Preconditioned Conjugate Gradient for Pivoting-based Complementarity Solvers in Multibody Simulations

Wing Hang Ho, School of Computer Science
McGill University, Montreal

August, 2022

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of

Master of Computer Science

©Wing Hang Henry Ho, 2022

Abstract

We present two numerical methods for solving stiff contact-rich physical systems modeled as mixed linear complementarity problems. The first approach combines a preconditioned conjugate gradient (PCG) solver and a block principal pivoting algorithm to solve large systems, whereas the second approach improves the robustness and convergence of the generalized conjugate gradient method through preconditioning. To improve convergence, we propose a variant of the zero-fill incomplete Cholesky preconditioner that uses imaginary values to avoid breakdown when negative diagonal elements are encountered during the factorization. Furthermore, we propose an adaptive regularization scheme to the friction rows of the system matrix to address variables that pivot too frequently, which improves convergence without introducing simulation artifacts. Various challenging scenarios are used to evaluate our proposed methods.

Abrégé

Nous présentons deux méthodes numériques pour résoudre des systèmes physiques de corps rigides riches en contacts modélisés comme des problèmes de complémentarité linéaire mixte. La première approche combine un solveur de gradient conjugué préconditionné (PCG) et un algorithme de pivotement principal de bloc pour résoudre de grands systèmes, tandis que la deuxième approche améliore la robustesse et la convergence de la méthode du gradient conjugué généralisé grâce au préconditionnement. Pour améliorer la convergence, nous proposons une variante du préconditionneur de la factorisation de Cholesky incomplète qui utilise des valeurs imaginaires pour éviter l'échec de la méthode lorsqu'il y a des éléments diagonaux négatifs lors de la factorisation. De plus, nous proposons un schéma de régularisation adaptatif aux lignes de friction de la matrice système pour traiter les variables qui pivotent trop fréquemment, ce qui améliore la convergence sans introduire d'incohérence de simulation. Divers scénarios difficiles sont utilisés pour évaluer nos méthodes proposées.

Acknowledgements

First of all, I would like to thank my supervisors Paul Kry and Sheldon Andrews for their patience and limitless support throughout my studies. It was their constant guidance that enable me to complete this thesis. I will always miss the 6 AM discussions with Paul and the past midnight discussions with Sheldon. I would also like to thank Xiao Wen Chang for his valuable feedback on my thesis and for giving me the chance to become a teaching assistant for the numerical computations course. Furthermore, I would like to thank Christopher Batty for encouraging me to pursue the field of physics-based animation during my undergraduate studies. Likewise, I am grateful to Sander Rhebergen and Lilia Krivodonova for their advice about conducting research.

Secondly, I would like to thank all group members in Paul's and Sheldon's group. It was a blast hanging out with you all, from lab meetings to our own group gatherings. I would like to thank Andreas and Quoc-Minh Ton-That in particular for all the research and mental support they provided me during my studies. Furthermore, I want to express my gratitude to Marek, Jozsef, Joe and Daniel for sharing research ideas with me and helping me with using the Vortex software for my research project.

Moreover, I would like to thank my friends and mentors at Waterloo: Brad Lushman, Yangang Chen, Jason Pye, Michael Waite, Edward Vrscay, Conrad Hewitt and all lab members from Christopher's, Sander's and Lilia's respective research groups for motivating me to pursue further studies. Special thanks to Michael Honke, Abdullah Ali Sivas, Brad and Yangang for always being there when I need someone to talk to.

Finally, deepest thanks to my parents for their unconditional love and support that allow me to keep up with my studies.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Interactive simulation of multibody dynamics with contact arises in many applications such as games and virtual-reality training. An important challenge, however, is the frictional contact problems that arise in systems with large mass ratios and many stiff and redundant constraints because this leads to large poorly conditioned problems. When systems are poorly conditioned, popular iterative methods struggle to converge to a solution, and a natural alternative is to make use of direct pivoting-based methods. However, these methods do not scale well to handle large problems.

In this thesis, we explore the modification of pivoting-based methods to make use of the preconditioned conjugate gradient (PCG) method. We propose two approaches. The first exploits the strengths of the block principal pivoting method by replacing the direct linear solver used within its inner iterations with a PCG solver. The second extends the generalized conjugate gradient method to use preconditioning. Because we are interested in simulating systems with poorly conditioned problems, preconditioning is critical to mitigating slow convergence. Specifically, we use an incomplete Cholesky preconditioner to exploit the sparsity of the systems, and in an effort to reduce the overall computation we exclusively use zero-fill incomplete Cholesky. We avoid the problem of negative diagonal elements that can arise in zero-fill incomplete Cholesky by proposing an inexpensive modification. We allow imaginary columns in the factorization, which permits a useful

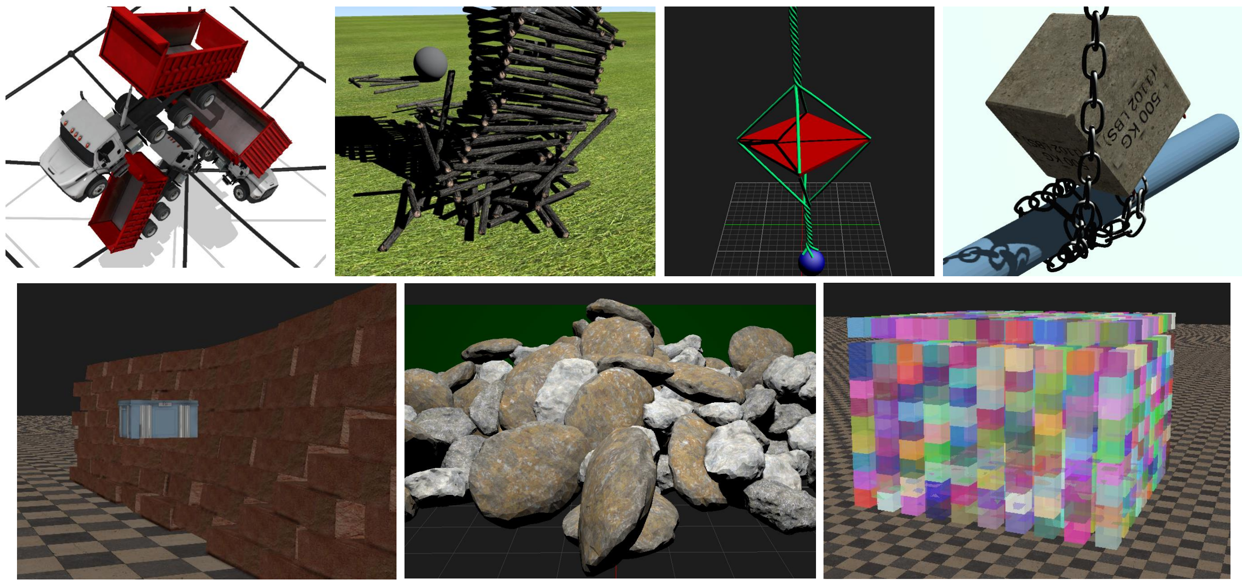


Figure 1.1: We evaluate the use of the preconditioned conjugate gradient method in pivoting-based solvers for these examples, which are challenging since they all have hundreds or thousands of constraints and contacts, and involve simulation with large mass ratios.

preconditioner to be factored without resorting to more costly approaches to deal with negative diagonals, such as revising an added diagonal compensation to the lead matrix and recomputing the preconditioner.

Figure 1.1 shows examples of the systems we simulate to evaluate our proposed methods. The proposed PCG-based solvers, when combined with our incomplete Cholesky preconditioner, yield reasonable convergence rates. We demonstrate that our incomplete Cholesky preconditioner is much more effective in speeding up the convergence of our PCG solvers than the Jacobi preconditioner and also comparable to the variant of the incomplete Cholesky that uses a relatively low drop tolerance to reduce fill-ins. We also propose an adaptive regularization technique for taming indecisive friction constraints (i.e., those that pivot too frequently) in order to reduce the number of iterations required to determine the correct index set. Finally, this thesis demonstrates the challenge of solving complicated contact problems, such that even with clever and efficient preconditioning, the PCG solvers still struggle to outperform the baseline direct solver due to the ill-conditioning of the contact problems we considered.

Due to the fact that the performance of direct methods is heavily dependent on the size of the system, this means the baseline direct solver will struggle as the number of constraints becomes orders of magnitude larger than the examples we considered. We believe this is when our PCG solvers will shine despite their limitations.

Chapter 2

Background and related work

A detailed introduction to the background and challenges of solving frictional contact problems for multibody dynamics can be found in the state of the art report of Bender et al. [4], and the recent course of Andrews and Erleben [2]. In this chapter, we review some of the content in both references that are related to the work of this thesis.

2.1 Equations of motion

The Newton-Euler equations give a second order ordinary differential equation that governs the dynamics of a physical system

$$\mathbf{M}(t)\dot{\mathbf{v}}(t) = \mathbf{f}(\mathbf{q}(t), \mathbf{v}(t), t), \tag{2.1}$$

where $\mathbf{q}(t)$ is the generalized position of the bodies in the physical system, $\mathbf{v}(t)$ is the generalized velocity of the bodies in the system, $\dot{\mathbf{v}}(t)$ is the acceleration, $\mathbf{f}$ is the generalized force acting on the system and $\mathbf{M}(t)$ is the mass of the bodies in the system. The term generalized is used to describe the variables that parameterize the equations of motion. Note that all terms in Equation 2.1 are dependent on time t. Hence, in the remainder of this thesis, we will assume such time dependency and omit the time parameter.

We treat each body in the system as a rigid body, meaning the distance between points in the body is fixed. In other words, the bodies in our system do not undergo any deformation. For each rigid body moving in three-dimensional space, there are 3 positional degrees of freedom and 3 rotational degrees of freedom. So for a physical system with n rigid bodies, the generalized velocity $\mathbf{v}$ and acceleration $\dot{\mathbf{v}}$ are $6n \times 1$ vectors. The generalized position provides the position and orientation of each body in the system. We use unit quaternions (a_i, b_i, c_i, d_i) to define the i-th body's orientation. With the 3 positional degrees of freedom, the generalized position of the system $\mathbf{q}$ is a $7n \times 1$ vector. The generalized positions are related to the generalized velocities in the following way

$$\dot{\mathbf{q}} = \mathbf{H}(\mathbf{q})\mathbf{v}, \tag{2.2}$$

where $\mathbf{H}$ is the $7n \times 6n$ block diagonal kinematic mapping matrix defined as

$$\mathbf{H} = \begin{bmatrix} \mathbb{I}_{3\times3} & & & & \\ & \mathbf{H}_i & & & \\ & & \ddots & & \\ & & & \mathbb{I}_{3\times3} & \\ & & & & \mathbf{H}_n \end{bmatrix}, \quad \text{with} \quad \mathbf{H}_i = \frac{1}{2}\begin{bmatrix} -b_i & -c_i & -d_i \\ a_i & d_i & -c_i \\ -d_i & a_i & b_i \\ c_i & -b_i & a_i \end{bmatrix}, \tag{2.3}$$

and $\mathbb{I}_{3\times3}$ is the 3-by-3 identity matrix.

The generalized forces in the system $\mathbf{f}$ is a $6n \times 1$ vector, as each body has linear forces, such as gravity, and torque applied to it. As for the mass matrix $\mathbf{M}$ of the system, it is a $6n \times 6n$ symmetric block diagonal matrix, where each block includes the linear mass m_i and the symmetric moment of inertia tensor $\mathbf{I}_i$ of each body i

$$\mathbf{M}_i = \begin{bmatrix} m_i\mathbb{I}_{3\times3} & \\ & \mathbf{I}_i \end{bmatrix}. \tag{2.4}$$

The moment of inertia tensor is positive definite, and if we consider rotating bodies around the principal axes, the inertia tensor is a diagonal matrix with positive entries. Hence, the mass matrix $\mathbf{M}$ is positive definite.

In the remainder of the thesis, when referring to the position and velocity of bodies in the system, we are referring to the generalized positions and velocities.

2.1.1 Discretizing the equations of motion

In order to simulate the dynamics at a particular instant of time t, we apply numerical integrators to discretize the equations of motion in terms of time. To do so, we choose a time step h and approximate $\dot{\mathbf{v}}$ using finite differences

$$\dot{\mathbf{v}} \approx \frac{\mathbf{v}^+ - \mathbf{v}}{h}, \tag{2.5}$$

where $\mathbf{v}$ denotes the velocity at the beginning of the time step and $\mathbf{v}^+$ is the velocity at the end of the time step. We solve the following discretized equations of motion for $\mathbf{v}^+$

$$\mathbf{M}\mathbf{v}^+ = \mathbf{M}\mathbf{v} + h\mathbf{f}, \tag{2.6}$$

where the terms on the right hand side are known quantities that are evaluated at the beginning of the time step. We use $\mathbf{v}^+$ to update the position at the end of the time step

$$\mathbf{q}^+ = \mathbf{q} + h\mathbf{H}(\mathbf{q})\mathbf{v}^+. \tag{2.7}$$

This time-stepping integrator is called the semi-implicit Euler method. The difference between the semi-implicit Euler method and the forward Euler method is semi-implicit Euler method uses $\mathbf{v}^+$ to update the position $\mathbf{q}^+$ rather than using $\mathbf{v}$. We use the semi-implicit Euler method because we can choose larger time steps without encountering numerical stability issues. Although we can instead consider using the backward Euler method, where we use $\mathbf{f}^+$ to update $\mathbf{v}^+$, and not worry about stability issues in our choice of the time step h, the method is more costly due to needing to solve a linear system for $\mathbf{v}^+$. Similar to any Euler time integration scheme, the semi-implicit time-integrator is first-order accurate, meaning it has a local truncation error of $O(h^2)$ and global error of $O(h)$. An important thing to note is for our problems, we only consider sliding motions on flat, isotropic planar surfaces, so the sliding motion is linear, and hence using linear first-order accurate numerical integrator is sufficient [2].

2.2 Constraints

In physics-based computer animation, we would like to accurately model interactions between bodies in the system. To do so, we need to enforce constraints on the position or velocity of the bodies.

Bilateral constraints are used to model positional constraints and joint constraints between two or more bodies. Bilateral constraints can be expressed as the equality constraint for some scalar function ϕ

$$\phi(\mathbf{q}) = 0. \tag{2.8}$$

Unilateral constraints are used to simulate contact between bodies. We want to satisfy the inequality constraint of a scalar function ϕ of the form

$$\phi(\mathbf{q}) \geq 0, \tag{2.9}$$

where the constraint $\phi = 0$ denotes two objects that are in contact with each other (Figure 2.1a), and $\phi > 0$ means that the objects are separated from each other (Figure 2.1b). The unilateral contact constraint helps prevent objects from interpenetrating each other (Figure 2.1c). Unilateral constraints are also used to model friction, which we will discuss in Section 2.4.

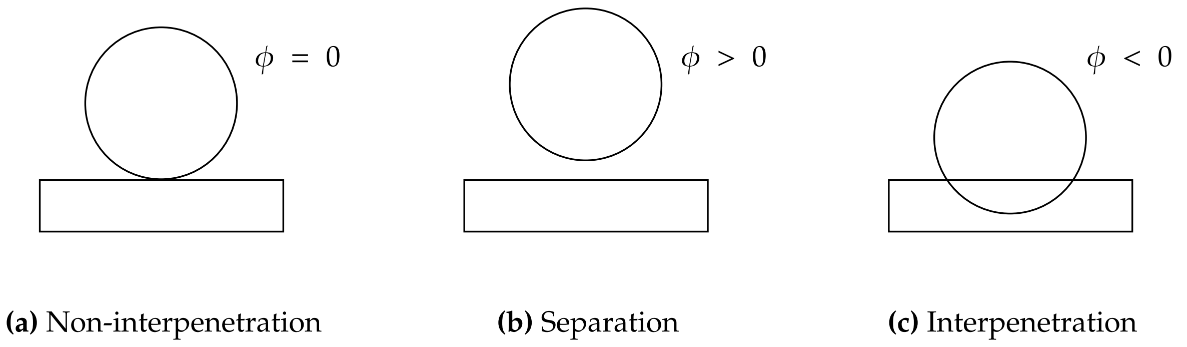


Figure 2.1: A notable example of unilateral constraints is the non-interpenetration constraint that is used to model contact between two bodies.

To make sure the bodies satisfy all constraints $\boldsymbol{\phi} \in \mathbb{R}^m$ in the system at the end of the time step, we enforce the constraints at the velocity level. We consider

$$\dot{\boldsymbol{\phi}} = \frac{\partial \boldsymbol{\phi}}{\partial \mathbf{q}} \frac{\partial \mathbf{q}}{\partial t} = \frac{\partial \boldsymbol{\phi}}{\partial \mathbf{q}} \mathbf{H}(\mathbf{q})\mathbf{v} = \mathbf{J}\mathbf{v}, \tag{2.10}$$

where $\mathbf{J} \in \mathbb{R}^{m \times 6n}$ is the constraint Jacobian matrix. The system of bilateral constraint equations in the velocity-level can be written as

$$\mathbf{J}\mathbf{v} = \mathbf{0}, \tag{2.11}$$

and the velocity-level unilateral constraint equations are

$$\mathbf{J}\mathbf{v} \geq \mathbf{0}. \tag{2.12}$$

One can also consider enforcing the constraints at the acceleration level, for example

$$\ddot{\boldsymbol{\phi}} = \dot{\mathbf{J}}\mathbf{v} + \mathbf{J}\dot{\mathbf{v}} = \mathbf{0}, \tag{2.13}$$

in order to make sure the position and velocity constraints are satisfied. However, due to the additional computational cost in computing the time derivative of the constraint Jacobian matrix $\dot{\mathbf{J}}$, we only enforce constraints at the velocity level in this thesis.

Other than enforcing position and velocity constraints, we also need to account for a constraint force that is used to make sure the bodies satisfy the constraints without doing actual work. Our discretized equations of motion become

$$\mathbf{M}\mathbf{v}^{+} - \mathbf{J}^{T}\boldsymbol{\lambda}^{+} = \mathbf{M}\mathbf{v} + h\mathbf{f}, \tag{2.14}$$

subject to constraints

$$\mathbf{J}\mathbf{v}^{+} = \mathbf{w}, \tag{2.15}$$

where $\boldsymbol{\lambda}^{+}$ is the constraint impulse at the end of the time step that we also need to solve for. Here, we are using a general form of the constraint velocity $\mathbf{w}$, where the constraint velocity $\mathbf{w}_i = 0$ if constraint i is a bilateral constraint and $\mathbf{w}_i \geq 0$ if it is a unilateral constraint. Notice that in Equation 2.15, we are using the Jacobian evaluated at the beginning of the time step rather than the Jacobian evaluated at the end of the time step $\mathbf{J}^{+}$. The reason is we want to avoid the expensive computation of the time derivative of the Jacobian matrix $\dot{\mathbf{J}}$ when Taylor approximating $\mathbf{J}^{+}$. Since unilateral constraints are dominant in the examples we consider, we illustrate how the constraint Jacobian for different unilateral constraints are computed in Appendix A.

2.3 Constrained rigid body systems

Combining Equations 2.14 and 2.15 gives the linear system

$$\begin{bmatrix} \mathbf{M} & -\mathbf{J}^{T} \\ \mathbf{J} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{v}^{+} \\ \boldsymbol{\lambda}^{+} \end{bmatrix} = \begin{bmatrix} \mathbf{M}\mathbf{v} + h\mathbf{f} \\ \mathbf{w} \end{bmatrix}, \tag{2.16}$$

where we need to solve for the velocities $\mathbf{v}^{+}$ and constraint impulses $\boldsymbol{\lambda}^{+}$ at the end of the time step. To eliminate the variable $\mathbf{v}^{+}$, we apply the Schur complement of the mass matrix M in Equation 2.16 and get a mixed linear complementarity problem (MLCP) with a reduced linear system where we only need to solve for the constraint impulses $\boldsymbol{\lambda}^+$:

$$\underbrace{\mathbf{J}\mathbf{M}^{-1}\mathbf{J}^T}_{\mathbf{A}}\boldsymbol{\lambda}^+ + \underbrace{\mathbf{J}\mathbf{M}^{-1}(\mathbf{M}\mathbf{v} + h\mathbf{f})}_{\mathbf{b}} = \mathbf{w}. \tag{2.17}$$

We will discuss MLCPs in Section 2.4, which essentially enforce bounds and conditions on the constraint impulses $\boldsymbol{\lambda}$ and constraint velocities $\mathbf{w}$ such that they can only represent physically correct quantities. The bounds on the constraint impulses depend on the constraint type.

2.3.1 Constraint violation

There are, however, two problems with the reduced linear system in Equation 2.17. Firstly, the matrix $\mathbf{A}$ may be rank deficient. The reason is most likely due to the existence of redundant constraints when modeling contact such that there are rows in the constraint Jacobian matrix $\mathbf{J}$ that are linearly dependent. The second problem with Equation 2.17 is we are assuming that the position constraints are satisfied when enforcing the velocity constraints in Equation 2.15. However, this assumption may not be true due to numerical drift. To explain why numerical drift occurs, suppose we want two bodies to remain in contact. In this case, the constraint function at the beginning of the time step will be zero, meaning $\boldsymbol{\phi}(\mathbf{q}^-) = \mathbf{0}$. Since we are enforcing constraints at the velocity-level, the time derivative of the constraint function, or the constraint velocity, at the beginning of the time step will be zero as well, where $\dot{\boldsymbol{\phi}}(\mathbf{q}^-) = \mathbf{J}\mathbf{v}^- = \mathbf{0}$. For two bodies to remain in contact at the end of the time step, we want to enforce $\boldsymbol{\phi}(\mathbf{q}^+) = \mathbf{0}$. Since $\boldsymbol{\phi}(\mathbf{q}^+)$ is not known, we apply the Taylor series approximation:

$$\boldsymbol{\phi}(\mathbf{q}^+) \approx \boldsymbol{\phi}(\mathbf{q}^-) + h\dot{\boldsymbol{\phi}}(\mathbf{q}^-) + \frac{h^2}{2}\ddot{\boldsymbol{\phi}}(\mathbf{q}^-) + O(h^3) \tag{2.18}$$

$$= \frac{h^2}{2}\ddot{\boldsymbol{\phi}}(\mathbf{q}^-) + O(h^3) \tag{2.19}$$

The issue is the second order term is non-zero, where $\ddot{\boldsymbol{\phi}}(\mathbf{q}^-) \neq \mathbf{0}$ due to only considering constraints at the velocity level. Hence, this error will accumulate regardless of our choice of the time step h. The other cause of numerical drift is due to round-off errors during numerical computations. Numerical drift will cause bodies to, for example, interpenetrate each other.

2.3.2 Constraint stabilization

There are multiple ways to fix the issue of constraint violations. Examples are Baumgarte stabilization [3], fast projection [15] and post-step stabilization [7]. In our work, we apply the Baumgarte stabilization technique [3]. Since we only enforce constraints at the velocity level, we consider

$$\boldsymbol{\lambda}_j^+ = -hk_j\boldsymbol{\phi}_j^+ - hb_j\dot{\boldsymbol{\phi}}_j^+. \tag{2.20}$$

for the j-th constraint in the system. The physical intuition of the Baumgarte stabilization technique is we apply a spring-like restoration impulse to bring the position of the bodies back to the constraint manifold, where k_j and b_j are the stiffness and damping coefficients of the Hookean spring, and $\boldsymbol{\phi}_j^+$ is the j-th constraint function evaluated with the position of the bodies at the end of the time step. We use a first-order accurate implicit integrator such as backward Euler to approximate $\boldsymbol{\phi}_j^+$

$$\boldsymbol{\phi}_j^+ \approx \boldsymbol{\phi}_j + h\dot{\boldsymbol{\phi}}_j^+, \tag{2.21}$$

in order to avoid numerical stability issues from using a stiff spring-like restoration impulse. We can rewrite Equation 2.20 as

$$(hb_j + h^2k_j)\dot{\boldsymbol{\phi}}_j^+ + \boldsymbol{\lambda}_j^+ = -hk_j\boldsymbol{\phi}_j. \tag{2.22}$$

Dividing the equation by $hb_j + h^2k_j$ gives

$$\dot{\boldsymbol{\phi}}_j^+ + \underbrace{\left(\frac{1}{hb_j + h^2k_j}\right)}_{\epsilon_j} \boldsymbol{\lambda}_j^+ = -\underbrace{\left(\frac{hk_j}{b_j + hk_j}\right)}_{\gamma_j} \frac{\boldsymbol{\phi}_j}{h}. \tag{2.23}$$

Considering individual stiffness and damping coefficients for each constraint, and the fact that $\dot{\boldsymbol{\phi}} = \mathbf{J}\mathbf{v}$, the velocity constraint with Baumgarte stabilization is

$$\mathbf{J}\mathbf{v}^+ + \mathbf{C}\boldsymbol{\lambda}^+ = -\boldsymbol{\Gamma}\frac{\boldsymbol{\phi}}{h}, \tag{2.24}$$

where $\mathbf{C}, \boldsymbol{\Gamma} \in \mathbb{R}^{m\times m}$ are diagonal matrices such that

$$\mathbf{C} = \begin{bmatrix} \epsilon_1 & & \\ & \ddots & \\ & & \epsilon_m \end{bmatrix} \quad \text{and} \quad \boldsymbol{\Gamma} = \begin{bmatrix} \gamma_1 & & \\ & \ddots & \\ & & \gamma_m \end{bmatrix}. \tag{2.25}$$

One issue with the Baumgarte stabilization technique is it can be tricky to pick appropriate values for ϵ_j and γ_j as they are dependent on the constraint type and what is being simulated. One can directly tune ϵ_j and γ_j, or can instead go for a more intuitive approach of tuning the stiffness and damping parameters of the implicit spring impulse.

Combining Equations 2.14 and 2.24 gives the linear system

$$\begin{bmatrix} \mathbf{M} & -\mathbf{J}^T \\ \mathbf{J} & \mathbf{C} \end{bmatrix} \begin{bmatrix} \mathbf{v}^+ \\ \boldsymbol{\lambda}^+ \end{bmatrix} = \begin{bmatrix} \mathbf{M}\mathbf{v} + h\mathbf{f} \\ \mathbf{w} - \boldsymbol{\Gamma}\frac{\boldsymbol{\phi}}{h} \end{bmatrix}. \tag{2.26}$$

The compliance matrix $\mathbf{C}$ is used to handle redundant constraints. Since the diagonals of the compliance matrix are positive, it makes the matrix $\mathbf{A}$ positive definite. The constraint violation term $\boldsymbol{\Gamma}\frac{\boldsymbol{\phi}}{h}$ is used to combat numerical drift. Taking the Schur complement of the

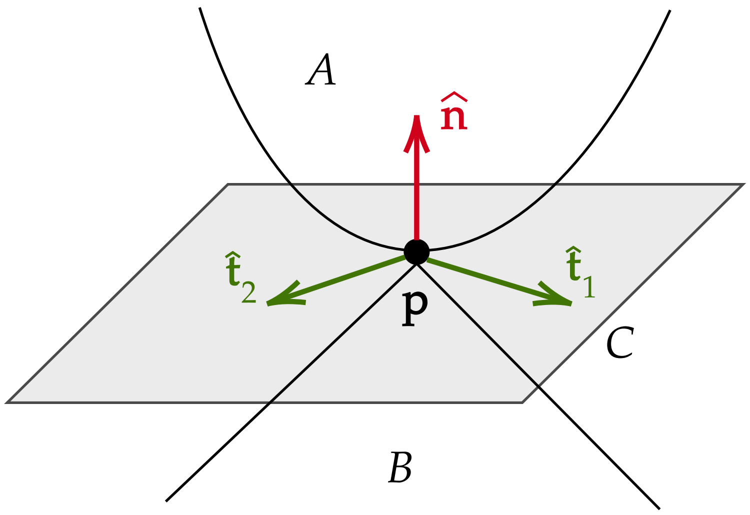


Figure 2.2: Local contact frame when surfaces A and B are in contact.

mass matrix M in Equation 2.26 gives the more compact linear system

$$\underbrace{\left(\mathbf{J}\mathbf{M}^{-1}\mathbf{J}^{T}+\mathbf{C}\right)}_{\mathbf{A}}\boldsymbol{\lambda}^{+}+\underbrace{\boldsymbol{\Gamma}\frac{\phi}{h}+\mathbf{J}\mathbf{M}^{-1}(\mathbf{M}\mathbf{v}+h\mathbf{f})}_{\mathbf{b}}=\mathbf{w}, \tag{2.27}$$

which only requires solving for the constraint impulses $\boldsymbol{\lambda}^{+}$. Here, M is block diagonal and so it is easy to invert. The matrix $\mathbf{A}$ is also symmetric. For the examples we are considering, most of our constraints are (unilateral) contact constraints between two bodies, which means the constraint Jacobian matrix $\mathbf{J}$ is sparse. Despite our examples having a large number of contact constraints, the number of bodies each body is in contact with is small, and so the number of contact constraints with overlapping bodies in the Jacobian matrix $\mathbf{J}$ is small. Hence the matrix $\mathbf{A}$ is sparse.

2.4 Modelling friction

In order to account for the friction impulse generated when there is sliding between two surfaces A and B, we consider a local contact frame spanned by the unit normal $\hat{\mathbf{n}}$ at contact point $\mathbf{p}$ of the contact plane C and two unit vectors $\hat{\mathbf{t}}_1$ and $\hat{\mathbf{t}}_2$ that are tangential to the contact plane C (Figure 2.2), where $\hat{\mathbf{n}}, \hat{\mathbf{t}}_1, \hat{\mathbf{t}}_2 \in \mathbb{R}^3$. The vectors $\hat{\mathbf{t}}_1$ and $\hat{\mathbf{t}}_2$ define the friction axes. In this local contact frame, the contact space velocity $\mathbf{w}$ and impulse $\boldsymbol{\lambda}$ each have three components, with $\mathbf{w} = (w_{\hat{\mathbf{n}}}, w_{\hat{\mathbf{t}}_1}, w_{\hat{\mathbf{t}}_2})$ and $\boldsymbol{\lambda} = (\lambda_{\hat{\mathbf{n}}}, \lambda_{\hat{\mathbf{t}}_1}, \lambda_{\hat{\mathbf{t}}_2})$. The relative tangential velocity at the contact point $\mathbf{w}_{\hat{\mathbf{t}}}$ is expressed as a linear combination of the friction axes $\hat{\mathbf{t}}_1$ and $\hat{\mathbf{t}}_2$, where $\mathbf{w}_{\hat{\mathbf{t}}} = w_{\hat{\mathbf{t}}_1}\hat{\mathbf{t}}_1 + w_{\hat{\mathbf{t}}_2}\hat{\mathbf{t}}_2$. Similarly, the tangential impulse, or the friction impulse, between two sliding objects is expressed as $\boldsymbol{\lambda}_{\hat{\mathbf{t}}} = \lambda_{\hat{\mathbf{t}}_1}\hat{\mathbf{t}}_1 + \lambda_{\hat{\mathbf{t}}_2}\hat{\mathbf{t}}_2$. We refer the readers to Appendix A.2 on how to compute the constraint Jacobian when frictional contact is present in the simulation.

2.4.1 Coulomb friction model

We use the Coulomb friction model to describe the friction impulse between two surfaces. Coulomb friction couples the normal impulse $\lambda_{\hat{\mathbf{n}}}$ and the friction impulse $\boldsymbol{\lambda}_{\hat{\mathbf{t}}}$ using the exact isotropic planar Coulomb friction cone constraint [2]

$$\|\boldsymbol{\lambda}_{\hat{\mathbf{t}}}\| \leq \mu \lambda_{\hat{\mathbf{n}}}, \tag{2.28}$$

where μ is the coefficient of friction that is dependent on the materials that are in contact. The geometric interpretation of the constraint in Equation 2.28 is a quadratic cone illustrated in Figure 2.3. The Coulomb friction cone models the isotropic Coulomb friction law such that if there is sliding motion between two objects, then the friction impulse opposes the sliding motion and will be at its maximum, where

$$\mu \lambda_{\hat{\mathbf{n}}} - \|\boldsymbol{\lambda}_{\hat{\mathbf{t}}}\| = 0, \quad \|\mathbf{w}_{\hat{\mathbf{t}}}\| > 0. \tag{2.29}$$

On the other hand, if the objects are not sliding, then the friction impulse can act in any direction and needs to satisfy the inequality in Equation 2.28 such that it is enough to counteract other impulses (Figure 2.3b). We can express this with

$$\mu \lambda_{\hat{\mathbf{n}}} - \|\boldsymbol{\lambda}_{\hat{\mathbf{t}}}\| > 0, \quad \|\mathbf{w}_{\hat{\mathbf{t}}}\| = 0. \tag{2.30}$$

Note that Equations 2.29 and 2.30 imply

$$(\mu\lambda_{\hat{\mathbf{n}}} - \|\boldsymbol{\lambda}_{\hat{\mathbf{t}}}\|)\,\|\mathbf{w}_{\hat{\mathbf{t}}}\| = 0. \tag{2.31}$$

We concisely write Equations 2.29 - 2.31 as a nonlinear complementarity problem (NCP)

$$\mu\lambda_{\hat{\mathbf{n}}} - \|\boldsymbol{\lambda}_{\hat{\mathbf{t}}}\| \geq 0 \;\perp\; \|\mathbf{w}_{\hat{\mathbf{t}}}\| \geq 0, \tag{2.32}$$

where we use the complementarity condition $a \geq 0 \;\perp\; b \geq 0$ to express $a \geq 0,\, b \geq 0$ and $ab = 0$.

To deal with the NCP in Equation 2.32, one can rewrite it as a nonlinear function in order to apply Newton's method to linearize a system of nonlinear equations and solve for the constraint impulses $\boldsymbol{\lambda}^{+}$ that satisfies the inequality in Equation 2.28 [20,22]. However, this involves the expensive Hessian computation. In order to avoid this costly computation, we consider using an approximation of the Coulomb friction cone. In the next section, we will introduce a cheap yet effective method to linearize the Coulomb friction cone.

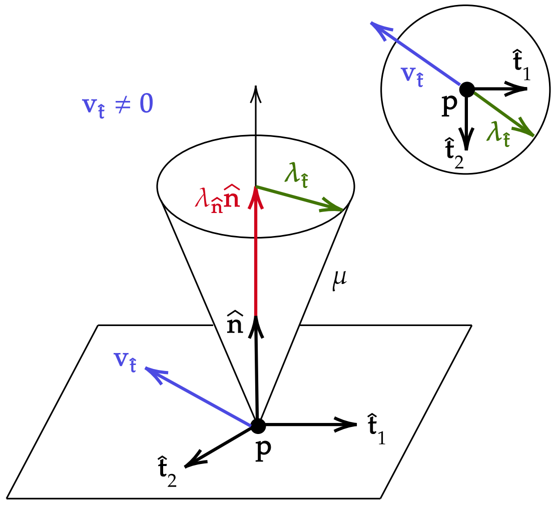
2.4.2 Box approximation of Coulomb friction cone

We use a box approximation to linearize the Coulomb friction cone (Figure 2.4). With this approximation, we can express the friction bounds in each friction direction using two independent inequalities

$$\lambda^{l}_{\hat{\mathbf{t}}_1} \leq \lambda_{\hat{\mathbf{t}_1}} \leq \lambda^{u}_{\hat{\mathbf{t}}_1}, \tag{2.33}$$

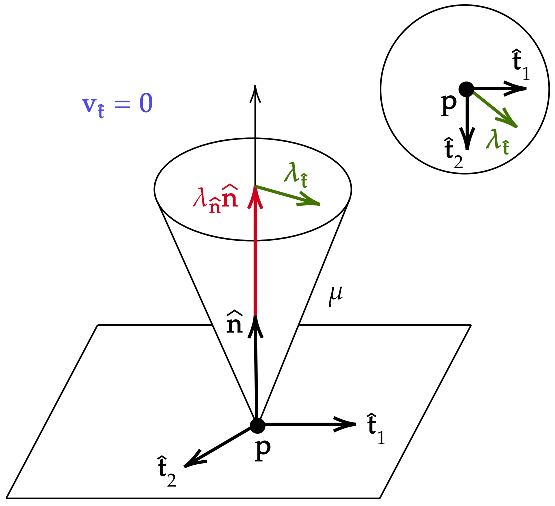
$$\lambda^{l}_{\hat{\mathbf{t}}_2} \leq \lambda_{\hat{\mathbf{t}_2}} \leq \lambda^{u}_{\hat{\mathbf{t}}_2}, \tag{2.34}$$

where $\lambda^{l}_{\hat{\mathbf{t}}_i} = -\mu\lambda_{\hat{\mathbf{n}}}$ and $\lambda^{u}_{\hat{\mathbf{t}}_i} = \mu\lambda_{\hat{\mathbf{n}}}$. To compute the friction bounds, one can first determine the normal impulse $\lambda_{\hat{\mathbf{n}}}$ for each contact constraint by assuming the physical system is frictionless, then solve Equation 2.27 for $\lambda_{\hat{\mathbf{n}}}$ for each contact constraint, and then compute



$\mathbf{v}_{\hat{\mathbf{t}}} \neq 0$
$\mathbf{v}_{\hat{\mathbf{t}}}$
$\hat{\mathbf{t}}_1$
p
$\hat{\mathbf{t}}_2$
$\lambda_{\hat{\mathbf{t}}}$
$\lambda_{\hat{\mathbf{n}}}\hat{\mathbf{n}}$
$\lambda_{\hat{\mathbf{t}}}$
μ
$\hat{\mathbf{n}}$
$\mathbf{v}_{\hat{\mathbf{t}}}$
$\hat{\mathbf{t}}_1$
$\hat{\mathbf{t}}_2$
p

(a) Sliding friction



$\mathbf{v}_{\hat{\mathbf{t}}} = 0$
$\hat{\mathbf{t}}_1$
p
$\hat{\mathbf{t}}_2$
$\lambda_{\hat{\mathbf{t}}}$
$\lambda_{\hat{\mathbf{n}}}\hat{\mathbf{n}}$
$\lambda_{\hat{\mathbf{t}}}$
μ
$\hat{\mathbf{n}}$
$\hat{\mathbf{t}}_1$
$\hat{\mathbf{t}}_2$
p

(b) Sticking friction

Figure 2.3: The nonlinear Coulomb friction model that is used to model isotropic friction.

the approximate friction bounds using $\lambda_{\hat{\mathbf{n}}}$ and the friction coefficient μ. Depending on the magnitude of the normal impulse, we can get friction bounds that are larger or smaller than the actual Coulomb friction bounds in Equation 2.28. To avoid solving Equation 2.27 for the normal impulses $\lambda_{\hat{\mathbf{n}}}$ in order to compute the box friction bounds, one can manually set $\lambda^l_{\hat{\mathbf{t}}_i}$ and $\lambda^u_{\hat{\mathbf{t}}_i}$. While this approach to determining the friction bounds is cheap, it is inaccurate as it essentially decouples the normal and friction impulse and will yield artifacts such as objects being too slippery if inappropriate friction bounds are chosen.

Box Linear Complementarity Problem

To obey the approximate Coulomb friction laws, we have to satisfy the following three conditions for the box friction model for contact constraint i

$$\mathbf{w}_i > 0 \quad \text{if} \quad \boldsymbol{\lambda}^+_i = \boldsymbol{\lambda}^l_i, \tag{2.35a}$$

$$\mathbf{w}_i < 0 \quad \text{if} \quad \boldsymbol{\lambda}^+_i = \boldsymbol{\lambda}^u_i, \tag{2.35b}$$

$$\mathbf{w}_i = 0 \quad \text{if} \quad \boldsymbol{\lambda}^l_i \leq \boldsymbol{\lambda}^+_i \leq \boldsymbol{\lambda}^u_i, \tag{2.35c}$$

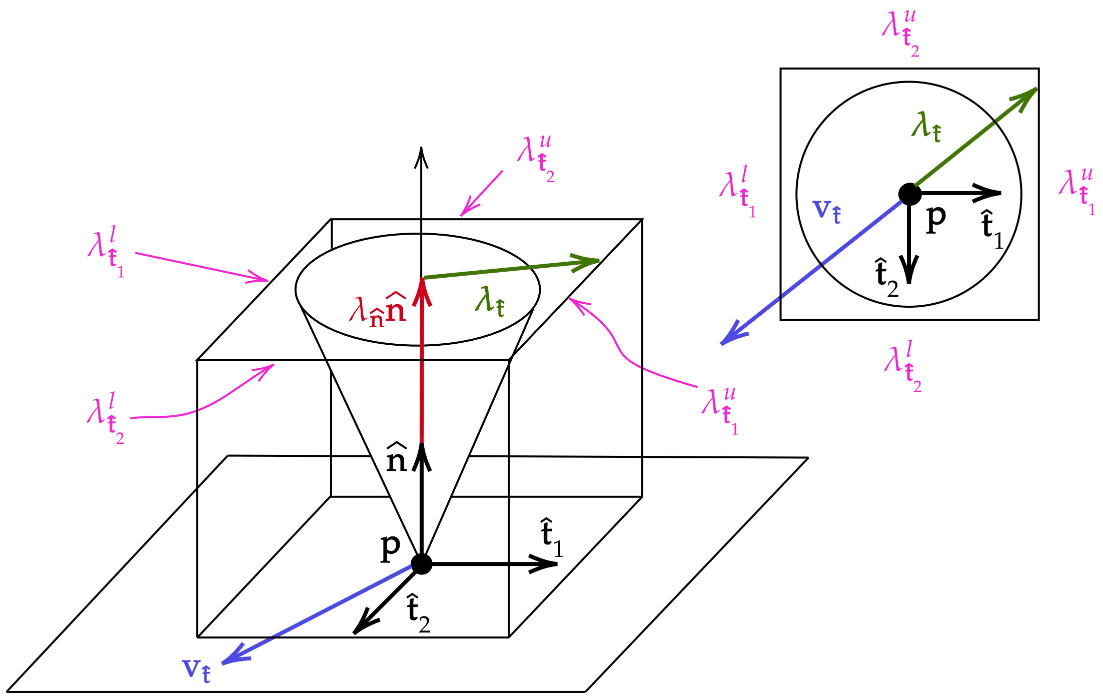


Figure 2.4: Box approximation of the Coulomb friction cone.

where $\mathbf{w}_i$ is the constraint velocity of contact constraint i, and $\boldsymbol{\lambda}^l_i$ and $\boldsymbol{\lambda}^u_i$ are the lower and upper bounds of contact constraint i. Equations 2.35a or 2.35b produces sliding friction, where the friction impulse $\boldsymbol{\lambda}^+_i$ at the end of the time step is at its maximum and opposes the constraint velocity $\mathbf{w}_i$, which in this case is the relative tangential contact velocity. Equation 2.35c produces sticking friction, where the objects are not sliding and the friction impulse is just strong enough to counteract other impulses acting on the objects. Equations 2.35a or 2.35c can also be used to model non-interpenetration constraints. In that case, $\mathbf{w}_i$ will be the normal component of the relative contact velocity and $\boldsymbol{\lambda}^+_i$ will be the normal impulse such that $\boldsymbol{\lambda}^l_i = 0$ and $\boldsymbol{\lambda}^u_i = \infty$ (Figure 2.5). In other words, we can conveniently express friction and non-interpenetration constraints using the same set of equations listed in 2.35a - 2.35c. As for bilateral constraints, the bounds will be $\boldsymbol{\lambda}^l_i = -\infty$ and $\boldsymbol{\lambda}^u_i = \infty$, as we do not bound the impulses but enforce constraints on the positional degrees of freedom.

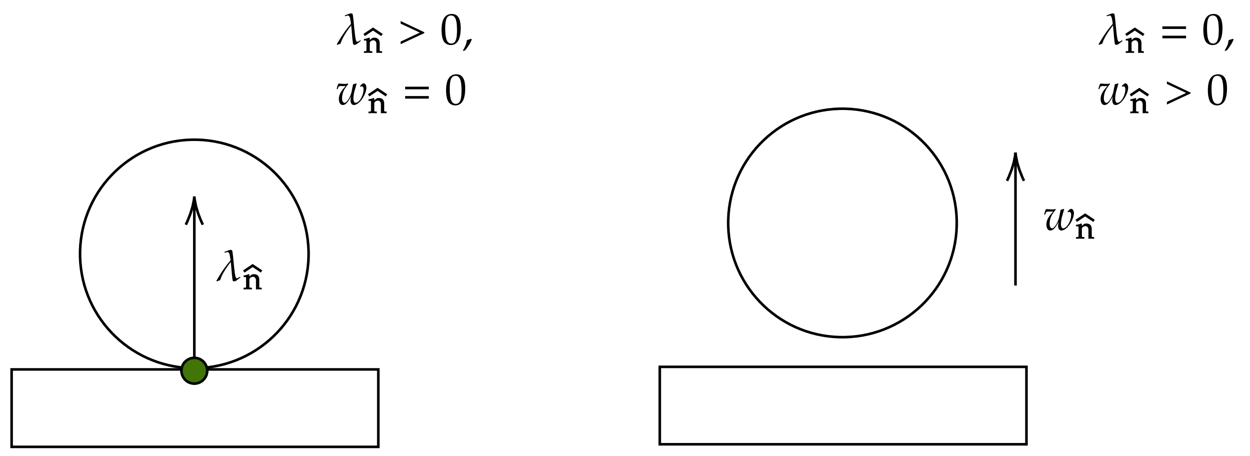

$\lambda_{\hat{\mathbf{n}}} > 0,$
$w_{\hat{\mathbf{n}}} = 0$
$\lambda_{\hat{\mathbf{n}}}$
$\lambda_{\hat{\mathbf{n}}} = 0,$
$w_{\hat{\mathbf{n}}} > 0$
$w_{\hat{\mathbf{n}}}$

Figure 2.5: A normal contact impulse $\lambda_{\hat{\mathbf{n}}}$ is only generated when two bodies are in contact with each other, where the normal relative contact velocity $w_{\hat{\mathbf{n}}}$ is zero.

Constrained equations of motion

Combining Equation 2.27 and Equations 2.35a - 2.35c yields the mixed linear complementarity problem (MLCP) formulation

$$\mathbf{A}\boldsymbol{\lambda}^{+} + \mathbf{b} = \mathbf{w} = \mathbf{w}_{+} + \mathbf{w}_{-}, \tag{2.36a}$$

subject to

$$\begin{cases} \mathbf{w}_{+} \geq \mathbf{0} \quad \perp \quad \boldsymbol{\lambda}^{+} - \boldsymbol{\lambda}^{l} \geq \mathbf{0} \\ \mathbf{w}_{-} \geq \mathbf{0} \quad \perp \quad \boldsymbol{\lambda}^{u} - \boldsymbol{\lambda}^{+} \geq \mathbf{0}. \end{cases} \tag{2.36b}$$

Here, $\mathbf{w}$ is the vector of constraint velocities that is decomposed into the slack velocities $\mathbf{w}_{+}$ and $\mathbf{w}_{-}$, where $\mathbf{w}_{+} = \max(\mathbf{w}, \mathbf{0})$ and $\mathbf{w}_{-} = -\min(\mathbf{w}, \mathbf{0})$. It is the slack velocities that must satisfy the non-negative complementarity condition in Equation 2.36b. The problem is a MLCP because there are lower and upper bounds, $\boldsymbol{\lambda}^{l}$ and $\boldsymbol{\lambda}^{u}$, on the constraint impulses $\boldsymbol{\lambda}^{+}$, and the complementarity conditions $\mathbf{w}_{+}^{T}\left(\boldsymbol{\lambda}^{+} - \boldsymbol{\lambda}^{l}\right) = 0$ and $\mathbf{w}_{-}^{T}\left(\boldsymbol{\lambda}^{u} - \boldsymbol{\lambda}^{+}\right) = 0$ must be satisfied. Since we have two tangent directions and one normal direction for the box friction model, the Jacobian matrix $\mathbf{J}$ can be computed in a similar way compared to when the nonlinear Coulomb friction cone is used (Appendix A.2). The main advantage of using the boxed MLCP formulation is the matrix $\mathbf{A}$ is symmetric positive definite, permitting the use of the Cholesky factorization (Algorithm 2 in Section 2.5.4) on the matrix $\mathbf{A}$ if we use direct methods to solve for Equation 2.36, or the use of iterative solvers such as variants of the preconditioned conjugate gradient method (Chapter 5). Furthermore, since the matrix $\mathbf{A}$ is positive definite, the solution $\boldsymbol{\lambda}^+$ to the MLCP in Equation 2.36b exists and is unique [9].

2.5 Solving the MLCP

There are many works on solving MLCPs for simulating frictional contact [2, 24]. Two of the approaches to solving MLCPs are iterative methods and pivoting methods. Iterative methods such as projected Gauss-Seidel [14], staggered projections [19], and projective dynamics [21], are quite popular. However, convergence can be an issue, and in particular, the projected Gauss-Seidel method is notoriously slow to converge for ill-conditioned problems, even when warm starting is applied. Pivoting methods can therefore be an attractive alternative, and this is likewise the focus of our work.

2.5.1 Pivoting methods

With the conditions 2.35a - 2.35c, pivoting methods use *index sets* to help correctly label constraint variables in the system. Constraint variables are placed into the free set $\mathbb{F}$ or the tight set $\mathbb{T}$, where the tight set $\mathbb{T}$ can be further decomposed into the tight lower set $\mathbb{T}_l$ and tight upper set $\mathbb{T}_u$. The sets are defined as

$$\mathbb{F} := \left\{ i : \boldsymbol{\lambda}_i^l < \boldsymbol{\lambda}_i < \boldsymbol{\lambda}_i^u \text{ and } \mathbf{w}_i = 0 \right\} \tag{2.37a}$$

$$\mathbb{T}_l := \left\{ i : \boldsymbol{\lambda}_i = \boldsymbol{\lambda}_i^l \text{ and } \mathbf{w}_i > 0 \right\} \tag{2.37b}$$

$$\mathbb{T}_u := \left\{ i : \boldsymbol{\lambda}_i = \boldsymbol{\lambda}_i^u \text{ and } \mathbf{w}_i < 0 \right\}, \tag{2.37c}$$

where $\mathbb{T} = \mathbb{T}_l \cup \mathbb{T}_u$. Recall that for friction constraints, the box friction bounds are $\boldsymbol{\lambda}_i^l = -\mu\lambda_{\hat{\mathbf{n}}}$ and $\boldsymbol{\lambda}_i^u = \mu\lambda_{\hat{\mathbf{n}}}$, with $\lambda_{\hat{\mathbf{n}}}$ being the normal impulse. As for non-interpenetration constraints, the bounds are $\boldsymbol{\lambda}_i^l = 0$ and $\boldsymbol{\lambda}_i^u = \infty$, whereas bilateral constraints are boundless such that $\boldsymbol{\lambda}_i^l = -\infty$ and $\boldsymbol{\lambda}_i^u = \infty$. As bilateral constraints have no impulse bounds and always have constraint velocities $\mathbf{w}$ of zero (Section 2.2), variables that correspond to bilateral constraints will always be marked as free. With these sets to label variables, we can partition the linear system in Equation 2.36 in the following way

$$\begin{bmatrix} \mathbf{A}_{\mathbb{F}\mathbb{F}} & \mathbf{A}_{\mathbb{F}\mathbb{T}_l} & \mathbf{A}_{\mathbb{F}\mathbb{T}_u} \\ \mathbf{A}_{\mathbb{T}_l\mathbb{F}} & \mathbf{A}_{\mathbb{T}_l\mathbb{T}_l} & \mathbf{A}_{\mathbb{T}_l\mathbb{T}_u} \\ \mathbf{A}_{\mathbb{T}_u\mathbb{F}} & \mathbf{A}_{\mathbb{T}_u\mathbb{T}_l} & \mathbf{A}_{\mathbb{T}_u\mathbb{T}_u} \end{bmatrix} \begin{bmatrix} \boldsymbol{\lambda}_{\mathbb{F}} \\ \boldsymbol{\lambda}_{\mathbb{T}_l} \\ \boldsymbol{\lambda}_{\mathbb{T}_u} \end{bmatrix} + \begin{bmatrix} \mathbf{b}_{\mathbb{F}} \\ \mathbf{b}_{\mathbb{T}_l} \\ \mathbf{b}_{\mathbb{T}_u} \end{bmatrix} = \begin{bmatrix} \mathbf{w}_{\mathbb{F}} \\ \mathbf{w}_{\mathbb{T}_l} \\ \mathbf{w}_{\mathbb{T}_u} \end{bmatrix}, \tag{2.38}$$

subject to conditions

$$\boldsymbol{\lambda}_{\mathbb{T}_l} < \boldsymbol{\lambda}_{\mathbb{F}} < \boldsymbol{\lambda}_{\mathbb{T}_u} \quad \text{and} \quad \mathbf{w}_{\mathbb{F}} = \mathbf{0} \tag{2.39a}$$

$$\boldsymbol{\lambda}_{\mathbb{T}_l} = \boldsymbol{\lambda}^l \quad \text{and} \quad \mathbf{w}_{\mathbb{T}_l} \geq \mathbf{0} \tag{2.39b}$$

$$\boldsymbol{\lambda}_{\mathbb{T}_u} = \boldsymbol{\lambda}^u \quad \text{and} \quad \mathbf{w}_{\mathbb{T}_u} \leq \mathbf{0}, \tag{2.39c}$$

where the submatrices in Equation 2.38 refer to rows and columns of the lead matrix $\mathbf{A}$ selected by the index sets. For example, $\mathbf{A}_{\mathbb{T}_u\mathbb{F}}$ corresponds to row indices of variables that are in the upper tight set $\mathbb{T}_u$ and column indices of variables that are in the free set $\mathbb{F}$. Since the lower and upper bounds $\boldsymbol{\lambda}^l$ and $\boldsymbol{\lambda}^u$ are known, we only need to solve for $\boldsymbol{\lambda}_{\mathbb{F}}$. Hence, we can reduce the linear system in Equation 2.38 to

$$\mathbf{A}_{\mathbb{F}\mathbb{F}}\boldsymbol{\lambda}_{\mathbb{F}} = -\mathbf{b}_{\mathbb{F}} - \mathbf{A}_{\mathbb{F}\mathbb{T}_l}\boldsymbol{\lambda}_{\mathbb{T}_l} - \mathbf{A}_{\mathbb{F}\mathbb{T}_u}\boldsymbol{\lambda}_{\mathbb{T}_u}, \tag{2.40}$$

where the principal submatrix $\mathbf{A}_{\mathbb{F}\mathbb{F}}$ is symmetric positive definite if the box friction model is used. This means if we are using direct methods to solve Equation 2.40, one can apply Cholesky factorization on the submatrix $\mathbf{A}_{\mathbb{F}\mathbb{F}}$, where $\mathbf{A}_{\mathbb{F}\mathbb{F}} = \mathbf{L}\mathbf{L}^T$, with $\mathbf{L}$ being a lower triangular matrix. After factorizing the matrix $\mathbf{A}$, we apply forward and backward substitution to solve Equation 2.40 for $\boldsymbol{\lambda}_{\mathbb{F}}$.

The essence of pivoting methods is to correctly determine the index set. For a system with m variables, if we partition variables into free and tight, there are 2^m possible index sets, while partitioning variables into free, lower tight, and upper tight will give 3^m possible index sets. In the next section, we are going to go over popular techniques to efficiently determine the correct index set without brute forcing through a large number of possible index sets.

2.5.2 Principal pivoting method

We start with an initial guess of the correct index set. A common choice is to label every variable as free. One can also use the free and tight sets from the previous time step, in which we set the tight entries of $\boldsymbol{\lambda}$ to the bounds $\boldsymbol{\lambda}^l$ and $\boldsymbol{\lambda}^u$.

With a guess of the index set, we solve Equation 2.40 for $\boldsymbol{\lambda}_{\mathbb{F}}$. After the linear solve, we compute the constraint velocities $\mathbf{w} = \mathbf{A}\boldsymbol{\lambda} + \mathbf{b}$ and then check whether variables in $\boldsymbol{\lambda}_{\mathbb{F}}$ satisfy Equations 2.39a - 2.39c. The pivoting algorithm is outlined below:

$$\mathbb{F} \leftarrow \mathbb{F} - \{i\} \text{ and } \mathbb{T}_l \leftarrow \mathbb{T}_l \cup \{i\}, \quad \text{if } \boldsymbol{\lambda}_i \leq \boldsymbol{\lambda}_i^l \text{ and } \mathbf{w}_i = 0, \tag{2.41}$$

$$\mathbb{F} \leftarrow \mathbb{F} - \{i\} \text{ and } \mathbb{T}_u \leftarrow \mathbb{T}_u \cup \{i\}, \quad \text{if } \boldsymbol{\lambda}_i \geq \boldsymbol{\lambda}_i^u \text{ and } \mathbf{w}_i = 0, \tag{2.42}$$

$$\mathbb{T}_l \leftarrow \mathbb{T}_l - \{i\} \text{ and } \mathbb{F} \leftarrow \mathbb{F} \cup \{i\}, \quad \text{if } \boldsymbol{\lambda}_i = \boldsymbol{\lambda}_i^l \text{ and } \mathbf{w}_i < 0, \tag{2.43}$$

$$\mathbb{T}_u \leftarrow \mathbb{T}_u - \{i\} \text{ and } \mathbb{F} \leftarrow \mathbb{F} \cup \{i\}, \quad \text{if } \boldsymbol{\lambda}_i = \boldsymbol{\lambda}_i^u \text{ and } \mathbf{w}_i > 0. \tag{2.44}$$

The first two cases 2.41 and 2.42 concerns *infeasible* free variables in the constraint impulse $\boldsymbol{\lambda}$ that exceeds the impulse bounds after the linear solve. We clamp these infeasible variables to their bounds and classify those variables as tight. Cases 2.43 and 2.44 refer to variables in the tight set that are not physical such that the constraint velocities $\mathbf{w}$ and the friction impulses $\boldsymbol{\lambda}$ are acting in the same direction. In other words, the *complementary* conditions are violated. In this case, we move these variables from the tight set back into the free set for subsequent linear solves to correct. Note that the principal pivoting method only pivots one variable at a time. So if there are multiple infeasible or non-complementary variables, we pick and pivot the smallest index. Another important thing to note is for matrices $\mathbf{A}_{\mathbb{F}\mathbb{F}}$ that are not ill-conditioned, the principal pivoting method is guaranteed to determine the correct index set in m pivots if the matrix $\mathbf{A}_{\mathbb{F}\mathbb{F}}$ is symmetric positive definite, where m denotes the number of constraints in the system [2]. However, for cases where the matrix $\mathbf{A}_{\mathbb{F}\mathbb{F}}$ is ill-conditioned, we do not have such guarantees and we might cycle back to an index set we have encountered in previous pivots, preventing progress. In such situations, we simply return the best solution we have seen so far, which is the index set with the least number of variables that are infeasible or not complementary.

While this method is effective for linear systems with a symmetric positive definite matrix, it is costly in the sense that we have to solve a linear system at each pivot. Hence, in the next section, we describe an extension of the principal pivoting method that pivots multiple infeasible variables and non-complementary variables simultaneously.

2.5.3 Block principal pivoting method

We outline the block principal pivoting (BPP) method in Algorithm 1 [18]. We see in lines 14 - 16 that the BPP algorithm pivots all infeasible and non-complementary variables in each iteration. Pivoting multiple variables at a time speeds up convergence and reduces the number of times we have to solve the linear system in Equation 2.40. However, the problem with BPP is it can cycle through the same index set even if the matrix $\mathbf{A}_{\mathbb{F}\mathbb{F}}$ is symmetric positive definite. To fix this, we have to implement robust cycle detection and prevention schemes to allow BPP to converge to a solution.

Cycle detection and cycle breaking

To break the cycle on the same index set, we first recover the index set to the state before pivoting, and then switch from block pivoting mode to single pivoting mode. How long

Algorithm 1 Block principal pivoting method

```
1: procedure BPP($\mathbf{A}$, $\mathbf{b}$, $\boldsymbol{\lambda}$, $\boldsymbol{\lambda}^l$, $\boldsymbol{\lambda}^u$, $\mathbb{F}$, $\mathbb{T}_\mathbf{l}$, $\mathbb{T}_\mathbf{u}$)
2:     Initialize $k = 0$
3:     while $\mathbb{F}$, $\mathbb{T}_l$ and $\mathbb{T}_u$ changes or $k <$ max iteration do
4:         for each $i \in \mathbb{T}_\mathbf{l}$ do                              ▷ Clamp infeasible variables
5:             $\boldsymbol{\lambda}_i \leftarrow \boldsymbol{\lambda}_i^l$
6:         end for
7:         for each $i \in \mathbb{T}_\mathbf{u}$ do                              ▷ Clamp infeasible variables
8:             $\boldsymbol{\lambda}_i \leftarrow \boldsymbol{\lambda}_i^u$
9:         end for
10:        Perform Cholesky factorization $\mathbf{A}_{\mathbb{F}\mathbb{F}} = \mathbf{L}\mathbf{L}^T$.
11:        Solve $\mathbf{L}\mathbf{y} = -\mathbf{b}_\mathbb{F} - \mathbf{A}_{\mathbb{F}\mathbb{T}_l}\boldsymbol{\lambda}_{\mathbb{T}_l} - \mathbf{A}_{\mathbb{F}\mathbb{T}_u}\boldsymbol{\lambda}_{\mathbb{T}_u}$      ▷ Forward substitution
12:        Solve $\mathbf{L}^T\boldsymbol{\lambda} = \mathbf{y}$                                   ▷ Backward substitution
13:        $\mathbf{w} = \mathbf{A}\boldsymbol{\lambda} + \mathbf{b}$
14:        for $i = 1$ to $m$ do                                   ▷ Pivot multiple variables
15:            Pivot index $i$ in $\mathbb{F}$, $\mathbb{T}_l$, or $\mathbb{T}_u$ using Equations 2.41 - 2.44.
16:        end for
17:        $k = k + 1$
18:    end while
19: end procedure
```

should the method remain in single pivoting mode is up to the user. One option is the user can run the single pivoting mode for a fixed number of iterations before switching back to block pivoting mode. Another option is to switch back to block pivoting mode when the method makes good progress and finds a solution that has the least number of infeasible and non-complementary variables compared to all the solutions it has found so far. For simplicity, Algorithm 1 omits cycle detection and the single pivoting fail-safe.

Even with cycle detection and prevention schemes, it is still possible for BPP to encounter cycles even after switching from block to single pivoting mode, or after cycle breaking has been attempted on that cycling index set. This occurs on complicated problems with a lot of redundant constraints. In these failure cases, we simply return the best solution we have seen so far.

2.5.4 Cholesky factorization

In order to compute an exact solution to the linear system with m constraints

$$\mathbf{A}\boldsymbol{\lambda} = \mathbf{b}, \tag{2.45}$$

where $\mathbf{A} \in \mathbb{R}^{m\times m}$ and $\boldsymbol{\lambda}, \mathbf{b} \in \mathbb{R}^m$, we have to factorize the matrix $\mathbf{A}$ in order to apply forward substitution (Algorithm 1, line 11) and backward substitution (Algorithm 1, line 12) to get the solution $\mathbf{x}$. However, if $\mathbf{A}$ is symmetric positive definite, then we can apply the Cholesky factorization $\mathbf{A} = \mathbf{L}\mathbf{L}^T$ [32], where $\mathbf{L}$ is lower triangular, such that we only need to compute and store one matrix $\mathbf{L}$. As we are using the box friction model, our system matrix $\mathbf{A}$ is symmetric positive definite, and so the Cholesky factorization is applicable to our problems. We outline the Cholesky factorization method in Algorithm 2 [32]. It is important to note that the diagonal elements of $\mathbf{L}$ remain positive during the Cholesky factorization steps [31]. Since our matrix $\mathbf{A}$ is sparse, some operations are skipped during the factorization. For example, line 9 will be skipped if $a_{ik} = 0$ or $a_{jk} = 0$.

The Cholesky factorization can introduce fill-ins in the factor $\mathbf{L}$ (Algorithm 2, lines 9 and 11). The computational complexity of the sparse Cholesky factorization is therefore $O(|\mathbf{L}|)$ [11], where $|\mathbf{L}|$ is the number of nonzeros in the Cholesky factor $\mathbf{L}$. To reduce the number of fill-ins introduced, one can apply ordering strategies such as Cuthill-McKee [10], reverse Cuthill-McKee [5], or approximate minimum degree [1]. We choose the Cuthill-McKee ordering approach for our problems [12,26].

2.5.5 Limitations of the principal pivoting method

While pivoting methods such as the BPP method are efficient in determining the solution $\boldsymbol{\lambda}$ that satisfies Equations 2.39a - 2.39c by pivoting multiple variables per iteration, the Cholesky factorization of $\mathbf{A}_{\mathbb{F}\mathbb{F}}$ does not scale to large problems due to the increasing ratio of fill-ins as the matrix size increases (Section 2.5.4). So, in chapters 3 - 5, we are going to

Algorithm 2 Cholesky factorization

```
1:  procedure CHOLESKY($\mathbf{A}$)
2:      for $j = 1 : m$ do                                        ▷ Iterate over columns
3:          for $k = 1 : j - 1$ do
4:              $a_{jj} = a_{jj} - a_{jk}a_{jk}$                  ▷ Update diagonal using previous columns
5:          end for
6:          $a_{jj} = \sqrt{a_{jj}}$
7:          for $i = j + 1 : m$ do                                ▷ Update elements below diagonal
8:              for $k = 1 : j - 1$ do
9:                  $a_{ij} = a_{ij} - a_{ik}a_{jk}$              ▷ Update using previous columns
10:             end for
11:             $a_{ij} = a_{ij}/a_{jj}$                          ▷ Update using diagonal
12:         end for
13:     end for
14:     $\mathbf{L} = \text{tril}(\mathbf{A})$                    ▷ Lower triangular part of $\mathbf{A}$
15: end procedure
```

discuss about the preconditioned conjugate gradient method and how to use it to solve large frictional contact problems.

Chapter 3

Conjugate Gradient Method

As described in the previous chapter, the computational cost in obtaining an exact solution to the linear system $\mathbf{A}\boldsymbol{\lambda} = \mathbf{b}$ increases as the problem size gets larger due to the Cholesky factorization becoming more expensive. In most cases, we only need a good approximate solution to the linear system. Since the matrix of the linear system is symmetric positive definite (SPD), we can apply the conjugate gradient (CG) method [16] to find a good approximate solution to the linear system. In this chapter, we will describe the CG method and also briefly look at the convergence theory of CG.

The CG method is outlined in Algorithm 3. Solving the linear system $\mathbf{A}\boldsymbol{\lambda} = \mathbf{b}$ is the same as finding the solution to the minimization problem

$$\min_{\boldsymbol{\lambda}} \underbrace{\frac{1}{2}\boldsymbol{\lambda}^T\mathbf{A}\boldsymbol{\lambda} - \boldsymbol{\lambda}^T\mathbf{b}}_{F(\boldsymbol{\lambda})}. \tag{3.1}$$

In our case, since the matrix $\mathbf{A}$ is SPD, $F(\boldsymbol{\lambda})$ is strictly convex, which means there is a global minimum to Equation 3.1 that can be determined by solving the linear system $\mathbf{A}\boldsymbol{\lambda} = \mathbf{b}$. To iteratively step towards the global minimum in the optimization process, at the k-th iteration, we pick a search direction $\mathbf{q}^{(k)}$ and update the approximate solution with:

$$\boldsymbol{\lambda}^{(k+1)} = \boldsymbol{\lambda}^{(k)} + \alpha^{(k)}\mathbf{q}^{(k)}. \tag{3.2}$$

The step size $\alpha^{(k)}$ at the k-th iteration of the CG method is computed using:

$$\alpha^{(k)} = \frac{\left(\mathbf{r}^{(k)}, \mathbf{r}^{(k)}\right)}{\left(\mathbf{q}^{(k)}, \mathbf{A}\mathbf{q}^{(k)}\right)}, \tag{3.3}$$

where $(\cdot, \cdot)$ denotes the inner product (Algorithm 3, line 12). The residual $\mathbf{r}^{(k)}$ (Algorithm 3, line 4) measures how close we are to the vector $\mathbf{b}$. If the norm of the residual $\|\mathbf{r}^{(k)}\|$ is small, then we are confident we have obtained a good approximate solution to the linear system $\mathbf{A}\boldsymbol{\lambda} = \mathbf{b}$. It is important to note that the step size is chosen such that it minimizes $F(\boldsymbol{\lambda}^{(\boldsymbol{k}+\mathbf{1})})$.

The essence of CG is its choice of the search direction $\mathbf{q}$. CG applies Gram-Schmidt conjugation to construct a set of search directions $\{\mathbf{q}^{(0)}, \mathbf{q}^{(1)}, ..., \mathbf{q}^{(k)}\}$ such that the search direction at the $k+1$-th iteration $\mathbf{q}^{(k+1)}$ is A-orthogonal to the previous search directions,

$$(\mathbf{q}^{(k)})^T \mathbf{A}\mathbf{q}^{(l)} = 0, \quad k \neq l. \tag{3.4}$$

The subspace that the search directions span is called the Krylov subspace. The theories of Krylov subspaces allows us to only need the search direction from the previous iteration $\mathbf{q}^{(k-1)}$ to update the search direction at the current iteration $\mathbf{q}^{(k)}$ (Algorithm 3, line 10), where:

$$\beta^{(k)} = \frac{\left(\mathbf{r}^{(k)}, \mathbf{r}^{(k)}\right)}{\left(\mathbf{r}^{(k-1)}, \mathbf{r}^{(k-1)}\right)}. \tag{3.5}$$

The theories of Krylov subspace also permits a slightly cheaper update of the residual (Algorithm 3, line 14), where the matrix-vector product $\mathbf{A}\mathbf{q}^{(k)}$ used to compute the step size $\alpha^{(k)}$ can be reused to update the residual.

The CG algorithm requires one matrix-vector multiplication and $10m$ FLOPs per iteration, and we only need to store and update the four vectors $(\boldsymbol{\lambda}, \mathbf{q}, \mathbf{r}, \mathbf{A}\mathbf{q})$ during the CG iterations [16]. Here, m denotes the number of variables in our linear system.

Algorithm 3 Conjugate gradient

```
1: procedure CG($\mathbf{A}, \mathbf{b}$)
2:     $k = 0$
3:     Initialize $\boldsymbol{\lambda}^{(k)} = \mathbf{0}$
4:     Initialize $\mathbf{r}^{(k)} = \mathbf{b} - \mathbf{A}\boldsymbol{\lambda}^{(k)}$          ▷ Residual
5:     do
6:         if $k = 0$ then
7:             Set $\mathbf{q}^{(k)} = \mathbf{r}^{(k)}$          ▷ Initialize search direction
8:         else
9:             Compute $\beta^{(k)}$          ▷ Equation 3.5
10:            $\mathbf{q}^{(k)} = \mathbf{r}^{(k)} + \beta^{(k)}\mathbf{q}^{(k-1)}$          ▷ Update search direction
11:        end if
12:        Compute $\alpha^{(k)}$          ▷ Equation 3.3
13:        $\boldsymbol{\lambda}^{(k+1)} = \boldsymbol{\lambda}^{(k)} + \alpha^{(k)}\mathbf{q}^{(k)}$          ▷ Update solution
14:        $\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha^{(k)}\mathbf{A}\mathbf{q}^{(k)}$          ▷ Update residual
15:        $k = k + 1$
16:    while ($k <$ max iteration or $\|\mathbf{r}^{(k)}\| > \varepsilon$)
17: end procedure
```

3.1 Convergence theory of CG

Theoretically, for a $m \times m$ symmetric positive definite matrix $\mathbf{A}$ with full rank, CG converges in at most m steps due to having at most m A-orthogonal search directions in $\mathbb{R}^m$ [16]. But, due to roundoff errors, this is not guaranteed in practice. Also, for linear systems where m is large, this convergence guarantee is not meaningful. Fortunately, there is a theorem that states the convergence rate of CG is dependent on the condition number of the matrix, $\mathbf{A}$ $\kappa(\mathbf{A})$, and is bounded by [16]

$$\left\|\boldsymbol{\lambda} - \boldsymbol{\lambda}^{(k)}\right\|_A \leq 2\left\|\boldsymbol{\lambda} - \boldsymbol{\lambda}^{(0)}\right\|_A \left(\frac{\sqrt{\kappa(\mathbf{A})} - 1}{\sqrt{\kappa(\mathbf{A})} + 1}\right)^k, \tag{3.6}$$

where $\{\boldsymbol{\lambda}^{(k)}\}$ is the sequence of iterates produced by CG, $\boldsymbol{\lambda}$ is the exact solution, and $\|\cdot\|_{\mathbf{A}}$ is the A-norm defined by $\|\mathbf{x}\|_A = \sqrt{\mathbf{x}^T\mathbf{A}\mathbf{x}}$ for vector $\mathbf{x} \in \mathbb{R}^m$ [16]. The condition number of the matrix $\mathbf{A}$ (with respect to the Euclidean norm) is given by $\kappa(\mathbf{A}) = \|\mathbf{A}\|_2 \|\mathbf{A}^{-1}\|_2$ [16]. It is important to note that Equation 3.6 gives the worst bound for the convergence of CG, and we can get a better bound if the eigenvalues of the matrix $\mathbf{A}$ are clustered [32].

Due to our problems being very ill-conditioned, where $\kappa(\mathbf{A})$ is large, CG will converge very slowly for our problems. In the next chapter, we will discuss different *preconditioning* techniques to construct the linear system $\widetilde{\mathbf{A}}\widetilde{\boldsymbol{\lambda}} = \widetilde{\mathbf{b}}$ from $\mathbf{A}\boldsymbol{\lambda} = \mathbf{b}$ such that $\kappa(\widetilde{\mathbf{A}}) \ll \kappa(\mathbf{A})$ and $\kappa(\widetilde{\mathbf{A}}) \approx 1$. The importance of constructing the linear system $\widetilde{\mathbf{A}}\widetilde{\boldsymbol{\lambda}} = \widetilde{\mathbf{b}}$ is CG will converge much faster when solving this linear system due to the condition number of $\widetilde{\mathbf{A}}$ being much smaller than $\mathbf{A}$.

Chapter 4

Preconditioning

Preconditioning is essential for speeding up the convergence of iterative solvers such as conjugate gradient (CG). It is a technique that is used to improve the condition number of a matrix. Suppose we have a symmetric positive definite (SPD) matrix $\mathbf{P}$ that approximates the matrix $\mathbf{A}_{\mathbb{F}\mathbb{F}}$ and is easy to invert. We want to solve for this modified linear system instead

$$\mathbf{P}^{-1}\mathbf{A}_{\mathbb{F}\mathbb{F}}\boldsymbol{\lambda} = \mathbf{P}^{-1}\mathbf{b}. \tag{4.1}$$

The reason why we consider solving this modified linear system is if $\mathbf{P}$ approximates $\mathbf{A}_{\mathbb{F}\mathbb{F}}$, then the condition number $\kappa(\mathbf{P}^{-1}\mathbf{A}_{\mathbb{F}\mathbb{F}}) \approx \kappa(\mathbf{I}) = 1$. Note that if $\mathbf{P}$ is SPD, its inverse $\mathbf{P}^{-1}$ is also SPD. However, despite the matrix $\mathbf{A}_{\mathbb{F}\mathbb{F}}$ and $\mathbf{P}^{-1}$ being SPD, the matrix $\mathbf{P}^{-1}\mathbf{A}_{\mathbb{F}\mathbb{F}}$ might not be SPD, which will be problematic for CG.

To make sure the matrix of the modified linear system is SPD, we use the fact that the SPD matrix $\mathbf{P}$ has a Cholesky factorization $\mathbf{P} = \mathbf{L}\mathbf{L}^T$, where $\mathbf{L}$ is lower triangular, and then transform the problem into

$$\underbrace{\mathbf{L}^{-1}\mathbf{A}_{\mathbb{F}\mathbb{F}}\mathbf{L}^{-T}}_{\widetilde{\mathbf{A}}}\underbrace{\mathbf{L}^{T}\boldsymbol{\lambda}}_{\widetilde{\boldsymbol{\lambda}}} = \underbrace{\mathbf{L}^{-1}\mathbf{b}}_{\widetilde{\mathbf{b}}}, \tag{4.2}$$

in order to solve a linear system where the matrix $\tilde{\mathbf{A}}$ is SPD (because $\mathbf{L}^{-1}$ has full column rank) [29].

There are many choices for the preconditioning matrix $\mathbf{P}$. The simplest choice is the Jacobi preconditioner, where $\mathbf{P}$ is simply the diagonals of the matrix $\mathbf{A}_{\mathbb{FF}}$. However, the Jacobi preconditioner does not provide much improvement in the convergence rate of CG [29]. In order to speed up the convergence rate of CG, we will need a more effective preconditioner.

4.1 Incomplete Cholesky factorization

A better preconditioning choice will be the incomplete Cholesky preconditioner [16], where $\mathbf{P} = \widehat{\mathbf{L}}\widehat{\mathbf{L}}^T$, with the matrix $\widehat{\mathbf{L}}$ being lower triangular. The incomplete Cholesky preconditioning technique aims to reduce the computation and memory cost of the full Cholesky factorization while getting a reasonable approximation of the matrix $\mathbf{A}_{\mathbb{FF}}$. There are different variants of the incomplete Cholesky preconditioner. We are going to first discuss the different existing variants and then propose our own variant of such a preconditioner.

4.1.1 Zero-fill incomplete Cholesky

The first variant we will introduce is incomplete Cholesky with zero-fill [16], IC(0), where all steps that introduce fill-ins are ignored. It is commonly used due to it being computationally and memory efficient. The zero-fill incomplete Cholesky factorization is summarized in Algorithm 4. The result is a matrix $\widehat{\mathbf{L}}$ that has the same sparsity pattern as the lower triangular part of the matrix $\mathbf{A}_{\mathbb{FF}}$. As matrix $\mathbf{A}_{\mathbb{FF}}$ is sparse, the incomplete Cholesky factor $\widehat{\mathbf{L}}$ is cheap to compute. However, a problem with IC(0) is that the remaining matrix after a factorization step may consist of negative diagonal elements. This is because elements (that would introduce fill-ins) are dropped during the factorization step. As the algorithm takes the square root of the diagonal elements

Algorithm 4 Zero-fill incomplete Chokesky IC(0)

```
procedure IC0($\mathbf{A}_{\mathbb{FF}}$)
    Initialize $\widehat{\mathbf{L}} = \text{tril}(\mathbf{A}_{\mathbb{FF}})$.    ▷ Lower triangular part of $\mathbf{A}_{\mathbb{FF}}$.
    for each $i \in \mathbb{F}$ do    ▷ Iterate over columns
        $\widehat{\mathbf{L}}_{ii} = \sqrt{\widehat{\mathbf{L}}_{ii}}$
        for each $j \in \mathbb{F}$ following $i$ do
            if $\widehat{\mathbf{L}}_{ji} \neq 0$ then    ▷ Skip fill-ins
                $\widehat{\mathbf{L}}_{ji} = \widehat{\mathbf{L}}_{ji}/\widehat{\mathbf{L}}_{ii}$
            end if
        end for
        for each $j \in \mathbb{F}$ following $i$ do    ▷ Update lower right block
            for each $k \in \mathbb{F}$ following $i$ do
                if $\widehat{\mathbf{L}}_{kj} \neq 0$ then    ▷ Skip fill-ins
                    $\widehat{\mathbf{L}}_{kj} = \widehat{\mathbf{L}}_{kj} - (\widehat{\mathbf{L}}_{ki}\widehat{\mathbf{L}}_{ji})$    ▷ Update step
                end if
            end for
        end for
    end for
end procedure
```

of the matrix (Algorithm 4, line 4), this will potentially cause the factorization to break down due to taking the square root of a negative diagonal element.

4.1.2 Incomplete Cholesky with diagonal compensation

One way to resolve breakdowns is to instead compute the incomplete Cholesky factorization of the diagonally shifted matrix $\mathbf{A}_{\mathbb{FF}} + \gamma\mathbf{I}$, where $\gamma > 0$ [27]. If the factorization of $\mathbf{A}_{\mathbb{FF}}$ fails, we keep increasing the shift γ applied to the matrix $\mathbf{A}_{\mathbb{FF}}$ until the factorization succeeds. The problem with this approach is the shift γ can be very large, which drastically reduces the effectiveness of the preconditioner. Note that rather than applying a shift of $\gamma\mathbf{I}$, one can instead apply a shift of $\gamma\text{diag}(\mathbf{A}_{\mathbb{FF}})$ [23].

4.1.3 Incomplete Cholesky with drop tolerance

Another approach to resolving breakdowns is to keep more fill-ins and only drop elements that fall below a computed drop tolerance. We describe this approach using

Algorithm 5 Incomplete Cholesky with drop tolerance ICT

```
 1: procedure ICT($\mathbf{A}_{\mathbb{FF}}$, $\epsilon$)
 2:     Initialize $\widehat{\mathbf{L}} = \text{tril}(\mathbf{A}_{\mathbb{FF}})$.        ▷ Lower triangular part of $\mathbf{A}_{\mathbb{FF}}$.
 3:     Compute $\boldsymbol{\tau}$ using $\epsilon$.        ▷ Drop tolerance for each column.
 4:     for each $i \in \mathbb{F}$ do
 5:         for each $j \in \mathbb{F}$ following $i$ do
 6:             if $\widehat{\mathbf{L}}_{ji} < \boldsymbol{\tau}_i$ then
 7:                 $\widehat{\mathbf{L}}_{ji} = 0$.        ▷ Drop element
 8:             end if
 9:         end for
10:         $\widehat{\mathbf{L}}_{ii} = \sqrt{\widehat{\mathbf{L}}_{ii}}$
11:         for each $j \in \mathbb{F}$ following $i$ do
12:             $\widehat{\mathbf{L}}_{ji} = \widehat{\mathbf{L}}_{ji} / \widehat{\mathbf{L}}_{ii}$
13:             if $\widehat{\mathbf{L}}_{ji} < \boldsymbol{\tau}_i$ then
14:                 $\widehat{\mathbf{L}}_{ji} = 0$.        ▷ Drop element
15:             end if
16:         end for
17:         Keep $p$ largest element of column $i$.
18:         for each $j \in \mathbb{F}$ following $i$ do        ▷ Update lower right block
19:             for each $k \in \mathbb{F}$ following $i$ do
20:                 $\widehat{\mathbf{L}}_{kj} = \widehat{\mathbf{L}}_{kj} - (\widehat{\mathbf{L}}_{ki}\widehat{\mathbf{L}}_{ji})$        ▷ Update step
21:                 if $\widehat{\mathbf{L}}_{kj} < \boldsymbol{\tau}_j$ then
22:                     $\widehat{\mathbf{L}}_{kj} = 0$.        ▷ Drop element
23:                 end if
24:             end for
25:         end for
26:     end for
27: end procedure
```

Algorithm 5 [27]. We compute a drop tolerance $\boldsymbol{\tau}_i$ for each column i using a user-defined parameter $\epsilon > 0$ and the norm of column i of the lower triangular part of the matrix $\mathbf{A}_{\mathbb{FF}}$, $\|\mathbf{a}_i\|$, where $\boldsymbol{\tau}_i = \epsilon\|\mathbf{a}_i\|$ (Algorithm 5, line 3).

In order to control the number of fill-ins and reduce the computational cost in subsequent factorization steps, after updating an entry in a column of the incomplete Cholesky factor $\widehat{\mathbf{L}}$, we check and see whether we should discard that entry by comparing against the drop tolerance of that column (Algorithm 5, lines 13 - 15 and lines 21 - 23). To further reduce memory cost, we also drop elements before updating a column (Algorithm 5, lines 6 - 8). Moreover, after fully updating a column, one can only keep the p largest elements and the diagonal element in that column, where p is a user-defined parameter. Note that with any of these dropping schemes, it is possible for the original nonzero entries of the matrix $\mathbf{A}_{\mathbb{F}\mathbb{F}}$ to be dropped.

While this method helps resolve breakdowns and in general computes a more effective preconditioner compared to IC(0) due to having more fill-ins, the factorization cost increases as we decrease our drop tolerance (by choosing a smaller ϵ). Unfortunately, for the matrices of our problems, we need to choose a very small ϵ in order to avoid breakdowns. Although this method will give a very effective preconditioner, the computational cost of the preconditioner approaches the computational cost of the full Cholesky factorization, which defeats the purpose of using an iterative solver. One possible way to mitigate this issue is to apply this method to the shifted matrix $\mathbf{A}_{\mathbb{F}\mathbb{F}} + \gamma\mathbf{I}$ (or $\mathbf{A}_{\mathbb{F}\mathbb{F}} + \gamma\text{diag}(\mathbf{A}_{\mathbb{F}\mathbb{F}})$) as described in the previous section. Scott and Tuma [28] describe an incomplete Cholesky preconditioner where one can control both the density of the incomplete Cholesky factor, the amount of diagonal shift added to the system matrix if a negative pivot occurs, and the amount of memory used. Despite being able to pick a larger ϵ, we, however, may still have to pay the cost of refactoring the whole matrix again if our choice of γ does not avoid breakdowns.

4.1.4 Incomplete Cholesky with partial shift

Since re-computing the whole factorization because of breakdowns is costly, Chen et al. [6] propose a partial shift strategy that only shifts the diagonal elements of the columns that affect the column with the negative diagonal element during the IC(0) factorization step, where the diagonals of those columns are incrementally shifted by $2^{p_i}\gamma$, with p_i being the number of times we shift the column i and γ is chosen to be 1e-4. Although this method does reduce the re-computation cost of the zero-fill incomplete Cholesky factorization, due to how often breakdowns occur in our matrices, we still have to re-factorize many times and apply large diagonal shifts to the problematic columns.

Algorithm 6 IC(0) with imaginary columns $\boldsymbol{\eta}$

```
1: procedure ImaginaryIC0($\mathbf{A}_{\mathbb{FF}}$)
2:     Initialize $\widehat{\mathbf{L}} = \text{tril}(\mathbf{A}_{\mathbb{FF}})$.        ▷ Lower triangular part of $\mathbf{A}_{\mathbb{FF}}$.
3:     Initialize $\boldsymbol{\eta} = \mathbf{1}$        ▷ Initialize all columns to be real.
4:     for each $i \in \mathbb{F}$ do
5:         if $\widehat{\mathbf{L}}_{ii} < 0$ then        ▷ If diagonal is negative
6:             $\boldsymbol{\eta}_i = -1$        ▷ Mark column $i$ imaginary
7:         end if
8:         $\widehat{\mathbf{L}}_{ii} = \sqrt{\boldsymbol{\eta}_i \widehat{\mathbf{L}}_{ii}}$
9:         for each $j \in \mathbb{F}$ following $i$ do
10:            if $\widehat{\mathbf{L}}_{ji} \neq 0$ then        ▷ Skip fill-ins
11:                $\widehat{\mathbf{L}}_{ji} = \widehat{\mathbf{L}}_{ji} / \widehat{\mathbf{L}}_{ii}$
12:                $\widehat{\mathbf{L}}_{ji} = \boldsymbol{\eta}_i \widehat{\mathbf{L}}_{ji}$        ▷ Complex conjugate multiply
13:            end if
14:        end for
15:        for each $j \in \mathbb{F}$ following $i$ do        ▷ Update lower right block
16:            for each $k \in \mathbb{F}$ following $i$ do
17:                if $\widehat{\mathbf{L}}_{kj} \neq 0$ then        ▷ Skip fill-ins
18:                    $\widehat{\mathbf{L}}_{kj} = \widehat{\mathbf{L}}_{kj} - (\boldsymbol{\eta}_i \widehat{\mathbf{L}}_{ki} \widehat{\mathbf{L}}_{ji})$        ▷ Update step
19:                end if
20:            end for
21:        end for
22:    end for
23: end procedure
```

4.1.5 Zero-fill incomplete Cholesky with imaginary entries

To prevent the factorization from breaking down while minimizing the cost of the factorization, we develop a method that allows imaginary entries in the incomplete Cholesky factor. The main idea is to make the best of a bad situation. Instead of restarting the incomplete factorization with a diagonal compensation or drop tolerance, we do not discard the work we have done and complete the factorization while making note of the columns that become imaginary. Rather than explicitly storing the imaginary entries of the incomplete Cholesky factor, we only store the *magnitude* of the imaginary numbers. This is due to the fact that there can only be purely imaginary entries in the incomplete Cholesky factor. We will use $\widehat{\mathbf{L}}$ to denote our implementation of the imaginary incomplete Cholesky factor and use $\widetilde{\mathbf{L}}$ to denote the theoretical incomplete Cholesky factor with

$$\overset{\text{Input data}}{\underset{\widehat{\mathbf{L}}}{\begin{bmatrix} 3.5 & & \\ 2.4 & -4 & \\ & 2 & 9.3 \end{bmatrix}} \quad \underset{\eta}{\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}}} \longrightarrow \overset{\text{Square root of negative diagonal}}{\underset{\widehat{\mathbf{L}}}{\begin{bmatrix} 3.5 & & \\ 2.4 & \sqrt{-4\eta_2} & \\ & 2 & 9.3 \end{bmatrix}} \quad \underset{\eta}{\begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix}}} \longrightarrow \overset{\text{Complex division}}{\underset{\widehat{\mathbf{L}}}{\begin{bmatrix} 3.5 & & \\ 2.4 & 2 & \\ & 1\eta_2 & 9.3 \end{bmatrix}} \quad \underset{\eta}{\begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix}}} \longrightarrow \overset{\text{Output data}}{\underset{\widehat{\mathbf{L}}}{\begin{bmatrix} 3.5 & & \\ 2.4 & 2 & \\ & -1 & 9.3 \end{bmatrix}} \quad \underset{\eta}{\begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix}}}$$

$$\underset{\widetilde{\mathbf{L}}}{\begin{bmatrix} 3.5 & & \\ 2.4 & -4 & \\ & 2 & 9.3 \end{bmatrix}} \longrightarrow \underset{\widetilde{\mathbf{L}}}{\begin{bmatrix} 3.5 & & \\ 2.4 & \sqrt{-4} & \\ & 2 & 9.3 \end{bmatrix}} \longrightarrow \underset{\widetilde{\mathbf{L}}}{\begin{bmatrix} 3.5 & & \\ 2.4 & 2i & \\ & -i & 9.3 \end{bmatrix}} \longrightarrow \underset{\widetilde{\mathbf{L}}}{\begin{bmatrix} 3.5 & & \\ 2.4 & 2i & \\ & -i & 9.3 \end{bmatrix}}$$

Figure 4.1: Since the imaginary incomplete Cholesky factor $\widetilde{\mathbf{L}}$ can only have purely imaginary entries (bottom), we only store the magnitude of the purely imaginary numbers when computing our factor $\widehat{\mathbf{L}}$ with Algorithm 6 (top). As column 2 only contains purely imaginary entries, we set $\boldsymbol{\eta}_2 = -1$ to mark column 2 imaginary.

actual imaginary entries. Since we only store the magnitude of the purely imaginary numbers in $\widetilde{\mathbf{L}}$, our incomplete Cholesky factor $\widehat{\mathbf{L}}$ only contains real entries. Our approach is equivalent to a non-optimal diagonal compensation, and while it can still break down if we have a zero on the diagonal, we do not observe this to happen in practice.

Pseudocode for the algorithm to compute our incomplete Cholesky factor $\widehat{\mathbf{L}}$ is shown in Algorithm 6. We use Figure 4.1 to illustrate how Algorithm 6 handles a column with a negative diagonal element during the factorization step. Essentially, we use a sign vector $\boldsymbol{\eta}$ to prevent the square root of a negative diagonal element. If the diagonal of column i is negative (meaning $\widetilde{\mathbf{L}}_{ii}$ and $\widehat{\mathbf{L}}_{ii}$ are negative), we set $\boldsymbol{\eta}_i = -1$ (Algorithm 6, line 6), and then multiply $\widehat{\mathbf{L}}_{ii}$ by $\boldsymbol{\eta}_i$ in order to take the square root of a positive number. With these updates, the entry $\widehat{\mathbf{L}}_{ii}$ is equal to the magnitude of the purely imaginary number $\sqrt{\widetilde{\mathbf{L}}_{ii}}$. For example, in Figure 4.1, after updating $\widehat{\mathbf{L}}_{22}$, we see that $\widehat{\mathbf{L}}_{22} = 2$, which is equal to the magnitude of the purely imaginary number $\sqrt{\widetilde{\mathbf{L}}_{22}} = 2i$.

Once the diagonal of the column i becomes purely imaginary, the remaining entries in column i are also purely imaginary since we divide them by the diagonal entry. Notice there is an extra factor of $\boldsymbol{\eta}_i = -1$ when we divide the real row entries below the diagonal of column i by $\widehat{\mathbf{L}}_{ii}$ (Algorithm 6, line 12). This is to mimic the multiplication of the numerator and denominator of the row entries below the diagonal by the complex conjugate of the purely imaginary diagonal element when performing the complex division. We see in Figure 4.1 that after dividing $\widehat{\mathbf{L}}_{32}$ by $\widehat{\mathbf{L}}_{22}$, we multiply $\widehat{\mathbf{L}}_{32}$ by $\boldsymbol{\eta}_2 = -1$ in order to obtain $\widehat{\mathbf{L}}_{32} = -1$, which matches the magnitude of the purely imaginary number $\widetilde{\mathbf{L}}_{32} = -i$. Moreover, when updating the real columns in the lower right block of $\widehat{\mathbf{L}}$ with column i, those columns remain real because the multiplication of two purely imaginary numbers gives a real number (Algorithm 6, line 18).

An important fact is a column in $\widetilde{\mathbf{L}}$ can either contain only real entries or only purely imaginary entries. So in Algorithm 6, we also use the sign vector $\boldsymbol{\eta}$ to mark real and imaginary columns. The important thing to note is entries in the imaginary columns of our factor $\widehat{\mathbf{L}}$ are real numbers. For instance, in Figure 4.1, we see that the imaginary column 2 of our factor $\widehat{\mathbf{L}}$ consists of real entries that denote the magnitude of the purely imaginary numbers in column 2 of the factor $\widetilde{\mathbf{L}}$. Observe that we mark $\boldsymbol{\eta}_2 = -1$ for the imaginary column 2.

With purely imaginary entries in $\widetilde{\mathbf{L}}$, we obtain the preconditioner $\mathbf{P} = \widetilde{\mathbf{L}}\widetilde{\mathbf{L}}^H$, where $\widetilde{\mathbf{L}}^H$ is the conjugate transpose of the $\widetilde{\mathbf{L}}$. The matrix $\mathbf{P} = \widetilde{\mathbf{L}}\widetilde{\mathbf{L}}^H$ contains only real entries, as

$$\mathbf{P}_{ij} = \sum_{k=1}^{\min(i,j)} \widetilde{\mathbf{L}}_{ik}\overline{\widetilde{\mathbf{L}}_{jk}} \tag{4.3}$$

only consist of products of entries in the same column, and we know the product of two purely imaginary numbers gives a real number. Hence $\mathbf{P}_{ij}$ is real. Here, $\overline{\widetilde{\mathbf{L}}_{jk}}$ denotes the complex conjugate of the entry $\widetilde{\mathbf{L}}_{jk}$. Although our zero-fill imaginary incomplete Cholesky factorization differs from the full Cholesky factorization such that we are introducing purely imaginary entries into the factor $\widetilde{\mathbf{L}}$ and skipping entries that introduce fill-ins, our goal is to come up with a good enough preconditioner $\mathbf{P} = \widetilde{\mathbf{L}}\widetilde{\mathbf{L}}^H$ where $\kappa\left(\widetilde{\mathbf{L}}^{-1}\mathbf{A}_{\mathbb{F}\mathbb{F}}\widetilde{\mathbf{L}}^{-H}\right) \ll \kappa(\mathbf{A}_{\mathbb{F}\mathbb{F}})$.

The linear solve with our incomplete Cholesky factor $\widehat{\mathbf{L}}\widehat{\mathbf{L}}^H\mathbf{z} = \mathbf{r}_{\mathbb{F}}$ illustrated in Algorithm 7 gives a real solution $\mathbf{z}$ because we divide the imaginary entries introduced

Algorithm 7 IC(0) linear solve with imaginary columns $\boldsymbol{\eta}$

```
1: procedure ModifiedLinearSolve($\widehat{\mathbf{L}}$, $\mathbf{r}_{\mathbb{F}}$, $\boldsymbol{\eta}$)
2:     Initialize $\mathbf{z}_i = \mathbf{r}_{\mathbb{F}}$
3:     for each $i \in \mathbb{F}$ do                              ▷ Forward substitution $\widehat{\mathbf{L}}\mathbf{y} = \mathbf{r}_{\mathbb{F}}$
4:         $\mathbf{z}_i = \eta_i \frac{\mathbf{z}_i}{\widehat{\mathbf{L}}_{\mathbf{ii}}}$
5:         for each $j \in \mathbb{F}$ following $i$ do
6:             $\mathbf{z}_j = \mathbf{z}_j - (\boldsymbol{\eta}_i \widehat{\mathbf{L}}_{ji} \mathbf{z}_i)$
7:         end for
8:     end for
9:     for each $i \in \mathbb{F}$ do                              ▷ Backward substitution $\widehat{\mathbf{L}}^H\mathbf{z} = \mathbf{y}$
10:        for each $j \in \mathbb{F}$ preceding $i$ do
11:            $\mathbf{z}_i = \mathbf{z}_i - (\boldsymbol{\eta}_i \widehat{\mathbf{L}}_{ji} \mathbf{z}_j)$
12:        end for
13:        $\mathbf{z}_i = \frac{\mathbf{z}_i}{\widehat{\mathbf{L}}_{ii}}$
14:    end for
15: end procedure
```

during forward elimination (Algorithm 7, line 4) by the diagonal element in the imaginary column i (Algorithm 7, line 13). Note that we never need to deal with complex numbers because matrix entries and intermediate values are either real or purely imaginary, permitting a smaller memory footprint and faster implementation than one needing to deal with complex numbers whose real and imaginary parts are not zero.

4.1.6 Matrix pre-shifting

One other way to mitigate or reduce the frequency of encountering breakdowns in the factorization is to factorize the preshifted matrix $\mathbf{S}\mathbf{A}_{\mathbb{F}\mathbb{F}}\mathbf{S}$ rather than the original matrix $\mathbf{A}_{\mathbb{F}\mathbb{F}}$, where S is a diagonal scaling matrix. There are various options for the scaling matrix S. For example, S can be the inverse of the diagonals of the matrix $\mathbf{A}_{\mathbb{F}\mathbb{F}}$ [6] or the L_2-norm of the columns of $\mathbf{A}_{\mathbb{F}\mathbb{F}}$ [28].

Algorithm 8 Preconditioned conjugate gradient

```
1: procedure PCG($\mathbf{A}, \mathrm{b}$)
2:     $k = 0$
3:     Initialize $\boldsymbol{\lambda}^{(k)} = \mathbf{0}$
4:     Initialize $\mathbf{r}^{(k)} = \mathbf{b} - \mathbf{A}\boldsymbol{\lambda}^{(k)}$                ▷ Residual
5:     Compute $\mathbf{P}$
6:     do
7:         Solve $\mathbf{P}\mathbf{z}^{(k)} = \mathbf{r}^{(k)}$ for $\mathbf{z}^{(k)}$
8:         if $k = 0$ then
9:             Set $\mathbf{q}^{(k)} = \mathbf{z}^{(k)}$                ▷ Initialize search direction
10:        else
11:            Compute $\beta^{(k)}$                ▷ Equation 4.5
12:            $\mathbf{q}^{(k)} = \mathbf{z}^{(k)} + \beta^{(k)}\mathbf{q}^{(k-1)}$                ▷ Update search direction
13:        end if
14:        Compute $\alpha^{(k)}$                ▷ Equation 4.4
15:        $\boldsymbol{\lambda}^{(k+1)} = \boldsymbol{\lambda}^{(k)} + \alpha^{(k)}\mathbf{q}^{(k)}$                ▷ Update solution
16:        $\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha^{(k)}\mathbf{A}\mathbf{q}^{(k)}$                ▷ Update residual
17:        $k = k + 1$
18:    while ($k <$ max iteration or $\left\|\mathbf{r}^{(k)}\right\| > \varepsilon$)
19: end procedure
```

4.2 Preconditioned Conjugate Gradient method

Now we will discuss about applying CG on the transformed linear system $\widetilde{\mathbf{A}}\widetilde{\boldsymbol{\lambda}} = \widetilde{\mathbf{b}}$. With some mathematical manipulations [16], we do not need to explicitly construct the modified linear system $\widetilde{\mathbf{A}}\widetilde{\boldsymbol{\lambda}} = \widetilde{\mathbf{b}}$ when applying preconditioning to conjugate gradient. The preconditioned conjugate gradient (PCG) method is outlined in Algorithm 8.

Notice the PCG algorithm (Algorithm 8) is very similar to the standard CG algorithm (Algorithm 3), but that PCG needs to do the extra work of computing the preconditioning matrix $\mathbf{P}$ (Algorithm 8, line 5) and performing a linear solve (Algorithm 8, line 7). Also, the computation of the step size α and scalar β is slightly different for PCG as well, where

$$\alpha^{(k)} = \frac{\left(\mathbf{r}^{(k)}, \mathbf{z}^{(k)}\right)}{\left(\mathbf{q}^{(k)}, \mathbf{A}\mathbf{q}^{(k)}\right)}, \tag{4.4}$$

and

$$\beta^{(k)} = \frac{\left(\mathbf{r}^{(k)}, \mathbf{z}^{(k)}\right)}{\left(\mathbf{r}^{(k-1)}, \mathbf{z}^{(k-1)}\right)}. \tag{4.5}$$

Although PCG converges faster to a good approximate solution, the step size α in Equation 4.4 does not guarantee our solution to satisfy the complementarity conditions 2.35a - 2.35c, which makes the standard PCG method not suitable for solving MLCPs. In the next chapter, we will discuss two methods of applying PCG to solve MLCPs.

Chapter 5

Using PCG in MLCPs solvers

There are many algorithms that may be used to solve the linear system in Equation 2.40. When the system is large, iterative solvers are often preferable. Additionally, since the linear system is symmetric positive definite, the preconditioned conjugate gradient (PCG) method is a viable and effective algorithm, assuming that a suitable preconditioner can be found. Solving MLCPs, however, requires a method that can also handle the feasibility and complementarity conditions in Equation 2.36b, and the standard PCG method does not guarantee the satisfaction of such conditions.

In this thesis, we are specifically interested in principal pivoting methods that identify constraints that are at their bounds (tight) and provide a solution to the linear system formed by the remaining free variables. In this chapter, we propose two methods that make use of the PCG method in solving MLCPs. In addition, we also describe a regularization technique that will potentially speed up the convergence of our methods.

5.1 BPP with PCG

The first method we introduce involves a simple modification to the BPP method (Algorithm 1) proposed by Júdice [18]. The modification is using PCG to solve for the linear system in Equation 2.40 (Algorithm 10, line 11) rather than obtaining an exact solution to the linear system using the Cholesky factorization. The point of using PCG instead is to mitigate the curse of dimensionality. We summarize the BPP+PCG method in Algorithm 10.

The BPP+PCG method is similar to the PGS-SM method proposed by Silcowitz et al. [30] in that they also used PCG to solve the system in Equation 2.40. However, PGS-SM uses projected Gauss-Seidel to determine and reduce the free set, whereas our BPP+PCG method first executes a PCG solve on an unbounded problem to determine variable bounds and index sets, followed by pivoting and bound updates in subsequent iterations.

5.2 Preconditioned generalized conjugate gradient

The second method we propose is a preconditioned version of the generalized conjugate gradient method derived from the work of O'Leary [25]. We build on their work by applying an incomplete Cholesky preconditioner (Section 4.1). Our preconditioned generalized conjugate gradient (PGCG) method is summarized in Algorithm 11. The method uses the step size $\alpha_{\max}$ to make sure the constraint impulses $\boldsymbol{\lambda}$ stays within the bounds (Algorithm 11, lines 10 - 12). The step size $\alpha_{\max}$ at each inner iteration is given by

$$\alpha_{\max} = \min \left\{ \min_{\substack{i=1,\dots,s \\ \mathbf{q}_i < 0}} \frac{\boldsymbol{\lambda}_i^l - \boldsymbol{\lambda}_i}{\mathbf{q}_i}, \min_{\substack{i=1,\dots,s \\ \mathbf{q}_i > 0}} \frac{\boldsymbol{\lambda}_i^u - \boldsymbol{\lambda}_i}{\mathbf{q}_i} \right\}, \tag{5.1}$$

where $\boldsymbol{\lambda}_i^l$ and $\boldsymbol{\lambda}_i^u$ are the lower and upper bounds of the i-th component of the constraint impulse, $\mathbf{q}_i$ is the search direction and s denotes the size of the free set $\mathbb{F}$. In each inner iteration, the PGCG algorithm places all variables in $\mathbb{F}$ that hit the lower bound into the lower tight set $\mathbb{T}_\mathbf{l}$ and all variables that hit the upper bound into the upper tight set $\mathbb{T}_\mathbf{u}$ (Algorithm 11, lines 14 and 15). With the free set $\mathbb{F}$ changing, the matrix $\mathbf{A}_{\mathbb{F}\mathbb{F}}$ changes as well, and so the residual vector $\mathbf{r}_\mathbb{F}$, search direction $\mathbf{q}_\mathbb{F}$ and the preconditioning matrix $\mathbf{P}$ have to be re-computed (Algorithm 11, lines 16 - 19).

Algorithm 9 Adaptive regularization

```
1: procedure ADAPTIVEREGULARIZE($\mathbf{A}_{\mathbb{FF}}$, $\epsilon$, $\mathbf{d}$)
2:     for each $i \in \mathbb{F}$ do
3:         if PIVOTCOUNT($i$) $> \mathbf{d}_i$ then
4:             $\mathbf{A}_{\mathbb{FF}_{i,i}} = \mathbf{A}_{\mathbb{FF}_{i,i}} + \epsilon$
5:             $\mathbf{d}_i = 2\mathbf{d}_i$
6:         end if
7:     end for
8: end procedure
```

If no variables move from the free set $\mathbb{F}$ into the tight set $\mathbb{T}$ during an inner iteration, then we apply standard PCG (Algorithm 11, lines 11 and 21 - 22) to update the constraint impulses $\boldsymbol{\lambda}_{\mathbf{F}}$. Note that if the problem only contains bilateral constraints, then our PGCG algorithm will be equivalent to the standard preconditioned conjugate gradient algorithm, as no variables will ever be placed into the tight set $\mathbb{T}$.

After the inner iteration, similar to BPP+PCG (Algorithm 10), we put variables that do not satisfy Equation 2.35a or 2.35b from the tight set $\mathbb{T}$ back into the free set $\mathbb{F}$ for subsequent PGCG inner iterations to correct (Algorithm 11, lines 27 - 29).

The PGCG and BPP+PCG algorithms share some similarities, but there are also important differences. BPP+PCG pivots variables after running the standard PCG algorithm, whereas PGCG modifies the standard PCG algorithm to pivot variables that reach a bound into the tight sets. We observed that the fine-scaled pivoting in PGCG also helps to avoid cycling of the index set, which impacts solver progress.

5.3 Warm starting

To have a better initial guess of the free set $\mathbb{F}$, tight set $\mathbb{T}$ and the constraint impulses $\boldsymbol{\lambda}$, we use the free and tight sets from the previous time step. We set the tight entries of $\boldsymbol{\lambda}$ to the bounds $\mathbf{l}$ or $\mathbf{u}$ and the free entries to zero.

Algorithm 10 Block principal pivoting with PCG

```
1: procedure BPPWithPCG($\mathbf{A}$, $\mathbf{b}$, $\boldsymbol{\lambda}$, $\boldsymbol{\lambda}^l$, $\boldsymbol{\lambda}^u$, $\mathbb{F}$, $\mathbb{T}_l$, $\mathbb{T}_u$, $\mathbf{d}$)
2:     Initialize $k = 0$
3:     while $\mathbb{F}$, $\mathbb{T}_l$ and $\mathbb{T}_u$ changes or $k <$ max iteration do
4:         for each $i \in \mathbb{T}_\mathbf{l}$ do                          ▷ Clamp infeasible variables
5:             $\boldsymbol{\lambda}_i \leftarrow \boldsymbol{\lambda}_i^l$
6:         end for
7:         for each $i \in \mathbb{T}_\mathbf{u}$ do                          ▷ Clamp infeasible variables
8:             $\boldsymbol{\lambda}_i \leftarrow \boldsymbol{\lambda}_i^u$
9:         end for
10:        $\widetilde{\mathbf{b}} \leftarrow -\mathbf{b}_\mathbb{F} - \mathbf{A}_{\mathbb{F}\mathbb{T}_l}\boldsymbol{\lambda}_{\mathbb{T}_l} - \mathbf{A}_{\mathbb{F}\mathbb{T}_u}\boldsymbol{\lambda}_{\mathbb{T}_u}$
11:        $\boldsymbol{\lambda}_\mathbb{F} \leftarrow \mathrm{PCG}(\mathbf{A}_{\mathbb{F}\mathbb{F}}, \widetilde{\mathbf{b}})$                          ▷ Section 4.2
12:        $\mathbf{w} = \mathbf{A}\boldsymbol{\lambda} + \mathbf{b}$
13:        for $i = 1$ to $m$ do
14:            Pivot index $i$ in $\mathbb{F}$, $\mathbb{T}_l$, or $\mathbb{T}_u$ using Equations 2.41 - 2.44.
15:        end for
16:        if useAdaptiveReg then
17:            AdaptiveRegularize($\mathbf{A}_{\mathbb{F}\mathbb{F}}$, $\epsilon$, $\mathbf{d}$)                          ▷ Section 5.4
18:        end if
19:        $k = k + 1$
20:    end while
21: end procedure
```

5.4 Adaptive regularization

Due to our problems being ill-conditioned, our PCG-based solvers may still have trouble determining the correct index set with reasonable outer iteration counts. To mitigate this, we propose adding a varying amount of positive shift ϵ to the diagonal entries of the matrix $\mathbf{A}_{\mathbb{F}\mathbb{F}}$ that correspond to friction variables in the system. The physical meaning of this adaptive regularization technique is the addition of instantaneous slip, where we make the problem better conditioned by allowing objects in the simulation to slip more than it should. Because we prioritize avoiding interpenetration, we only shift the friction variable entries in $\mathbf{A}_{\mathbb{F}\mathbb{F}}$.

We outline the adaptive regularization technique in Algorithm 9. The amount of shift we add to a diagonal element is dependent on how often the corresponding friction variable goes into and out of the tight set $\mathbb{T}$. If the number of times the friction variable i

goes into and out of the tight set, PIVOTCOUNT(i), exceeds a certain limit, $\mathbf{d}_i$, we shift the diagonal entry of column i by ϵ (Algorithm 9, lines 3 - 4).

The adaptive regularization is applied after updating the free and tight set in the PCG solvers at the end of each outer iteration (lines 16 - 18 of Algorithm 10 and lines 30 - 32 of Algorithm 11), where we *double* the pivot limit whenever we apply such regularization in order to further soften the problem if the friction variables pivot more in later iterations (Algorithm 9, line 5). For simplicity, we initialize all friction variables to have the same initial pivot limit.

An important note is our adaptive regularization technique is different than the diagonal compensation [27] or partial shift [6] preconditioning technique since the latter's goal is used to obtain a successful incomplete Cholesky factorization, while our regularization technique is used to soften the conditioning of the problem. Another distinction is this adaptive regularization technique can also be applied to the direct BPP solver.

Algorithm 11 Preconditioned generalized conjugate gradient

```
1: procedure PGCG($\mathbf{A}, \mathbf{b}, \boldsymbol{\lambda}, \boldsymbol{\lambda}^l, \boldsymbol{\lambda}^u$, $\mathbb{F}$, $\mathbb{T}_l$, $\mathbb{T}_u$, $\mathbf{d}$)
2:     $k = 0$
3:     do
4:         $\mathbf{r}_\mathbb{F} = -\mathbf{b}_\mathbb{F} - \mathbf{A}_{\mathbb{F}\mathbb{F}}\boldsymbol{\lambda}_\mathbb{F} - \mathbf{A}_{\mathbb{F}\mathbb{T}_l}\boldsymbol{\lambda}_{\mathbb{T}_l} - \mathbf{A}_{\mathbb{F}\mathbb{T}_u}\boldsymbol{\lambda}_{\mathbb{T}_u}$        ▷ Residual
5:         Compute $\mathbf{P}$        ▷ Section 4.1
6:         Solve $\mathbf{P}\mathbf{z} = \mathbf{r}_\mathbb{F}$ for $\mathbf{z}$
7:         $j = 0$
8:         Initialize $\mathbf{q} = \mathbf{z}$        ▷ Search direction
9:         do
10:            Compute $\alpha_{\max}$        ▷ Equation 5.1
11:            Compute $\alpha_{\mathrm{CG}}$        ▷ Equation 3.3
12:            $\alpha = \min\{\alpha_{\mathrm{CG}}, \alpha_{\max}\}$        ▷ Step size
13:            $\boldsymbol{\lambda}_\mathbb{F} = \boldsymbol{\lambda}_\mathbb{F} + \alpha\mathbf{q}$
14:            if variables in $\boldsymbol{\lambda}_\mathbb{F}$ hit bounds in $\boldsymbol{\lambda}^l$ or $\boldsymbol{\lambda}^l$ then
15:                Place variables into $\mathbb{T}_l$ or $\mathbb{T}_u$.
16:                Recompute $\mathbf{r}_\mathbb{F}$        ▷ See step 4
17:                Recompute $\mathbf{P}$.        ▷ Section 4.1
18:                Solve $\mathbf{P}\mathbf{z} = \mathbf{r}_\mathbb{F}$ for $\mathbf{z}$
19:                Reset $\mathbf{q} = \mathbf{z}$
20:            else        ▷ Standard PCG
21:                $\mathbf{q} = \mathbf{z} + \beta\mathbf{q}$
22:                $\mathbf{r}_\mathbb{F} = \mathbf{r}_\mathbb{F} - \alpha\mathbf{A}_{\mathbb{F}\mathbb{F}}\mathbf{q}$
23:            end if
24:            $j = j + 1$
25:        while ($j <$ max inner iteration) and ($\|\mathbf{r}_\mathbb{F}\| < \varepsilon$)
26:        $\mathbf{w} = \mathbf{A}\boldsymbol{\lambda} + \mathbf{b}$
27:        for $i = 1$ to $m$ do
28:            Pivot index $i$ in $\mathbb{F}$, $\mathbb{T}_l$, or $\mathbb{T}_u$ using Equations 2.41 - 2.44.
29:        end for
30:        if useAdaptiveReg then
31:            AdaptiveRegularize($\mathbf{A}_{\mathbb{F}\mathbb{F}}$, $\epsilon$, $\mathbf{d}$)        ▷ Section 5.4
32:        end if
33:        $k = k + 1$
34:    while ($k <$ max outer iteration) and ($\mathbb{F}$, $\mathbb{T}$ changes)
35: end procedure
```

Chapter 6

Results

This section evaluates the performance of the PCG variant solvers for several challenging scenarios. All simulations were performed on an Intel Core i7-7700HQ (2.80 GHz) with 32 GB of RAM. We use the Vortex physics engine [8] to perform collision detection and computing constraint Jacobian matrices. A time step of $h = 1/60$ s is used for all of the examples.

The numerical methods, including computation of preconditioners, are implemented in C++ using the Eigen linear algebra library [17]. Double precision is used for scalar, vector, and matrix operations. A compressed sparse column format is used to store the incomplete Cholesky factorizations and other sparse matrices. Eigen's sparse Cholesky factorization implementation is used for the baseline BPP solver. A maximum of 20,000 inner and outer iterations is used for the PCG variant solvers in order to allow them to fully converge. We use a tolerance of $\varepsilon = 1\text{e}{-6}$ for terminating the PCG iterations (line 18 of Algorithm 8 and line 25 of Algorithm 11) and use a tolerance of $1\text{e}{-10}$ for satisfying the condition of Equation 2.35.

6.1 Examples

Figure 1.1 shows screenshots from the examples used in our experiments, and Table 6.1 summarizes important information about each simulation (e.g., maximum number of constraints, maximum condition number). Due to large mass ratios and redundant constraints in all these examples, the MLCP matrices for these examples have large condition numbers. To handle redundant constraints, constraint compliances (diagonal values of **C** in Equation 2.27) ranging from 10^{-7} to 10^{-10} are used to regularize the system. Note that unconnected systems are split into multiple islands, and we apply the PCG solvers to get the constraint impulses for each island. We now give brief descriptions of the five challenging scenarios.

6.1.1 Boxes

This example is composed of boxes of mass 2.5 kg each, falling onto the ground. Eventually, the boxes form separate stable stacks. This is a simple scene testing how the solvers handle examples with a large amount of static friction.

6.1.2 Tower

This example consists of a log tower made from 160 logs of mass 50 kg each, stacked with a 2500 kg block falling from the top. The initially stable stack will collapse completely due to the rolling ball colliding with it. A large number of contact constraints are generated between the logs when the tower collapses.

6.1.3 Spinner

This example is a complex system where a 10 kg plate and a 2500 kg ball are suspended by 8 inextensible cables. Each cable contains cable links of 0.1 kg each that are coupled to neighbouring links using a joint with 6 degrees of freedom. This system is used to test the ability of the PCG solvers to handle ill-conditioned problems with large mass ratios.

6.1.4 Chain

This example simulates a stiff chain wrapping around a rod. The chain is composed of 100 links of mass 0.25 kg that are connected by a universal joint. It has a box of mass 500 kg attached to it. Many friction variables are sliding (tight) when the chain wraps fully around the rod.

6.1.5 Trucks

This simulation involves four trucks of mass 8000 kg each falling onto a net of very stiff cables. Each cable has a linear density of 5 kg/m and is constructed from 48 - 65 bodies and more than 280 constraints. Each truck has 22 bodies and more than 120 constraints. We use this example to stress test the performance of the solvers.

6.1.6 Rocks

This example has a large pile of rocks with mass ranging from $80-250$ kg each falling onto the ground, which is then pushed by a stiff, lightweight shovel of 1 kg. A large number of non-interpenetration and friction constraints are present throughout the simulation due to the collisions between rocks in the pile.

6.1.7 Wall

This test case involves a static brick wall constructed with bricks of mass 2.5 kg that is hit by a metal box of mass 10 kg. The brick wall starts to collapse when the metal box comes into contact with it. Initially, there are small gaps between the bricks so that there is no contact at the start of the simulation. Although the setup and simulation of this example are similar to the tower example, only part of the brick wall collapses, meaning there is

Table 6.1: Maxima of mass ratio, total number of constraints, and condition number of $\mathbf{A}$.

Example	**Mass ratio**	**Constraints**	**Condition**
Boxes	$-$	$12,000$	1e+08
Tower	50:1	$1,300$	1e+08
Spinner	$25,000$:1	$2,920$	1e+06
Chain	$2,000$:1	$1,480$	1e+08
Trucks	$8,000$:1	$5,650$	1e+10
Rocks	250:1	$1,800$	1e+09
Wall	4:1	$1,006$	1e+09

a mixture of kinetic and static friction during the impact between the metal box and the wall.

6.2 Performance comparison

The performance of the PCG solvers is evaluated in three ways: i) the number of outer iterations required to converge to the correct index set, ii) the LCP error produced by the solver, and iii) the computational time required to solve the MLCP. Additionally, we include iv) the average number of inner PCG iterations required for the PCG solvers to converge to a solution per frame in order to further analyze the PCG solvers' performance.

Our comparisons focus on challenging frames with a non-trivial solution (i.e., where the tight set $\mathbb{T}$ is non-empty). Simulations of multiple unconnected islands are not parallelized, and the results simply report the sum of the solve times, outer iterations, and LCP errors across all islands for each frame. We then compute the average of the three quantities respectively across all the frames for each example. As for the average inner PCG iterations, we report the results from the island that takes the longest time to solve.

Table 6.2: Comparison of average outer iteration counts.

Example	BPP+PCG	PGCG	BPP
Boxes	1.6	1.5	1.6
Tower	40.9	31.7	40.9
Spinner	9.6	9.5	9.6
Chain	68.0	17.2	68.0
Trucks	47.6	7.8	47.6
Rocks	17.5	64.4	17.5
Wall	1.3e+03	32.9	1.9e+03

Table 6.3: Maxima of LCP error [13] produced by each solver.

Example	BPP+PCG	PGCG	BPP
Boxes	2.3e−11	9.7e−23	2.3e−11
Tower	1.9e−11	7.6e−12	2.8e−17
Spinner	6.3e−14	6.0e−14	2.7e−22
Chain	1.2e−13	1.1e−13	3.5e−20
Trucks	8.6e−11	9.3e−11	1.4e−13
Rocks	9.4e−17	2.4e−16	6.4e−19
Wall	1.8e+10	1.1e−11	1.8e+10

6.2.1 Outer iteration count

Table 6.2 presents the average outer iteration count needed to determine the correct index sets $\mathbb{F}$ and $\mathbb{T}$ across all frames for both PCG solvers and the baseline BPP solver. Since BPP+PCG essentially replaces the Cholesky factorization and linear solve of the baseline BPP solver with standard PCG, we observe that the outer iterations needed to determine the index sets for BPP and BPP+PCG are similar for almost all examples.

Table 6.2 also illustrates that the wall and trucks examples are where the PGCG solver is most effective in terms of determining the index set in the least number of outer iterations. We believe this is due to the additional pivoting within the inner iteration that helps the PGCG solver make faster progress in determining the correct index set. It appears that the additional pivoting is even necessary for convergence for the wall example, where BPP+PCG and BPP struggle to converge on frames where the metal box is in contact with the static brick wall. On the other hand, for the chain example, we see

Table 6.4: Comparison of average solve time (ms) per frame for each solver method.

Example	BPP+PCG	PGCG	BPP
Boxes	2.2e+01 (3.1×)	2.7e+01 (2.6×)	6.8e+01
Tower	6.7e+01 (0.5×)	1.6e+02 (0.2×)	3.5e+01
Spinner	1.3e+00 (4.1×)	1.0e+00 (5.6×)	5.3e+00
Chain	1.1e+03 (0.5×)	1.4e+03 (0.4×)	5.1e+02
Trucks	9.3e+03 (0.1×)	5.1e+03 (0.2×)	9.8e+02
Rocks	2.0e+04 (0.03×)	5.0e+04 (0.01×)	6.3e+02
Wall	5.4e+04 (0.09×)	6.0e+02 (7.9×)	4.7e+03

that PGCG has a slower solve time than BPP+PCG despite a lower average outer iteration count. There are three reasons for this. The first reason is there are actually more frames where BPP+PCG (and BPP) converge in fewer outer iterations than the PGCG solvers. We believe these frames in the chain example convey the limitation of the additional pivoting in PGCG; the pivoting within the inner iterations gives a poor guess of the correct index set. Secondly, there exist frames in the chain example where BPP+PCG (and BPP) requires many outer iterations to converge and this explains the increased average outer iteration count for both solvers. Nevertheless, there are few frames where both solvers struggle to converge in a reasonable number of outer iterations. Thirdly, the system sizes of the chain example are not very large, and thus, the additional IC(0) factorization cost incurred by BPP+PCG for frames where it struggles is small enough that it still does better on average than PGCG.

Figure 6.1 illustrates the total inner PCG iterations across all outer iterations for each frame of the chain example. Our robust zero-fill incomplete Cholesky preconditioner not only allows PGCG to converge 100 times faster than the Jacobi preconditioner, but is also comparable to the more effective version of the incomplete Cholesky preconditioner with a relatively low drop tolerance of $1\text{e}{-5}$. The reason is the convergence of the solver is affected by the large shifts that are added to the diagonals of the lead matrix $\mathbf{A}_{\mathbb{FF}}$ during the factorization step of ICT (in order to prevent negative diagonal entries). Unsurprisingly, as the drop tolerance used for ICT is smaller, it becomes a much more

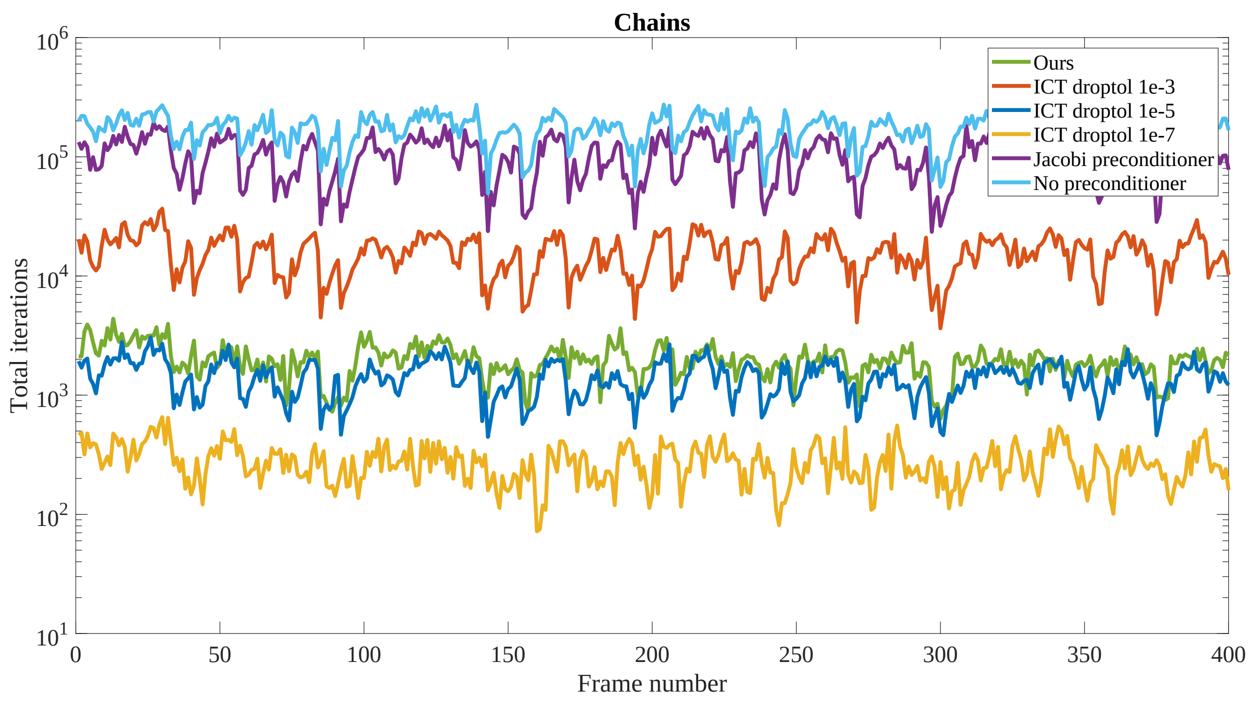


Figure 6.1: With our implementation of the incomplete Cholesky preconditioner that handles negative pivots, for the chains example, our PGCG solver converges two magnitudes faster than the Jacobi preconditioner and is comparable to ICT with a relatively low drop tolerance of $1\text{e}{-5}$.

effective preconditioner compared to our zero-fill incomplete Cholesky preconditioner due to it being much closer to the full Cholesky factor and large diagonal shifts not required to prevent having negative diagonal elements during the factorization step.

6.2.2 LCP error

To measure the accuracy of the solvers, the unit consistent error measure for MLCPs proposed by Enzenhöfer et al. [13] is computed and averaged over all constraints and all frames. The maximal LCP errors (in Joules) for all examples are presented in Table 6.3. We observe that the errors are reasonably low for all three solvers in all examples other than the wall example. This is because the solvers converge for almost all examples and because our choice of the stopping tolerance is small. Note that due to rounding the values to 1 decimal place, we see some values in Table 6.3 being identical.

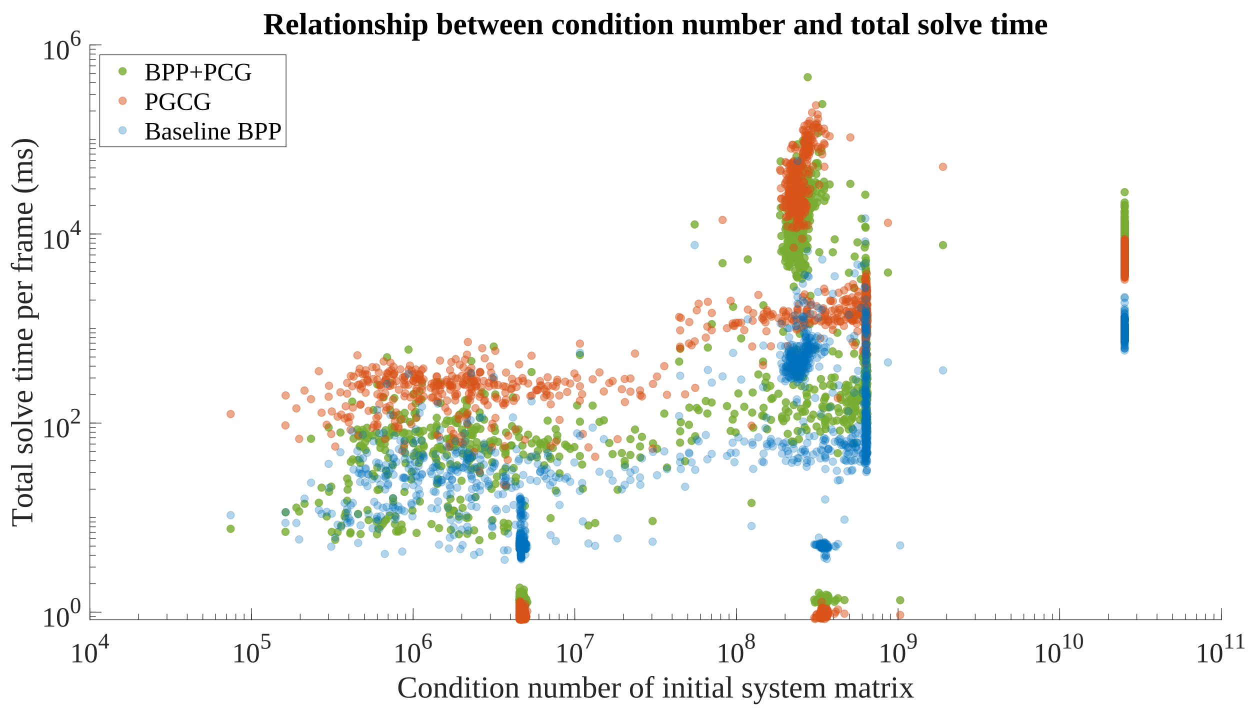


Figure 6.2: A log-log plot demonstrating how the total solver time of both PCG variant solvers worsens as the problems become more ill-conditioned. This plot includes the total solve time per frame of all examples we considered.

6.2.3 Solve time

The average MLCP solve time of PCG variant solvers is compared with the baseline direct BPP solver in Table 6.4, where the numbers inside the brackets are the speed up factors. We observe that the PCG solvers are much slower than the direct BPP solver in almost all cases, with the box, spinner, and the wall being the only examples where the PCG variant solvers outperform BPP. One reason for this is the BPP+PCG and BPP solver struggles to converge to an index set for the wall example on certain frames (Section 6.2.1). Another reason is the convergence rate of the PCG solvers depends on the condition number of the system matrix and also the number of imaginary columns in the incomplete Cholesky factor. The box and spinner are examples that consist of incomplete Cholesky factors with the least number of imaginary columns compared to the other examples, which explains why the PCG solvers perform best in the box and spinner examples. The log-log plot in Figure 6.2 demonstrates how the ill-conditioning of the problem seriously impacts the performance of the PCG solvers, to a point where BPP still outperforms the PCG solvers even when the system size increases (Figure 6.3). Note the clusters in Figure 6.2 and 6.3 where the PCG solvers outperforms the BPP solver corresponds to frames from the spinner example.

Despite this, it is known that direct methods struggle for large examples due to the large number of fill-ins introduced during the Cholesky factorization (even when a reasonable ordering strategy is chosen). A large number of fill-ins will increase the FLOPs needed to perform the factorization, which significantly slows down the factorization time. The fill-ins introduced might even cause the factorization to fail due to insufficient memory [6]. Hence, we anticipate that for problems where the number of constraints is orders of magnitude larger than the examples we considered, we will see our PCG solvers start to shine. We believe the effect of the condition number on the convergence of the PCG solvers will not be as serious as the effect of a large number of fill-ins on the Cholesky factorization.

6.2.4 Inner PCG iteration count

We now take a further look into how the number of imaginary columns in our incomplete Cholesky preconditioner affects the inner PCG iterations of our two methods. Figure 6.4 illustrates that the performance of the inner PCG solve plummets as the number of imaginary columns in the incomplete Cholesky preconditioner increases. We focus on the chains, trucks and rocks examples because these are the examples where the PCG solvers struggle the most. Note that we only display the results from the island that is most challenging to solve in Figure 6.4 in order to get a clear picture of the effect of the number of imaginary columns on the convergence of the PCG solvers.

We see the rocks example is the example that contains the most imaginary columns, which explains why the PCG variant solvers perform particularly bad on the rocks example such that the solve time is more than 100 times slower. The PGCG solver

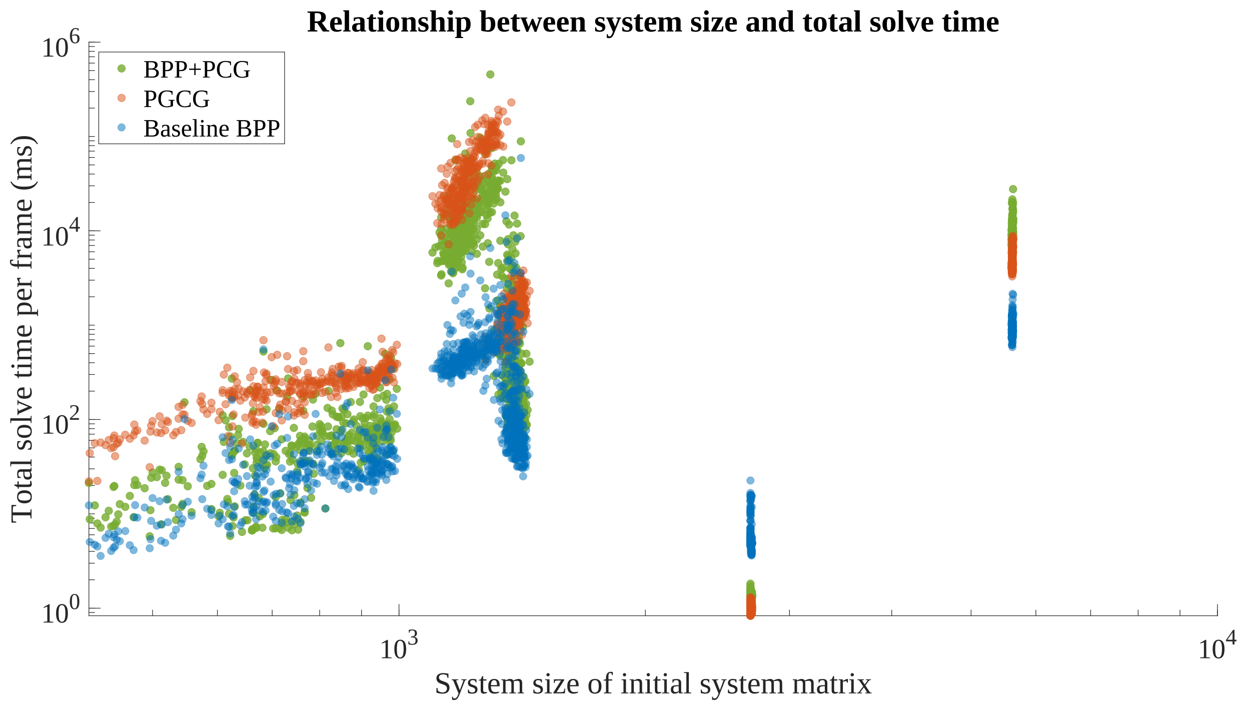


Figure 6.3: A log-log plot illustrating that the PCG solvers still performs poorly compared to BPP even when the initial system matrix **A** gets larger.

struggles with this example due to more frequent computation of the preconditioner. The deterioration in the performance of the inner PCG solve due to large number of negative imaginary columns also affects the quality of the index set obtained per outer iteration, as we see PGCG took much more outer iterations to determine the correct index set (Table 6.2). Furthermore, Figure 6.4 also shows that the average percentage of columns that encountered negative diagonal elements is less for the PGCG solver. This is due to having preconditioner computations that involves less imaginary columns in the modified inner PCG iterations.

6.2.5 Visual comparison

Our PCG variant solvers produce simulations that are numerically stable and physically similar compared to the baseline BPP solver, as demonstrated in Figure 6.5. This is due to the LCP errors being low for the PCG solvers (see Table 6.3). Despite this similarity,

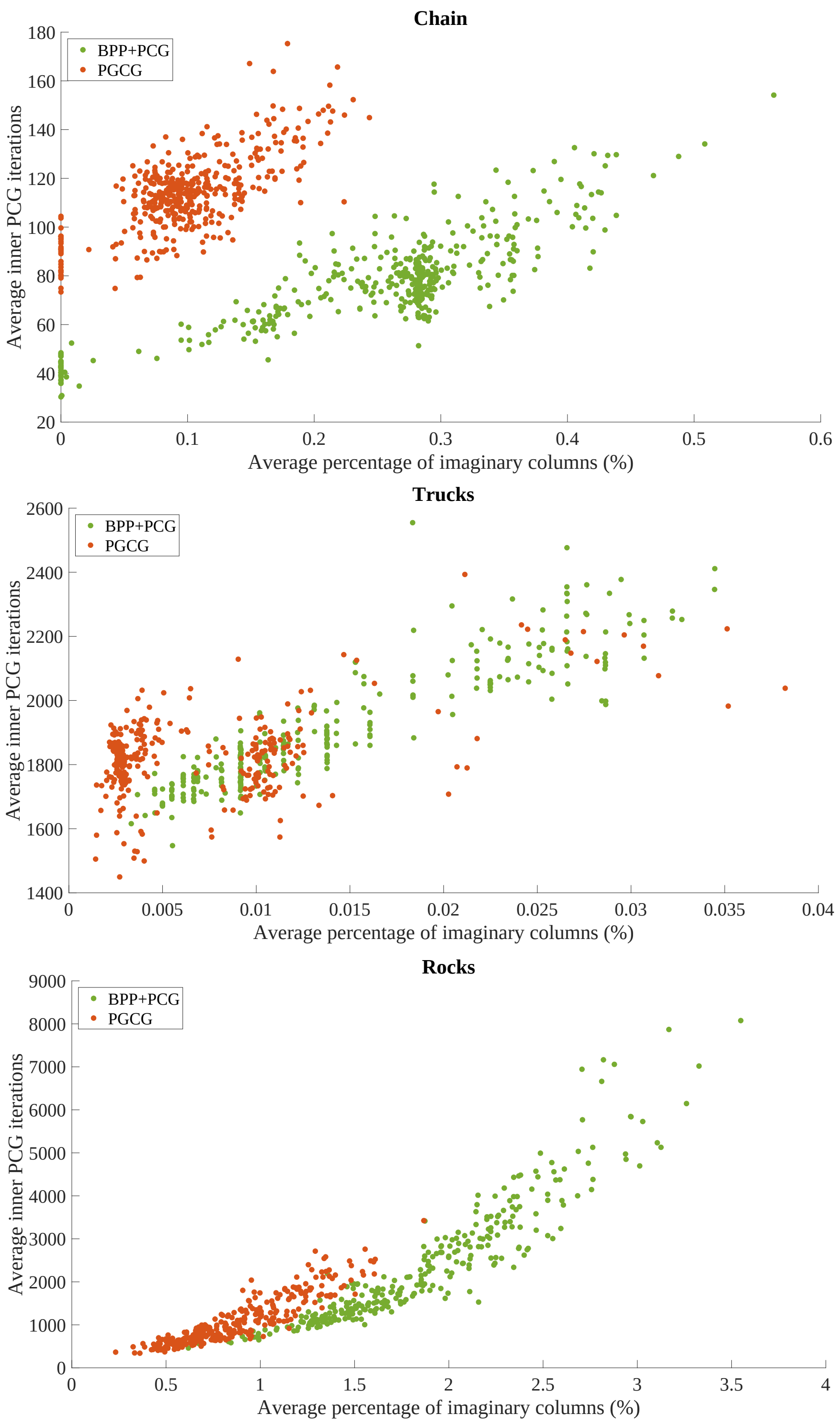


Figure 6.4: As the number of imaginary columns increases, the inner iterations increases for the PCG solvers. The average imaginary columns is less for PGCG because it has preconditioner computations with less imaginary columns in the inner iterations.

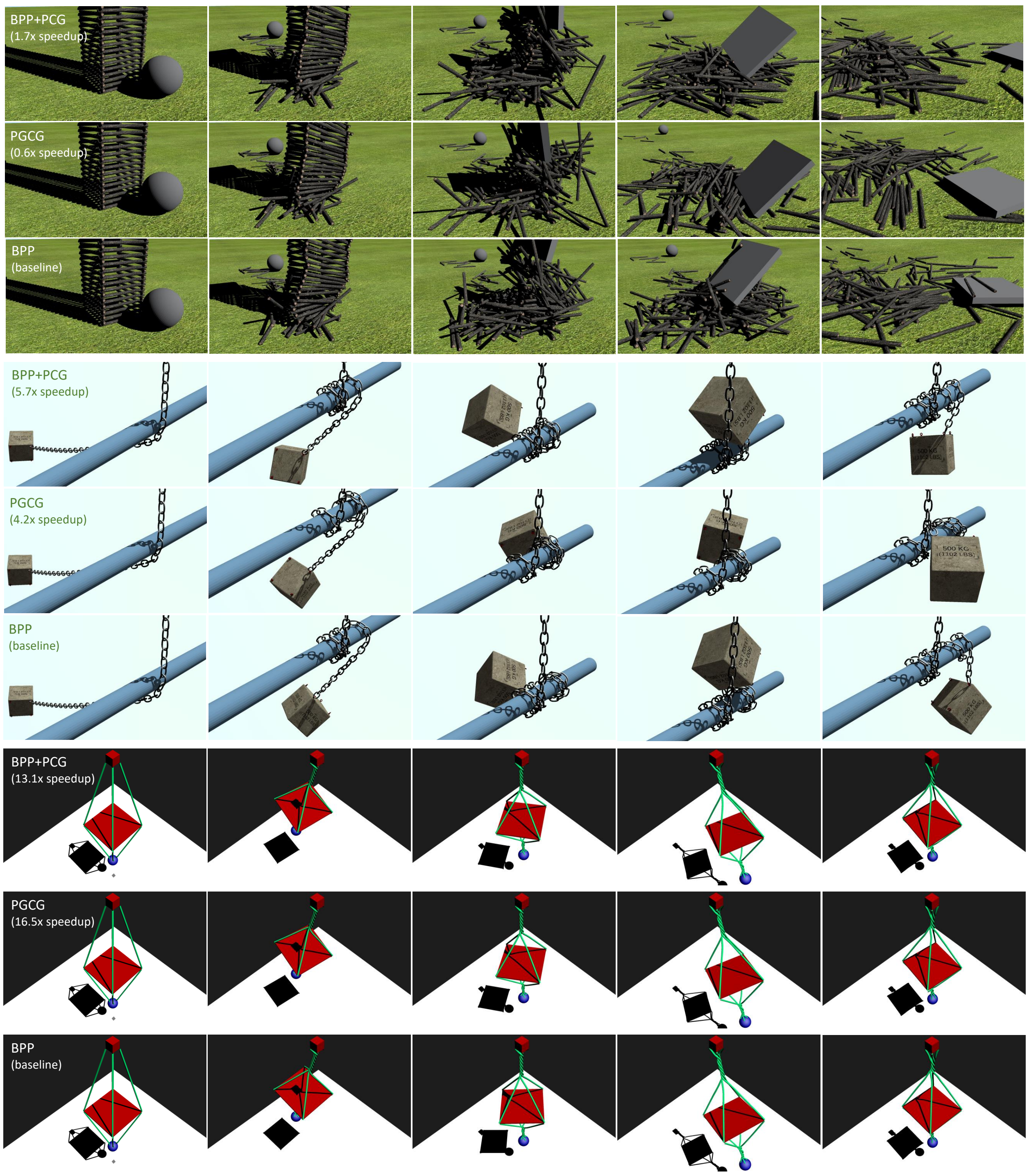


Figure 6.5: Snapshots showing how the log tower (top), chain (middle) and spinner (bottom) examples progress when different solvers are used. The three solvers produce visually similar results.

Figure 6.5 shows that both solvers produce different behaviors for some examples. This is because the solvers produce different index sets in some frames. The different index sets will, for example, cause the logs to interact differently when the tower is collapsing, which leads to the block and the logs to land at different positions once the tower collapses completely. Another instance of this behavior difference is seen in the chain example. The different index sets affect the accumulated twists and how the block collides with the chain while the chain is wrapping around the rod, leading to a difference in how the chain fully wraps around the rod.

6.3 Performance with adaptive regularization

We illustrate how different parameters of the adaptive regularization technique described in Section 5.4 improves the convergence rate of the BPP+PCG solver. Table 6.5 demonstrates the result of adding different amounts of adaptive regularization to the system matrices of the three most challenging examples when applying the BPP+PCG solver. We experimented with different initial pivot limits and different amounts of positive shifts. Recall we initialize all friction variables to have the same initial pivot limit for simplicity sake, and the pivot limit $\mathbf{d}_i$ is doubled each time its corresponding diagonal entry is shifted. We see a clear boost in convergence rate when adaptive regularization is added, where the BPP+PCG solver converges around 3.9 times faster for the wrapping chain example if an initial pivot limit of 10 is used, and even faster if a smaller initial pivot limit is used.

One main caveat of this technique, however, is if we add too much regularization to the system matrix, objects may appear overly slippery and overly compliant, as seen from the right image in Figure 6.6, when an initial pivot limit of 5 is used for the wrapping chain example. On the other hand, if we use an initial pivot limit of 10 (middle image of Figure 6.6), the chain wraps around the rod in a similar manner as when no adaptive

Table 6.5: Average outer iteration count when different amount of regularization is added. A shift of 1e-3 is added to the diagonal entries of the system matrix whenever the corresponding friction variables exceed their pivot limits.

Example	Initial pivot limit **d**	Average outer iterations
Chain	–	68.0
	10	17.5 (3.9 ×)
	5	14.8 (4.6 ×)
Trucks	–	47.6
	10	28.2 (1.7 ×)
	5	19.6 (2.4 ×)
Rocks	–	17.5
	10	16.5 (1.1 ×)
	5	17.1 (1.0 ×)

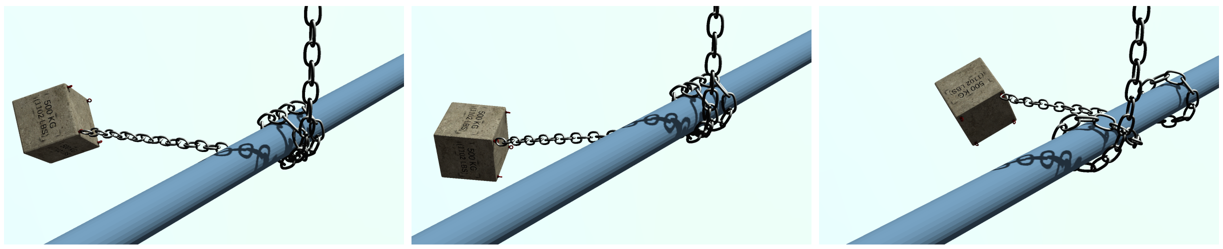


Figure 6.6: Different adaptive amounts of diagonal shift, showing results for an initial pivot limit of 10 (left), 5 (middle), and no regularization (right). The chain and rod become slippery if too much regularization is added.

regularization is used (left image of Figure 6.6). But this artifact mostly arises when there is a sufficient amount of sliding motion in the simulation.

The adaptive regularization technique can also be applied to the PGCG solver on problems where it struggles. However, the user has to be even more careful in choosing the parameters of the adaptive regularization technique for the PGCG solver due to pivoting happening even more often within the solver.

Notice that the adaptive regularization technique does not fix the poor performance of the BPP+PCG solver on the rock pile example. This is because the poor performance is related to the large number of imaginary entries in the incomplete Cholesky preconditioner.

Chapter 7

Conclusion and future work

In this thesis, we investigate two different ways of applying the preconditioned conjugate gradient method to solving frictional contact problems. We also propose an inexpensive failure-free zero-fill incomplete Cholesky preconditioner. While our PCG variant solvers are outperformed by the baseline BPP solver in most scenarios, we do obtain much more reasonable convergence rates when our incomplete Cholesky preconditioner is used instead of the Jacobi preconditioner. In some cases, our preconditioner is even comparable to the incomplete Cholesky with drop tolerance preconditioner for a relatively low drop tolerance. To help improve the number of outer iterations to determine the correct index set, we introduce an adaptive regularization scheme for friction variables that pivots too frequently to further speed up convergence. However, one must be careful with picking reasonable parameters when using the adaptive regularization technique to prevent the artifact of contacts that become noticeably slippery.

There are several directions to explore for future work. Despite the slow solve time results, we believe that our PCG variant solvers will begin to shine when applied to examples that have orders of magnitude more constraints than the examples we considered, as the Cholesky factorization will simply be too costly for those examples. We also believe our PCG variant solvers are well-suited for use in sub-substructuring solvers, such as the one recently proposed by Peiret et al. [26], if our PCG variant solvers are effective in solving larger scale problems. The interface solver in this work is a bottleneck because a direct method is used to solve interface constraint forces. We believe performance could be improved by using a parallelized version of our PCG variant solvers. Another important consideration is to reuse the zero-fill incomplete Cholesky factor whenever pivoting occurs rather than recomputing the factor from scratch. We did preliminary experiments on low-rank downdates to incrementally modify the incomplete Cholesky factorizations based on the index set, similar to the one proposed by Enzenhöfer et al. [12]. This idea shows promise and will benefit from further investigation.

Appendix A

Computing the constraint Jacobian

We first describe the physical interpretation of the constraint Jacobian. The rows of the constraint Jacobian are the directions in which motion is not allowed or directions in which an object can move without doing actual work. To describe the computation of the constraint Jacobian, we first elaborate more on the non-interpenetration constraint. Consider a physical system with body A colliding with body B (Figure A.1), with the point $\mathbf{p}$ being the contact point and $\hat{\mathbf{n}}$ being the unit contact normal. We use $\mathbf{c}_i$ to denote the center of mass position of body i, and $\mathbf{v}_i$ and $\boldsymbol{\omega}_i$ to be the linear and angular velocity of body i. The relative velocity at the contact point $\mathbf{p}$ is

$$\Delta \mathbf{v}_{\mathbf{p}} = (\mathbf{v}_B + \boldsymbol{\omega}_B \times \mathbf{r}_B) - (\mathbf{v}_A + \boldsymbol{\omega}_A \times \mathbf{r}_A). \tag{A.1}$$

where $\mathbf{r}_A = \mathbf{p} - \mathbf{c}_A$ and $\mathbf{r}_B = \mathbf{p} - \mathbf{c}_B$.

A.1 Constraint Jacobian for non-interpenetration constraints

To prevent bodies from penetrating into each other, we have to apply the following constraint on the normal component of the relative velocity at the contact point $w_{\hat{\mathbf{n}}}$ with

$$w_{\hat{\mathbf{n}}} = \Delta \mathbf{v}_{\mathbf{p}} \cdot \hat{\mathbf{n}} = 0. \tag{A.2}$$

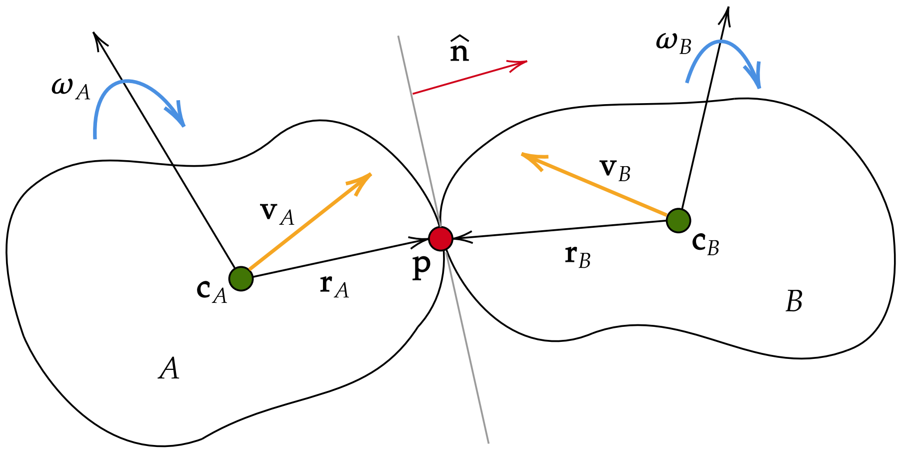


Figure A.1: When two bodies A and B are in contact, we need to apply constraints on the relative velocity in the normal direction in order to avoid interpenetration.

Expressing the velocity constraint $w_{\hat{\mathbf{n}}}$ in matrix-vector form, we have

$$w_{\hat{\mathbf{n}}} = \underbrace{\begin{bmatrix} -\hat{\mathbf{n}}^T & \hat{\mathbf{n}}^T[\mathbf{r}_A]_\times & \hat{\mathbf{n}}^T & -\hat{\mathbf{n}}^T[\mathbf{r}_B]_\times \end{bmatrix}}_{\mathbf{J}} \underbrace{\begin{bmatrix} \mathbf{v}_A \\ \boldsymbol{\omega}_A \\ \mathbf{v}_B \\ \boldsymbol{\omega}_B \end{bmatrix}}_{\mathbf{v}^+} = 0, \tag{A.3}$$

where we treat the cross product as a linear operator and express the cross product $\mathbf{r} \times \boldsymbol{\omega}$ as a matrix-vector product $[\mathbf{r}]_\times \boldsymbol{\omega}$ by defining

$$[\mathbf{r}]_\times \equiv \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{bmatrix}, \quad \text{where} \quad \mathbf{r} = (r_x, r_y, r_z). \tag{A.4}$$

Generally, for a system with n bodies and m non-interpenetration constraints, the constraint Jacobian matrix for the non-interpenetration constraint is a $m \times 6n$ matrix, as each body has 6 degrees of freedom. Note that only the blocks corresponding to the indices of the two bodies in contact will be non-zero for each non-interpenetration constraint. Hence the constraint Jacobian matrix $\mathbf{J}$ is sparse.

A.2 Constraint Jacobian for contact constraints

When simulating frictional contact, the constraint Jacobian is defined as:

$$\mathbf{J} = \begin{bmatrix} \hat{\mathbf{n}} & \hat{\mathbf{t}}_1 & \hat{\mathbf{t}}_2 \end{bmatrix}^T \begin{bmatrix} -\mathbf{I}_{3\times 3} & [\mathbf{r}_A]_\times & \mathbf{I}_{3\times 3} & -[\mathbf{r}_B]_\times \end{bmatrix}, \tag{A.5}$$

where $\hat{\mathbf{t}}_1$ and $\hat{\mathbf{t}}_2$ are the friction axes, $\mathbf{I}_{3\times 3}$ is the 3-by-3 identity matrix and the cross product matrix $[\mathbf{r}]_\times$ is defined in Equation A.4. We can see the constraint Jacobian $\mathbf{J}$ computation is very similar to the one described in Section A.1, with the only difference being we need to take into account the friction axes $\hat{\mathbf{t}}_1$ and $\hat{\mathbf{t}}_2$. Since we consider two bodies for each contact constraint, for a system with n bodies and m contact constraints, the constraint Jacobian matrix $\mathbf{J} \in \mathbb{R}^{m\times 6n}$ is sparse. Note that the non-interpenetration constraint Jacobian $\mathbf{J}_{\hat{\mathbf{n}}}$

$$\mathbf{J}_{\hat{\mathbf{n}}} = \begin{bmatrix} -\hat{\mathbf{n}}^T & \hat{\mathbf{n}}^T[\mathbf{r}_A]_\times & \hat{\mathbf{n}}^T & -\hat{\mathbf{n}}^T[\mathbf{r}_B]_\times \end{bmatrix}, \tag{A.6}$$

is exactly like the one shown in Equation A.3.

Bibliography

[1] AMESTOY, P. R., DAVIS, T. A., AND DUFF, I. S. Algorithm 837: Amd, an approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw. 30*, 3 (September 2004), 381–388.

[2] ANDREWS, S., AND ERLEBEN, K. Contact and friction simulation for computer graphics. *SIGGRAPH '21 Courses* (August 2021), 1–124.

[3] BAUMGARTE, J. Stabilization of constraints and integrals of motion in dynamical systems. *Computer Methods in Applied Mechanics and Engineering 1*, 1 (1972), 1–16.

[4] BENDER, J., ERLEBEN, K., AND TRINKLE, J. Interactive simulation of rigid body dynamics in computer graphics. *Computer Graphics Forum 33*, 1 (2014), 246–270.

[5] CHAN, W. M., AND GEORGE, A. A linear time implementation of the reverse cuthill-mckee algorithm. *BIT 20*, 1 (March 1980), 8–14.

[6] CHEN, J., SCHÄFER, F., HUANG, J., AND DESBRUN, M. Multiscale cholesky preconditioning for ill-conditioned problems. *ACM Trans. Graph. 40*, 4 (July 2021), 1–13.

[7] CLINE, M., AND PAI, D. Post-stabilization for rigid body simulation with contact and constraints. *Proc. of IEEE International Conference Robotics and Automation 3* (October 2003), 3744 – 3751.

[8] CM-LABS SIMULATIONS. Vortex studio, 2021.

[9] COTTLE, R., PANG, J., AND STONE, R. *The Linear Complementarity Problem*. Academic Press, 1992.

[10] CUTHILL, E., AND MCKEE, J. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th National Conference* (New York, NY, USA, 1969), ACM '69, Association for Computing Machinery, p. 157–172.

[11] DAVIS, T. A. *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. Society for Industrial and Applied Mathematics, USA, 2006.

[12] ENZENHÖFER, A., LEFEBVRE, N., AND ANDREWS, S. Efficient block pivoting for multibody simulations with contact. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2019), I3D'19, ACM.

[13] ENZENHÖFER, A., PEIRET, A., TEICHMANN, M., AND KÖVECSES. A unit-consistent error measure for mixed linear complementarity problems in multibody dynamics simulation with contact. *Proceedings of the ASME 2018 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference* (2018).

[14] ERLEBEN, K. Velocity-based shock propagation for multibody dynamics animation. *ACM Trans. Graph 26*, 2 (2007), 20 pages.

[15] GOLDENTHAL, R., HARMON, D., FATTAL, R., BERCOVIER, M., AND GRINSPUN, E. Efficient simulation of inextensible cloth. *ACM Trans. Graph. 26*, 3 (July 2007), 49–56.

[16] GOLUB, G. H., AND VAN LOAN, C. F. *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD, 2013.

[17] GUENNEBAUD, G., JACOB, B., ET AL. Eigen v3. http://eigen.tuxfamily.org, 2021.

[18] JÚDICE, J. J., AND PIRES, F. M. A block principal pivoting algorithm for large scale strictly monotone linear complementarity problems. *Computers & operations research 21*, 5 (1994), 587–596.

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

[20] Larionov, E., Fan, Y., and Pai, D. K. Frictional contact on smooth elastic solids. *ACM Trans. Graph. 40*, 2 (April 2021), 1–17.

[21] Ly, M., Jouve, J., Boissieux, L., and Bertails-Descoubes, F. Projective dynamics with dry frictional contact. *ACM Trans. Graph. 39*, 4 (July 2020), 1–8.

[22] Macklin, M., Kenny, E., Müller, M., Chentanez, N., Jeschke, S., and Makoviychuk, V. Non-smooth newton methods for deformable multi-body dynamics. *ACM Trans. Graph. 38*, 5 (2019), 1–20.

[23] MATLAB. Incomplete cholesky factorization (ichol), 2020.

[24] Niebe, S., and Erleben, K. Numerical methods for linear complementarity problems in physics-based animation. *Synthesis Lectures on Computer Graphics and Animation 7*, 1 (2015), 1–159.

[25] O'Leary, D. P. A generalized conjugate gradient algorithm for solving a class of quadratic programming problems. *Linear Algebra Appl. 34* (1980), 371–399.

[26] Peiret, A., Andrews, S., Kövecses, J., Kry, P. G., and Teichmann, M. Schur complement-based substructuring of stiff multibody systems with contact. *ACM Trans. Graph. 38*, 5 (2019), 1–17.

[27] Saad, Y. *"Preconditioning Techniques." Iterative Methods for Sparse Linear Systems.* PWS Publishing Company, 1996.

[28] Scott, J. A., and Tuma, M. Mi28: An efficient and robust limited-memory incomplete cholesky factorization code. *ACM Transactions on Mathematical Software 40*, 4 (June 2014), 1–19.

[29] SHEWCHUK, J. R., ET AL. An introduction to the conjugate gradient method without the agonizing pain, 1994.

[30] SILCOWITZ, M., NIEBE, S., AND ERLEBEN, K. Projected gauss-seidel subspace minimization method for interactive rigid body dynamics - improving animation quality using a projected gauss-seidel subspace minimization method. *GRAPP 2010 - Proceedings of the International Conference on Computer Graphics Theory and Applications 229* (05 2010), 38–45.

[31] VAN DE GEIJN, R. A. Notes on cholesky factorization, March 2011.

[32] WENDLAND, H. *Numerical Linear Algebra, An Introduction*. Cambridge University Press, United Kingdom, 2017.