded structure of the contact and friction Hessians described in Chapter 4, and currently the CUBLAS library only supports general dense matrix-matrix multiplication. A sequential version of `symbgmm` written in C++ is given in 6.3. The bodies of the helper functions are omitted for brevity (see Appendix B).

A straightforward extension to CUDA is given in Listing 6.4. As in the matrix addition example Listing 6.2, the two outer loops indexing rows i and column j are replaced with a unique thread for each element C_{ij}. However, due to slow global memory access this kernel is not ideal. For example, a row b_i^T is required by multiple threads to compute the dot product $b_i^T a_j$, yet need not be read from global memory by each thread. Rewriting Equation (6.1) in block form yields

$$C_{IJ} = \sum_{K} B_{IK} A_{KJ}. \tag{6.2}$$

We map the computation of the $s \times s$ block C_{IJ} to a block on the computational

```
struct Matrix{
  int width;
  int height;
  int ld; // leading dimension
  float* elements;
};
struct BandedMatrix : Matrix{
  int rho; // # subdiagonals
};

// helper functions
float getElement(const Matrix A, int i, int j);
float getElement(const BandedMatrix B, int i, int j);
void setElement(Matrix A, int i, int j, float value);

// computes C = B * A, where B is a symmetric banded matrix
void symbgmmHost(BandedMatrix B, Matrix A, Matrix C){

  float Cval;

  // loop through each element of C
  for (int i = 0; i < C.height; i++){
    for (int j = 0; j < C.width; j++){

      Cval = 0.0;

      // compute the dot product b_i^T a_j
      for (int k = max(0, i - B.rho);
        k < min(B.width - 1, i + B.rho); k++)
        Cval += getElement(B, i, k) * getElement(A, k, j);

      setElement(C, i, j, Cval);
    }
  }
}
```

Listing 6.3: Sequential function of the symbgmm operation.

grid. To avoid redundant memory fetching, sub-matrices B_{IK} and A_{KJ} can be loaded into shared memory for each block, with each thread loading a sub-matrix element in parallel. A thread computes an element of the sub-matrix product $C_S = B_{IK}A_{KJ}$ and stores the result in register memory. Summing all such products over K yields the desired block C_{IJ}, which is then written to device memory. Due to the banded structure of B we have

$$K = (k_0 : k_0 + s), ..., (k_f : k_f + s),$$

where $k_0 = \max(1, I_1 s - \rho)$, $k_f = \min(n - s, I_1 s + \rho)$. A kernel using this shared memory approach is given in Listing 6.5. The `___shared___` prefix of the sub-matrix arrays defines them to be stored in shared memory. The `___syncthreads`() function synchronizes all threads of a given block, so that all previous code has been executed. The observant reader will notice that reading sub-matrices in memory may fail when for example requesting an element below the band of B, or when the block size s does not divide n or m. In such a case the sub-matrix element is set to 0, which produces the desired result. These details are abstracted away in the helper bodies, given in Chapter B.

Figure 6.2 compares the runtimes of the different `symbgmm` routines, along with a comparison to NVIDIA's CUBLAS matrix multiplication for general matrices. The shared implementation significantly outperforms the naive implementation.

Along with the symgbmm operation, several other custom kernels were written to perform the following operations:

- forming the contact and friction Hessians $H^{\mathcal{C}}$ and $H^{\mathcal{F}}$, respectively,
- forming the free contact Hessian $G^{\mathcal{C}}$, and

```
// computes C = B * A, where B is a symmetric banded matrix
__global__ void symbgmmNaive(BandedMatrix B, Matrix A, Matrix C){

  int i = blockDim.y * blockIdx.y + threadIdx.y; // row index
  int j = blockDim.x * blockIdx.x + threadIdx.x; // column index

  if (i >= C.height || j >= C.width)
    return;

  float Cval = 0.0;

  // compute the dot product b_i^T a_j
  for (int k = max(0, i - B.rho);
  k < min(B.width - 1, i + B.rho); k++)
    Cval += getElement(B, i, k) * getElement(A, k, j);

  setElement(C, i, j, Cval);
}
```

Listing 6.4: Naive kernel of the symbgmm operation.

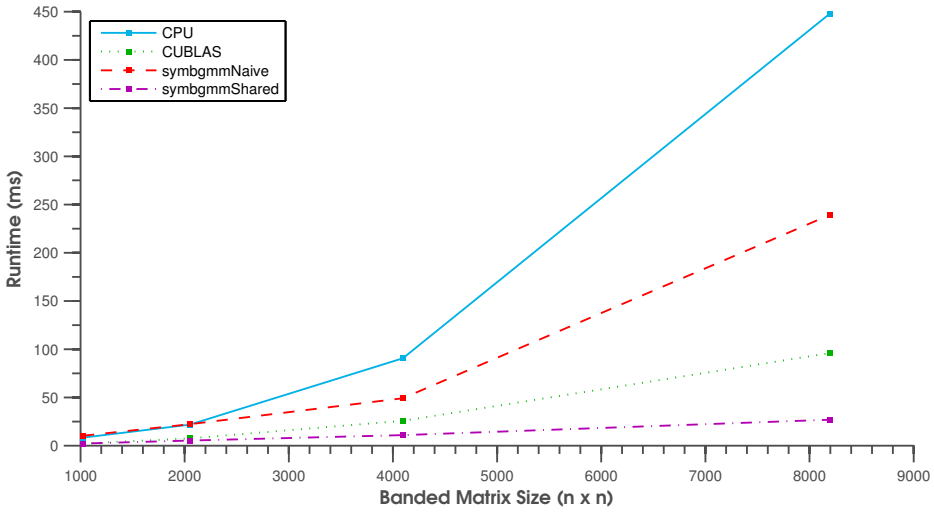


Figure 6.2: Runtimes of symbgmm, the symmetric-banded general matrix-matrix multiplication operation, as a function of the matrix size. For comparison, we also show the performance of CUBLAS (NVIDIA, 2012a) for general matrix-matrix multiplication.

```
#define BLOCK_SIZE 16 // block size loaded into shared memory

// computes C = B * A, where B is a symmetric banded matrix
__global__ void symbgmmShared(BandedMatrix B, Matrix A, Matrix C){

  int blockRow = blockIdx.y;
  int blockCol = blockIdx.x;

  int row = threadIdx.y;
  int col = threadIdx.x;

  int i = blockRow * BLOCK_SIZE + row; // row index
  int j = blockCol * BLOCK_SIZE + col; // column index

  if (i >= C.height || j >= C.width)
    return;

  float Cval = 0.0;

  // loop through each nonzero block of row blockRow
  int startBlock = max(0, blockRow * BLOCK_SIZE - B.rho);
  int endBlock = min(B.width-1, blockRow * BLOCK_SIZE + B.rho + BLOCK_SIZE - 1);

  for (int m = startBlock; m <= endBlock; m += BLOCK_SIZE){

    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

    // each thread loads an element of the submatrices
    Bs[row][col] = getElement(B, i, m + col);
    As[row][col] = getElement(A, m + row, j);

    __syncthreads(); // ensure submatrices are loaded

    // compute the submatrix product B_s A_s
    for (int k = 0; k < BLOCK_SIZE; k++)
      Cval += Bs[row][k] * As[k][col];

    __syncthreads(); // ensure dot product is computed
  }

  setElement(C, i, j, Cval);
}
```

Listing 6.5: Kernel of the symbgmm operation using shared memory.

- forming the reduced friction Hessian $G^{\mathcal{F}}$.

The details of these functions are left out of this thesis, but they all follow similar design principles to Listing 6.5.

6.3 CUDA Libraries

Several publicly available libraries were crucial in the efficacy of the SCHURPA GPU implementation. These include :

- CUBLAS (NVIDIA, 2012a) - a GPU implementation of the BLAS API, developed by NVIDIA,
- CUSPARSE (NVIDIA, 2012c) - a GPU library for manipulating sparse data and performing sparse matrix operations, developed by NVIDIA,
- CULA (Humphrey et al., 2010) - a GPU implementation of the LAPACK API, developed by EM Photonics, and
- Thrust (NVIDIA, 2012d) - a high-level templated interface analogous to the STL library of C++, developed by Jared Hoberock and Nathan bell.

Thrust was used for device memory management as well as vector operations such as reduction. CUSPARSE handled the sparse matrix data and operations including the constraint Jacobian and tangent matrices in the simulator. CUBLAS provided the low level functionality such as device-to-device memory copying, dot products, matrix vector products etc. in the SCHURPA algorithm. The highly optimized banded cholesky solve in CULA was used for the factorization of the matrices $G^{\mathcal{C}}$ and $G^{\mathcal{F}}$.

Chapter 7

Results

We demonstrate the effectiveness of our SCHURPA implementation in the context of generic QPs and the friction and contact QP subproblems arising in the staggered projections algorithm. We compare our method with a direct approach that computes a new factorization in each iteration of PDASM, which we call DIRECT.

In Section 7.1 we randomly generate and solve QPs of fixed dimension. Runtimes are compared against CPU and GPU-based implementations of DIRECT and SCHURPA. We show that as the initial working set $\mathcal{A}^0$ approaches the optimal active set $\mathcal{A}^*$, SCHURPA significantly outperforms the direct version. We then leverage the staggered projections algorithm with SCHURPA to resolve frictional contact in the Eulerian solids simulator discussed in Chapter 4 . The results are given in Section 7.2.

All experiments were performed on an Intel i7 2.80GHz CPU running a 64-bit Windows 7 OS, with an NVIDIA GeForce GTX 580 GPU. A MATLAB interface was created using mex (MATLAB executable) files to allow fast prototyping of the experiments as well as visualization capabilities.

7.1 Randomly Generated QPs

To investigate the warm-starting capabilities of SCHURPA, we randomly generated QPs of the form

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2}x^T Hx - c^T x \\ \text{subject to} \quad & x \geq 0, \end{aligned} \tag{7.1}$$

where $x, c \in \mathbb{R}^n$ and $H \in \mathbb{R}^{n \times n}$ is a banded SPD matrix with bandwidth ρ. In our experiments, $n = 8000$ and $\rho = 1000$ were held fixed. The solution to a randomly generated QP was first computed to determine $\mathcal{A}^*$, and $\mathcal{A}^0$ was constructed via altering $\mathcal{A}^*$. Specifically, we reverse the inclusion of the first δ constraints to obtain $\mathcal{A}^0$, so that $|\mathcal{A}^0 - \mathcal{A}^*| = \delta$, where $\mathcal{A}^0 - \mathcal{A}^*$ represents the set difference. CPU and GPU implementations of SCHURPA and DIRECT were then run on the QPs using $\mathcal{A}^0$. The CPU implementations were done in MATLAB. The saddle point systems of (7.1) were solved using the reduced Hessian method, as discussed in Section 5.1.1. Figure 7.1 shows the runtimes of the algorithms as a function of the working set perturbation, i.e., $|\mathcal{A}^0 - \mathcal{A}^*|$. We observe that the GPU versions significantly outperform their CPU counterparts, with the GPU version of SCHURPA being the most efficient. Notice that as the perturbation increases, the runtimes of both versions of SCHURPA increase, but the GPU version significantly less so than the CPU version; this occurs due to the parallel block solver discussed in Section 5.3. Figure 7.2 shows a breakdown of the GPU-based SCHURPA runtimes. We observe that the factorization runtime is constant as only one factorization occurs, independent of iteration number. The solve runtime increases as the perturbation increases due to a larger Schur-complement solve. Table 7.1 details the statistics of the GPU-based DIRECT and SCHURPA algorithms. We denote factorization runtimes by `f`(ms) and factorization count by #`f`, solving the saddle point system runtimes by `s`(ms) and the number of solves by #`s`, and total runtimes by `t`. The number of iterations for both methods are identical as they are both PDASMs. Notice that one factorization of SCHURPA is more expensive than one factor-

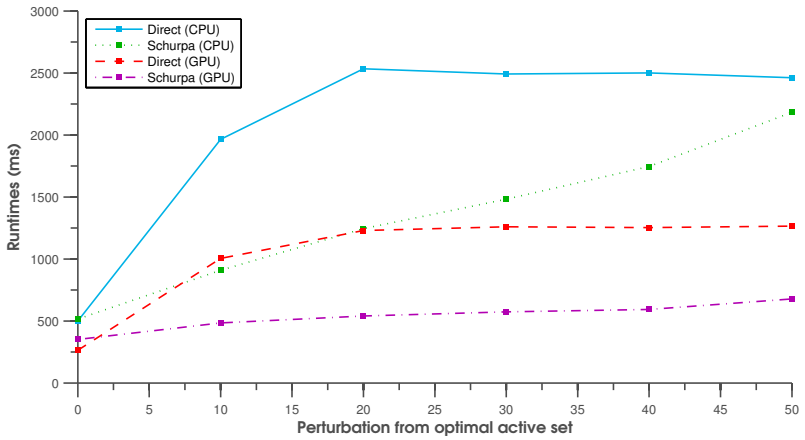


Figure 7.1: Runtimes of the CPU and GPU implementations of the DIRECT and SCHURPA algorithms, applied to (7.1), as a function of $|\mathcal{A}^0 - \mathcal{A}^*|$. For these experiments the matrix size $n = 8000$ and bandwidth $\rho = 1000$ were held fixed.

ization of DIRECT due to the pre-processing step of the block banded solver as explained in Section 5.3. Finally, we observe that the speedup SCHURPA attains over DIRECT is proportional to the number of iterations required by PDASM. For the case of $|\mathcal{A}^0 - \mathcal{A}^*| \leq 50$ a speedup of over $2\times$ is observed.

These results show that SCHURPA can successfully take advantage of warm-starting, and our GPU implementation scales well as a function of $|\mathcal{A}^0 - \mathcal{A}^*|$. Compared to DIRECT, SCHURPA is significantly more efficient, especially when the number of PDASM iterations is large.

7.2 Frictional Contact in the Simulator

We now show the results of the SCHURPA-SP method given by Algorithm 4.2 applied to a simulation involving frictional contact in the simulator. The simulation involves a collision of 3×3 grid of cylinders, as depicted in Figure 7.3. Each cylinder has a length of 5m and diameter 3m, and moves toward the center with a speed of 4m/s. The frictional coefficient $\mu = 1$ was used throughout the

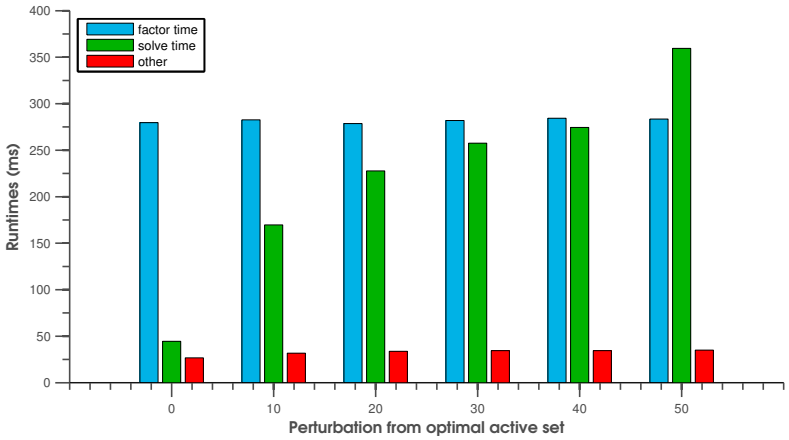
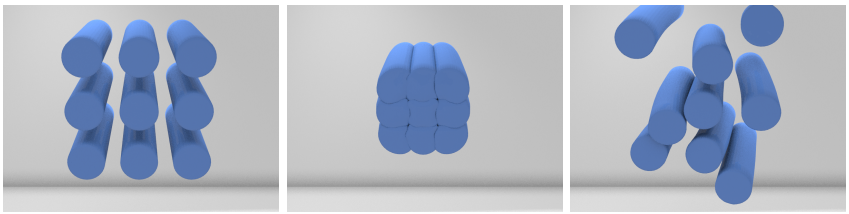


Figure 7.2: Decomposition of the GPU-based SCHURPA runtimes, applied to (7.1), as a function of $|\mathcal{A}^0 - \mathcal{A}^*|$. For these experiments the matrix size $n = 8000$ and bandwidth $\rho = 1000$ were held fixed.



(a) (b) (c)

Figure 7.3: Cylinder collision simulation. (a) The initial setup of the cylinders. (b) Contact begins, denoting $t = t_0 = 0$ seconds. (c) Resolution of the collision at $t = T = 0.5$ seconds.

	SCHURPA				DIRECT				# Iterations
$\lvert\mathcal{A}^0 - \mathcal{A}^*\rvert$	`f` (ms)	# `f`	`s` (ms)	# `s`	`t` (ms)	`f` (ms)	# `f`	`s` (ms)	# `s`	`t` (ms)	
0	279.574	1	44.5667	1	350.912	193.128	1	45.9466	1	263.926	1
10	282.66	1	169.643	15	484.104	767.985	4	183.745	4	1004.63	4
20	278.669	1	227.727	31	540.222	938.094	5	229.483	5	1229.05	5
30	281.972	1	257.621	42	574.07	967.221	5	229.828	5	1260.15	5
40	284.377	1	274.522	54	593.343	959.715	5	230.715	5	1253.28	5
50	283.504	1	359.458	72	678.072	972.877	5	229.687	5	1265.25	5

Table 7.1: Statistics of the GPU-based implementations of SCHURPA and DIRECT applied to (7.1). We denote factorization runtimes by `f`(ms) and factorization count by #`f`, solving the saddle point system runtimes by `s`(ms) and the number of solves by #`s`, and total runtimes by `t`.

simulation. We used a constant time step of $\Delta t = 0.005$ seconds, for a total time of $T = 0.5$ seconds. We compared the performance of DIRECT and SCHURPA by solving the contact and friction QP subproblems in SP using GPU-based implementations of both methods.

Recall that Algorithm 4.2 requires the initial working sets $\mathcal{A}_c^0$ and $\mathcal{A}_f^0$ for contact and friction, respectively, and the quality of these estimates is crucial for effective warm-starting. For our warm-starting strategy we chose a constraint c to be active if its corresponding grid cell Ω_c had a constraint that was active in the previous time step's optimal active set. As contact is initiated the warm-starting strategy fails as there are no constraints during the previous time step. More sophisticated warm-starting techniques which predict contact could improve this limitation. Figure 7.4 shows the proportions $\frac{\mathcal{A}_c^0}{\mathcal{A}_c^*}$ and $\frac{\mathcal{A}_f^0}{\mathcal{A}_f^*}$ over the course of the simulation. Once contact begins the ratios stay above 0.8 throughout the simulation, yielding very good initial estimates. As a result, SP required at most 2 iterations and SCHURPA required at most 8 iterations for contact and 11 iterations for friction throughout the simulation, demonstrating the utility of warm-starting both SP and SCHURPA. The warm-starting also reduces the number of solves required for Algorithm 4.2, which are shown for the contact and friction phases in Figure 7.6. Observe that near $t = 0.35$ seconds, $\frac{\mathcal{A}_c^0}{\mathcal{A}_c^*}$ and $\frac{\mathcal{A}_f^0}{\mathcal{A}_f^*}$ increase which improves the warm-starting and as a result the number of solves decreases. If the number of solves required by SCHURPA exceeds a threshold `max_solve`, we reset the algorithm as described in Section 3.3. In our implementation we set `max_solve` = `500`. Over the course of the simulation resetting was required at eight time steps in the friction solving phase. These were near the beginning of the simulation when the warm-starting algorithm failed to adequately predict $\mathcal{A}_f^*$. In the contact solving phase restarting never occurred.

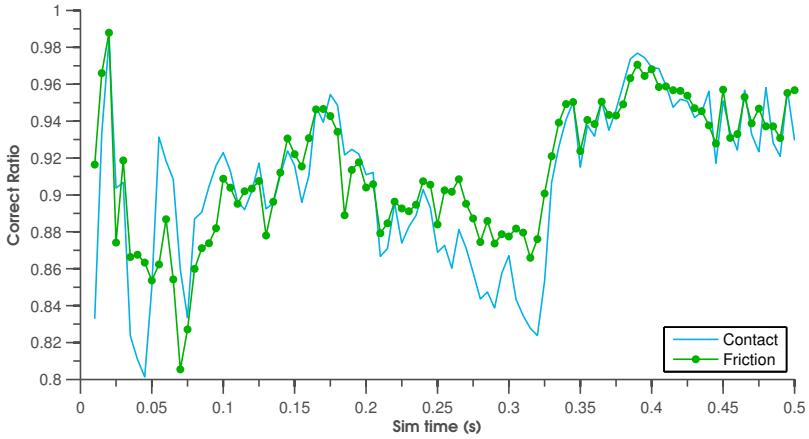


Figure 7.4: The proportion of correct indices in $\mathcal{A}_c^0$ and $\mathcal{A}_f^0$ over the course of the simulation.

Let $\rho^{\mathcal{C}}$ denote the bandwidth of the contact Hessian $H^{\mathcal{C}}$; the ratio $r^{\mathcal{C}} = \frac{\rho^{\mathcal{C}}}{m}$, where m is the number of contacts, determines the sparsity of $H^{\mathcal{C}}$. Over the simulation we observed $\max\left(r^{\mathcal{C}}\right) = 0.48$, $\min\left(r^{\mathcal{C}}\right) = 0.04$ and $\text{mean}\left(r^{\mathcal{C}}\right) = 0.18$, corroborating our previous postulate that our bands being small to medium sized. Solving the contact and friction QPs require factorizing and solving with the free Hessian $G^{\mathcal{C}}$ given by Equation (5.21) and the reduced Hessian $G^{\mathcal{F}}$ given by Equation (5.22), respectively. The number of contacts, dimensions of $G^{\mathcal{C}}$ and dimensions of $G^{\mathcal{F}}$ are given in Figure 7.6. These quantities peak at the time of maximum impact (≈ 0.1 seconds), and then decrease as the cylinders bounce apart (see Figure 7.3).

Figure 7.7 gives runtime comparisons of DIRECT and SCHURPA applied to the QP subproblems arising in SP. During the peak impact phase (≈ 0.1 seconds), SCHURPA outperforms DIRECT substantially. This occurs due to the more efficient factoring phase. Notice that the solving runtimes for SCHURPA are only marginally more than DIRECT, even though SCHURPA requires many solves per time step (see Figure 7.6). In contrast, DIRECT performed at most

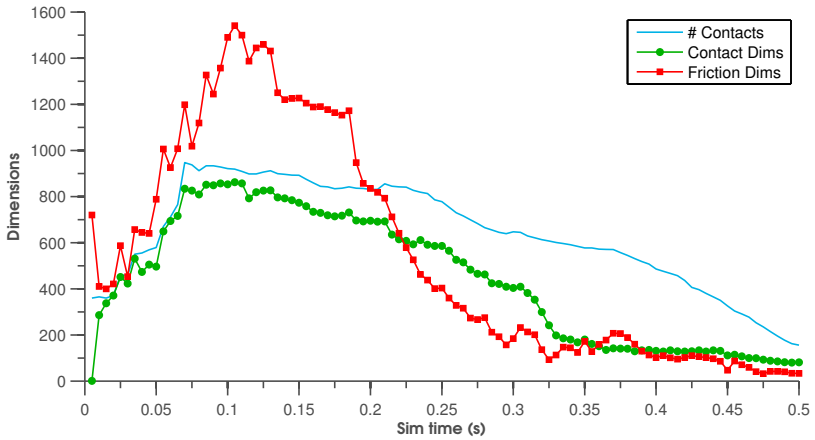


Figure 7.5: The number of contacts, dimensions of $G^{\mathcal{C}}$ and dimensions of $G^{\mathcal{F}}$ over the course of the simulation.

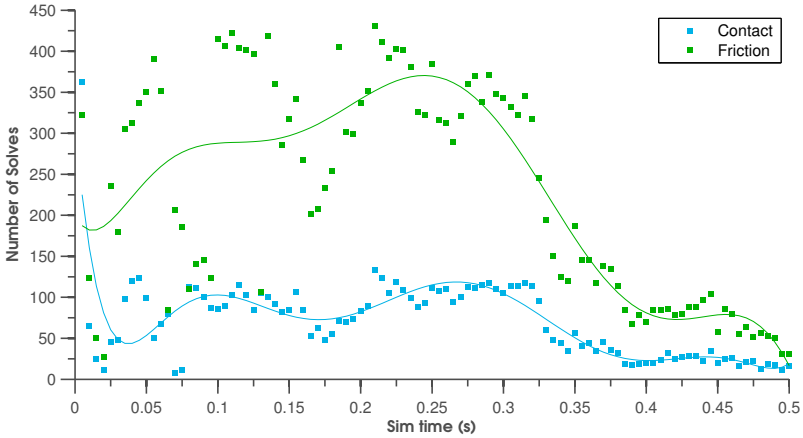


Figure 7.6: The number of solves performed in SCHURPA-SP for the contact and friction phases over the course of the simulation.

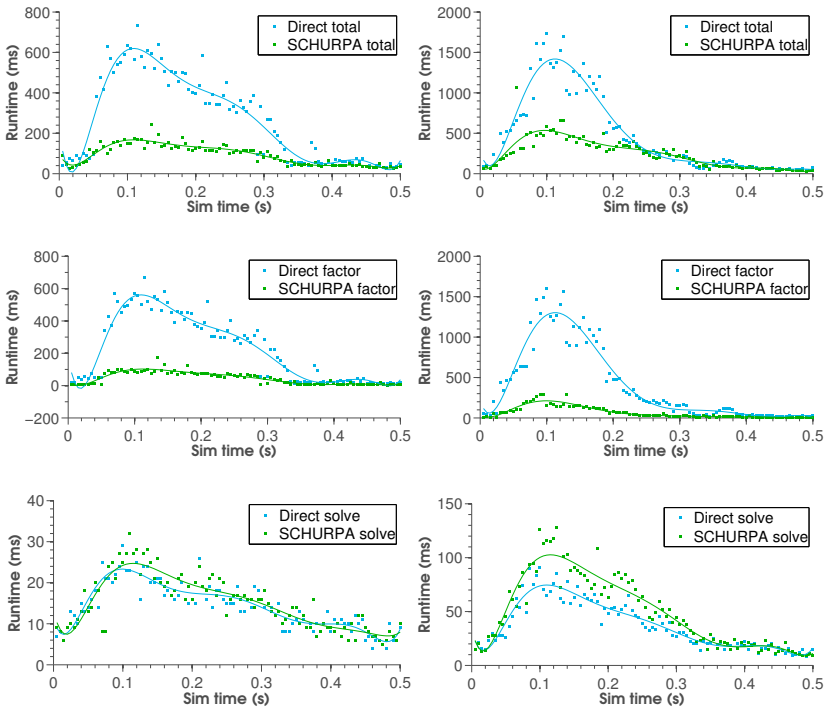


Figure 7.7: Runtimes of DIRECT and SCHURPA for the contact (left) and friction (right) solving phases over the course of the simulation. The total, factor and solve runtimes are given by the top, middle and bottom subfigures, respectively.

8 and 11 solves in the contact and friction phases, respectively, exemplifying the parallelization of the banded block solver. The total runtime speedup ratios of SCHURPA to DIRECT is given in Figure 7.8. We observe approximately a 4× and 3× speedup during the maximum impact for the contact and friction solving phases, respectively.

The presented results show the utility of SCHURPA applied to the contact and friction QP subproblems of SP. Equipped with a simple yet effective warm-starting technique, the number of iterations in SCHURPA and SP remained low. The multiple solves required by SCHURPA were efficiently handled by the parallel block solver, yielding superior performance as compared with DIRECT.

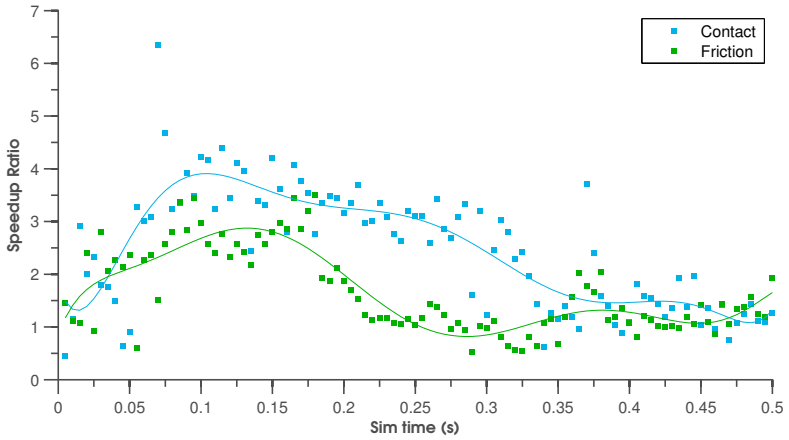


Figure 7.8: Total runtime speedup ratios of SCHURPA to DIRECT for the contact and friction phases over the course of the simulation.

Chapter 8

Conclusion

In this thesis we have developed a solution framework for frictional contact which incorporates the staggered projections algorithm, along with an efficient parallel implementation which exploits the underlying matrix structure of the QP subproblems. Doing so enabled an Eulerian solids simulator the ability to simulate frictional contact dynamics.

We first argued the merits of using active-set methods in the context of solving QPs arising in physical simulation due to their warm-starting capabilities. In particular, we desired a parallel active-set method to solve the QP subproblems of the staggered projections algorithm (Kaufman et al., 2008) to complement the parallel implementation of the simulator (Levin et al., 2011). We showed that there were two fundamental limitations attributed to parallelizing classical ASMs and their implementations. These were:

- classical ASMs could require many sequential iterations, and
- these iterations updated factorizations, an inherently sequential process.

To overcome these limitations, we introduced SCHURPA: a parallel implementation of PDASM using the Schur-complement method. The super-linear convergence property of PDASM overcomes the first limitation above, and the Schur-complement solution method to solving the saddle point systems arising in PDASM overcomes the second limitation by circumventing the factorization updating process.

We then reformulated the friction problem so as to be solvable by PDASM, and therefore SCHURPA. The reformulation incorporated a simplified linearized friction cone as the feasible set to the friction QP; convergence of PDASM applied to the reformulation could then be proven. We integrated SCHURPA into staggered projections to produce SCHURPA-SP, which only required the use of a single factorization for contact and friction, respectively, to solve all QP subproblems arising in staggered projections.

Further investigation of the structure of the contact and friction QPs in the simulator showed both problems may be phrased as banded SPD systems. A simple yet effective parallel block substitution method for banded SPD systems was designed and incorporated into a SCHURPA-SP implementation on the GPU using CUDA (NVIDIA, 2012b) to imbue the simulator with the ability to handle frictional contact.

The merits of SCHURPA were confirmed by the speedup attained over DIRECT, an implementation of PDASM which produces a factorization each iteration. Experiments run on generic randomly generated QPs demonstrated that, when using an initial working set defined by relatively small perturbations of the optimal active set, SCHURPA excels over DIRECT on both CPU and GPU implementations. The GPU implementation showed better performance for larger perturbations due to the parallelizability of the block solver; such perturbations are to be expected for large problems arising in physical simulation as the active-set is unknown and, unlike SQP-methods utilizing warm-starts, there is no active-set the QPs are converging to. Finally, we showed the improved performance of SCHURPA over DIRECT on a large-scale simulation using the simulator involving frictional contact.

Several avenues of future work could significantly improve SCHURPA's usability. More sophisticated warm-starting techniques could significantly reduce

the runtime of SCHURPA. A heterogenous CPU-GPU implementation would not only improve performance, but also broaden the algorithm's applicability to other simulations not confined to a GPU implementations. Along these lines, SCHURPA could be integrated into other simulation frameworks. For example, rigid body simulations share many of the same properties as the Eulerian solids simulator. Integration of SCHURPA would simply require the function that solves the initial saddle point system and could be extremely beneficial. Finally, ideas from SCHURPA could be used in a hybrid iterative-direct solver for the saddle point systems arising in ASMs to solve large, sparse problems.

Bibliography

Bartlett R.A. and Biegler L.T. QPSchur: A dual, active-set, Schur-complement method for large-scale and structured convex quadratic programming. *Optimization and Engineering*, 7(1):5–32, 2006.

Bergounioux M., Ito K., and Kunisch K. Primal-dual strategy for constrained optimal control problems. *SIAM Journal on Control and Optimization*, 37(4):1176, 1999.

Betts J.T. A Sparse Nonlinear Optimization Algorithm. *Journal of Optimization Theory and Applications*, 82(3):519–541, 1994.

Brunssen S., Schmid F., Schäfer M., and Wohlmuth B. A fast and robust iterative solver for nonlinear contact problems using a primal-dual active set strategy and algebraic multigrid. *International Journal for Numerical Methods in Engineering*, 69(3):524–543, 2007.

Daryina A.N. and Izmailov A.F. Semismooth newton method for quadratic programs with bound constraints. *Computational Mathematics and Mathematical Physics*, 49(10):1706–1716, 2009.

Davis T.A. and Hager W.W. Row modifications of a sparse cholesky factorization. *SIAM Journal on Matrix Analysis and Applications*, 26(3):621–639, 2006.

Gill P.E., Golub G., Murray W., and Saunders M. Methods for modifying matrix factorizations. *Mathematics of Computation*, 28(126):505–535, 1974.

Gill P.E., Murray W., Saunders M.A., and Wright M.H. Maintaining LU factors of a general sparse matrix. *Linear Algebra and its Applications*, 88:239–270, 1987.

Gill P.E., Murray W., Saunders M.A., and Wright M.H. A Schur-complement method for sparse quadratic programming. In *Reliable numerical computation*, Oxford Sci. Publ., 113–138. Oxford Univ. Press, New York, 1990.

Goldfarb D. and Idnani A. A numerically stable dual method for solving strictly convex quadratic programs. *Mathematical Programming*, 27(1):1–33, 1983.

Hintermuller M. The primal-dual active set method for a crack problem with non-penetration. *IMA Journal of Applied Mathematics*, 69(1):1–26, 2004.

Hintermüller M., Ito K., and Kunisch K. The primal-dual active set strategy as a semismooth newton method. *SIAM Journal on Optimization*, 13(3):865–888, 2002.

Hüeber S. and Wohlmuth B. A primal-dual active set strategy for non-linear multibody contact problems. *Computer Methods in Applied Mechanics and Engineering*, 194(27):3147–3166, 2005.

Humphrey J.R., Price D.K., Spagnoli K.E., Paolini A.L., and Kelmelis E.J. CULA: Hybrid GPU Accelerated Linear Algebra Routines. In *SPIE Defense, Security, and Sensing*. International Society for Optics and Photonics, 2010.

Ito K. and Kunisch K. *Lagrange multiplier approach to variational problems and applications*. Society for Industrial and Applied Mathematics, 2008.

Jean M. The non-smooth contact dynamics method. *Computer methods in applied mechanics and engineering*, 177(3):235–257, 1999.

Kaufman D.M. *Coupled principles for computational frictional contact mechanics*. Ph.D. thesis, New Brunswick Rutgers, The State University of New Jersey, 2009.

Kaufman D.M., Sueda S., James D.L., and Pai D.K. Staggered projections for frictional contact in multibody systems. *ACM Transactions on Graphics (TOG)*, 27(5):164:1–164:11, 2008.

Kunisch K. and Rendl F. An infeasible active set method for quadratic problems with simple bounds. *SIAM Journal on Optimization*, 14(1):35–52, 2003.

Kunisch K. and Rösch A. Primal-dual active set strategy for a general class of constrained optimal control problems. *SIAM Journal on Optimization*, 13(2):321–334, 2002.

Lanczos C. *The variational principles of mechanics*. Dover Publications, 1986.

Levin D.I.W., Litven J., Jones G.L., Sueda S., and Pai D.K. Eulerian solid simulation with contact. *ACM Transactions on Graphics (TOG)*, 30(4):36, 2011.

Moreau J. On unilateral constraints, friction and plasticity. *New Variational Techniques in Mathematical Physics*, 172–322, 1973.

Naumov M. Parallel Solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU. *NVIDIA Technical Report*, NVR-2011-0, 2011.

Nocedal J. and Wright S. *Numerical optimization*. Springer Verlag, 1999.

NVIDIA. Cublas : http://docs.nvidia.com/cuda/cublas/index.html. 2012a.

NVIDIA. Cuda : http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. 2012b.

NVIDIA. Cusparse : http://docs.nvidia.com/cuda/cusparse/index.html. 2012c.

NVIDIA. Thrust : http://docs.nvidia.com/cuda/thrust/index.html. 2012d.

Polizzi E. and Sameh A.H. A parallel hybrid banded system solver: the SPIKE algorithm. *Parallel Computing*, 32(2):177–194, 2006.

Qi L. and Sun J. A nonsmooth version of Newton's method. *Mathematical Programming*, 58(1-3):353–367, 1993.

Stewart D.E. Rigid-body dynamics with friction and impact. *SIAM Review*, 42(1):3–39, 2000.

Stewart D.E. and Trinkle J.C. An implicit time-stepping scheme for rigid body dynamics with inelastic collisions and coulomb friction. *International Journal for Numerical Methods in Engineering*, 39(15):2673–2691, 1996.

Stone J.E., Gohara D., and Shi G. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science and engineering*, 12(3):66–73, 2010.

Tomov S., Nath R., Ltaief H., and Dongarra J. Dense linear algebra solvers for multicore with GPU accelerators. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium*, 1–8. IEEE, 2010.

Ulbrich M. *Semismooth newton methods for variational inequalities and constrained optimization problems in function spaces*. Society for Industrial & Applied Mathematics, 2011.

Versteeg H. and Malalasekera W. *An Introduction to Computational Fluid Dynamics: The Finite Volume Method*. Prentice Hall, 2007.

Wright S.J. *Primal-dual Interior-point Methods*. Society for Industrial & Applied Mathematics, U.S., 1987.

Appendix A

Data Storage Formats

Dense matrices are stored in column-major order, which is defined for a matrix A as

$$A_{ij} = a[i + j * \text{lda}],$$

where a is the associated pointer to device memory and lda is the leading dimension, or stride, of the allocated memory storing A. A symmetric banded matrix B with kb sub-diagonals is stored in a (kb+1)×n matrix where row i contains the i^{th} sub-diagonal, beginning with the main diagonal, so that

$$B_{ij} = b[i - j + j * \text{ldb}] \quad \max(0,\text{i-kb}) \leq \text{j} \leq \text{i},$$

where ldb is the leading dimension of B.Table A.1 describes the data storage formats for dense and banded matrices on the GPU.

Matrix	Type	array	Data mapping
A	Dense	a	$A_{ij} = a[i + j * \text{lda}]$
B	Symmetric Banded	b	$B_{ij} = b[i - j + j * \text{ldb}]$, $\max(0, i - \text{kb}) \leq j \leq i$

Table A.1: Data storage formats for dense and symmetric banded matrices. For matrices A and B, a and b are their two-dimensional stored representation on GPU memory, respectively. lda and ldb are the leading dimensions of the arrays a and b, respectively, and kb is the number of sub/super diagonals of the matrix B.

Appendix B

Additional Code

The code B.1 contains the helper body code of the methods 6.3,Listing 6.4, and Listing 6.5. If a memory access other than a legal matrix element is attempted, 0 is returned, which pads the block with zeros. The resulting matrix multiplication of the sub-matrices in Equation (6.2) is then

$$\begin{bmatrix} B_{IK} & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} A_{KJ} & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} C_S & 0 \\ 0 & 0 \end{bmatrix},$$

obtaining the desired sub-matrix.

```
// helper functions
float getElement(const Matrix A, int i, int j){

    prec val = 0;

    if(i < A.height && j < A.width)
        val = A[i + j * A.ld];

    return val;
}

float getElement(const BandedMatrix B, int i, int j){

    prec val = 0.0;

    // if j > i, swap i & j
    int temp;
    if(j > i){
        temp = j;
        j = i;
        i = temp;
    }

    int min_j = max(0, i - B.rho);

    if(i < A.height && j < A.width && min_j <= j)
        val = B[i - j + j * ldb];

    return val;
}
void setElement(Matrix A, int i, int j, float value){

    if(i < A.height && j < A.width)
        A[i + j * A.ld] = value;
}
```

Listing B.1: Helper bodies for kernels.