Block Principal Pivoting with Incremental Cholesky Factorizations for Real-time Multibody Simulations

by

Nicolas LEFEBVRE

THESIS PRESENTED TO ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
IN PARTIAL FULFILLMENT OF A MASTER'S DEGREE
WITH THESIS IN SOFTWARE ENGINEERING
M.A.Sc.

MONTREAL, DECEMBER 17, 2020

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

BOARD OF EXAMINERS

THIS THESIS HAS BEEN EVALUATED

BY THE FOLLOWING BOARD OF EXAMINERS

Mr. Sheldon Andrews, Thesis Supervisor
Department of Software Engineering and IT, École de technologie supérieure

Mr. Michael McGuffin, Chair, Board of examiners
Department of Software Engineering and IT, École de technologie supérieure

Mr. Eric Paquette, Board of Examiners
Department of Software Engineering and IT, École de technologie supérieure

THIS THESIS WAS PRESENTED AND DEFENDED

IN THE PRESENCE OF A BOARD OF EXAMINERS AND THE PUBLIC

ON NOVEMBER 27, 2020

AT ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

ACKNOWLEDGEMENTS

First, I would like to thank the members of the McGill-ÉTS-CM Labs Applied Dynamics research group for their interest and their comments on my work. Also, Nabil Boukadida whose interest helped me refine my comprehension of the theory and Christopher Donnelly for reading and providing helpful feedback.

I also would like to thank Andreas Enzenhoefer, Daniel Holz and Marek Teichmann at CM Labs Simulations for all of their help in developing the efficient block pivoting prototype and their feedback on the main algorithm. Furthermore, Chapter 2 of this thesis is largely derived from the article *Efficient block pivoting for multibody simulations with contact* (Enzenhoefer, Lefebvre & Andrews, 2019) that was published and presented at the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D 2019). Andreas's help was integral for making this happen. Portions of the article were used in this thesis, including results and description of the efficient block pivoting algorithm, and this was done with the with permission of my co-authors.

Above all, I would like to give an extra special thank to my thesis supervisor Sheldon Andrews who introduced me to research, mentored me, gave me the space in which to work, and reviewed the drafts as I wrote them. Without his guidance and unwavering support, this work would never have been completed.

Finally, work done during the course of this thesis was supported by a Collaborative Research and Development (CRD) grant from the Natural Sciences and Engineering Research Council (NSERC) of Canada.

Pivot de Gauss par blocs avec factorisation de Cholesky incrémentales pour les simulations physiques en temps-réel

Nicolas LEFEBVRE

RÉSUMÉ

Les engins de simulation physiques sont au coeur d'un large éventail d'applications et doivent faire face à différents défis en fonction du contexte dans lequel ils sont utilisés. La formation en réalité virtuelle (RV) pour la conduite d'équipement lourd présente souvent des scénarios impliquants des interactions entre des objets ayant des rapports de masse importants et des contraintes rigides, comme une charge lourde soulevée par un fil d'acier. Pour avoir de la valeur en tant qu'outil de formation, ces simulateurs doivent effectuer des calculs très précis tout en gérant les interactions utilisateur et ce sous une contrainte de performance très stricte. Bien que rapides, les solveurs linéaires itératifs fonctionnent souvent mal dans de tels cas, conduisant à des simulations imprécises ou instables. Les solveurs utilisant des méthodes qui impliquent une factorisation de la matrice système sont préférées. Cependant, la factorisation a un coût de calcul important qui peut réduire les performances. Dans ce travail, nous présentons un solveur de système linéaire efficace pour un système avec des contraintes physiques rigides et des contacts avec friction, modélisé comme un problème de complémentarité linéaire mixte (MLCP). Notre méthode est basée sur un algorithme de pivot de Guass par block et réutilise les factorisations précédentes en appliquant des mises à jour de rang un à chaque étape de pivotement. Les performances sont davantage améliorées en exploitant l'étroitesse de la bande de la matrice principale. Nous analysons le gain de performance dans divers scénarios difficiles, certains allant jusqu'à 3,5 fois plus vite par rapport au recalcul de la factorisation à partir de zéro. Nous explorons aussi la possibilité d'accélérer notre méthode en mettant en cache les factorisations intermédiaires.

Mots-clés: simulation physique en temps-réel, pivot de Gauss par blocs, complémentarité linéaire, dynamique, Cholesky, cache

Block Principal Pivoting with Incremental Cholesky Factorizations for Real-time Multibody Simulations

Nicolas LEFEBVRE

ABSTRACT

Physics engines are at the heart of a wide array of applications and must deal with different challenges depending on the context in which they are used. Virtual reality (VR) training for heavy equipment operation often simulates scenarios involving interactions between elements with large mass ratios and stiff constraints, like a heavy weight lifted by a steel wire. To have any value as a training tool, simulators must must perform accurate simulations while handling arbitrary user input under very strict performance constraints. Iterative linear solvers, while being fast to compute approximate solutions, often perform poorly in such cases leading to inaccurate or unstable simulations, and so direct methods involving a factorization of the system matrix are preferred. However, the factorization has a significant computational cost that can reduce performance. In this work, we present an efficient linear solver for systems with stiff physical constraints and contacts, where the dynamics are modeled as a mixed linear complementarity problem (MLCP). Our method is based on a block principal pivoting (BPP) algorithm, and at each step previous factorizations are reused by applying low-rank downdates at each pivoting step. We obtain further performance improvements by exploiting the low bandwidth characteristics of the lead matrix. We analyze the performance gain in various challenging scenarios, some of which gained up to a 3.5× speed-up when compared to recomputing the factorization from scratch. We further explore the possibility of accelerating our method by caching intermediary factorizations.

Keywords: real-time physics simulation, block principal pivoting, LCP, multibody dynamics, Cholesky, caching

TABLE OF CONTENTS

Page

INTRODUCTION 1

CHAPTER 1 BACKGROUND AND RELATED WORK 5
1.1 Rigid body dynamics 5
1.1.1 Kinematics 5
1.1.2 Equations of motion 7
1.2 Kinematic constraints 8
1.2.1 Constrained equations of motion 10
1.2.2 Linear complementarity problem 12
1.3 Numerical Methods 14
1.3.1 Iterative methods 14
1.3.2 Pivoting methods 15
1.3.3 Hybrid methods 17
1.3.4 Low-rank matrix updates 17

CHAPTER 2 EFFICIENT BLOCK PIVOTING 19
2.1 Block principal pivoting 19
2.2 Modified BPP algorithm 21
2.2.1 Downdating the Cholesky factorization 22
2.2.2 Analysis of index sets 24
2.2.3 Skyline coding 27
2.3 Results 29
2.3.1 Examples 30
2.3.1.1 Forklift 30
2.3.1.2 Mobile crane 31
2.3.1.3 Vehicle parkour 31
2.3.1.4 Offshore 31
2.3.1.5 Forwarder 31
2.3.1.6 Winch 32
2.3.2 Performance comparison 32

CHAPTER 3 FACTORIZATION CACHING 37
3.1 Caching data structure 37
3.2 Cached efficient BPP 39
3.3 Experimentation 40
3.4 Discussion 42

CONCLUSION 45
4.1 Future work 46

BIBLIOGRAPHY 48

LIST OF TABLES

Page

Table 2.1 Simulation parameters used in our experiments 30

Table 2.2 Mass ratio, constraint count, and condition numbers for our examples 32

Table 2.3 Average dynamics solve time and speed-up for the test scenarios 33

Table 2.4 Numerical error introduced by downdating 33

Table 3.1 Caching algorithm statistics 42

LIST OF FIGURES

Page

Figure 1.1 Coulomb friction cone 13

Figure 2.1 Effect of pivoting the ith column of a Cholesky factorization 24

Figure 2.2 Examples simulated using our efficient block pivoting method 25

Figure 2.3 Percentage of variables in matrix $\mathbf{A}$ that pivot into the tight set $\mathbb{T}$ 26

Figure 2.4 Speed-up using low-rank downdates 26

Figure 2.5 Fill-in pattern of the system matrices 27

Figure 2.6 Visualization of the skyline data structure 28

Figure 2.7 Time to solve the constrained dynamics equations for our examples 34

Figure 3.1 Caching algorithm performance 41

Figure 3.2 Caching algorithm memory usage 41

Figure 4.1 Eliminations trees for different permutations of the lead matrix 46

LIST OF ALGORITHMS

Page

Algorithm 2.1 Block principal pivoting (Júdice & Pires, 1994) 21

Algorithm 2.2 Efficient block principal pivoting 22

Algorithm 2.3 Modified downdating procedure for a $\mathbf{LL}^\intercal$ factorization 23

Algorithm 2.4 Direct solver using an $\mathbf{LL}^\intercal$ Cholesky decomposition 29

Algorithm 2.5 Downdating an $\mathbf{LL}^\intercal$ factorization with skyline 35

Algorithm 2.6 Modified $\mathbf{LL}^\intercal$ Cholesky decomposition with skyline 36

Algorithm 3.1 Modified pivoting algorithm using caching 38

Algorithm 3.2 Procedure used to find the cache entry 39

LIST OF ABBREVIATIONS

BPP	block principal pivoting
DoF	degree of freedom
LCP	linear complementarity problem
MDR	minimum degree reordering
MLCP	mixed linear complementarity problem
NLCP	non-linear complementarity problem
ODE	Open Dynamics Engine
PGS	Projected Gauss-Seidel
RAM	random-access memory
RCM	reverse Cuthill-McKee
VR	virtual reality

LIST OF SYMBOLS AND UNITS OF MEASUREMENTS

$\mathbf{A}$	Lead matrix
$\mathbf{a}$	Generalized acceleration vector
$\mathbf{b}$	Right hand side of the constrained equation of motion
$\mathbf{C}$	Compliance matrix
$\mathbb{C}$	Cache data structure; Set of ordered pairs $\{\mathbb{K}, \mathbf{L}\}$
$\mathbb{F}$	Index set of free variables
$\mathbf{f}$	Generalized force vector
h	Time step (seconds)
$\mathbf{J}$	Constraint gradient matrix
$\mathbb{K}$	Index set of pivoted variables in $\mathbf{L}$
$\mathbf{L}$	Lower triangular matrix obtain from Cholesky factorization
$\boldsymbol{\lambda}$	Constraint forces vector
$\mathbf{M}$	Mass matrix
m	Number of constraints
n	Number of bodies
$\boldsymbol{O}$	Bachmann–Landau asymptotic notation
$\boldsymbol{\omega}$	Angular velocity vector; $\dot{\boldsymbol{\theta}}$
$\mathbf{p}$	Position vector
$\boldsymbol{\phi}$	Constraint functions

$\mathbf{q}$	Generalized coordinates vector
$\mathbf{s}$	Skyline information vector
$\mathbb{T}$	Index set of tight variables
$\boldsymbol{\theta}$	Orientation quaternion
$\mathbf{v}$	Generalized velocities vector
$\mathbf{w}$	Residual velocities vector
$\dot{\square}$	First order derivative with respect to time
$\ddot{\square}$	Second order derivative with respect to time
$\square^{+}$	Implicit value determined at the end of an integration step

INTRODUCTION

Physics simulations are used in a wide array of applications such as virtual prototyping, robot learning, training simulators, digital production, video games, weather prediction and many more. Some applications, such as weather prediction, require very precise simulation for the output to have any value. Whereas for other applications, such as video games, the priority is that the simulations achieve real-time frame rates and produce approximate, but plausible, physical behaviour. Yet other applications, such as robot learning, virtual reality (VR) and training simulators, require simulations that are both fast, stable and accurate. Typically in all of these applications, there is a conflict between precision and performance that must be considered, and simulation developers must decide where to make trade-offs.

In this thesis, we are interested in interactive VR simulations for training people on heavy machinery. These simulations are challenging since they involve rigid bodies with large mass ratios and stiff constraints that can lead to issues with stability (Andrews, Teichmann & Kry, 2017; Tournier, Nesme, Gilles & Faure, 2015). These simulations also have very tight performance constraints and must be capable of handling arbitrary input conditions due to a human-in-the-loop.

Motivation

At the heart of every physics engine is the linear solver that is responsible for determining the constraint forces and velocities that produce realistic behaviour for bodies in a physics simulation. Iterative solvers are popular for interactive physics simulations, and implementations of these types of solvers can be found in many off-the-shelf rigid body physics simulators (Coumans, 2019; Havok, 2019; Smith et al., 2005). While they produce plausible behaviour in a timely manner, iterative solvers often produce approximate rather than accurate solutions of the underlying physical model. In some challenging cases, they cannot guarantee convergence to a reasonable solution

Conversely, direct solver methods are well-suited to handle numerically challenging linear systems and produce accurate solutions to the constrained equations of motion. However, the disadvantage is that direct solver algorithms have a higher computational complexity compared to iterative methods. For instance, solvers using a Cholesky factorization have a worst case complexity of $\boldsymbol{O}(n^3)$ where n is the number of system variables, and for the pivoting solvers used by our research partner, CM Labs Simulations, the lead matrix often needs to be refactorized multiple times per simulation frame. This can mean that they take more time per frame to produce a solution, thus impacting the real-time frame rate requirements of our target applications.

Research Problem

In this work, we are principally motivated by speeding up complex multibody simulations involving contacts and stiff constraints in the CM Labs' physics engine Vortex. Their engine, developed for training simulators, needs to produce extremely precise results while keeping the simulation running smoothly. Furthermore, to assure the accuracy of the simulation, Vortex uses a direct solver which deals gracefully with simulations involving stiffs constraints in joints and large mass ratios. We aim to accelerate their direct solver by reducing the computation overhead of factorizing a problem lead matrix, specifically, by reusing previous factorizations within a time step.

Thesis Organization

The remainder of this thesis is organized as follows:

- **Chapter 1** covers the theoretical background on which our methodology is based, and reviews related literature in the area of multibody dynamics, contact simulation, and linear solvers;

- **Chapter 2** presents our algorithm for efficient block principal pivoting (BPP) that uses an incremental downdating strategy to reduce overhead associated with computing the Cholesky factorization;
- **Chapter 3** presents experiments on our attempts to further improve the efficiency of the algorithm in Chapter 3 by caching previously computed factorization;
- **Conclusion** summarizes the contributions and presents possible directions for future research on efficient solvers for rigid body physics solvers.

Published work

The contents of Chapter 2 are largely derived from the article *Efficient block pivoting for multibody simulations with contact* (Enzenhoefer *et al.*, 2019) that was published and presented at the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D 2019). The contributions of the author of this thesis to this work include the development of the modified block pivoting algorithm in C++, as well as the analysis and experimentation of the modified algorithm to demonstrate its viability for the target examples, and finally writing key sections of the paper.

CHAPTER 1

BACKGROUND AND RELATED WORK

In this chapter, a theoretical background on constrained multibody dynamics and the contact models used in our physics simulations is presented. Additionally, details of the algorithms for solving the related LCPs are briefly explained. The chapter also covers related work and methodologies that have addressed the problem of simulating rigid bodies with contact.

1.1 Rigid body dynamics

The simulations targeted in this thesis involve rigid bodies. A rigid body is an idealized solid whose size and shape are invariant when forces are applied to it. Because the body is not being subject to deformation, the distance between any pair of points within the rigid body stays constant.

We can describe the 3D kinematics of rigid bodies using a single position and orientation, along with their derivatives with respect to time, which are the linear and angular velocities. The reference point for the positions and velocities generally coincides with the center of mass of the rigid body, although any fixed point on the solid can be use for this purpose.

An in-depth explanation of rigid body dynamics with contact and related methods can be found in the state-of-the-art report by Bender, Erleben & Trinkle (2014). In the following sections, details leading up to the specific constrained dynamics formulation and solver methods used by the Vortex dynamics engine (CM Labs Simulations, 2018) are presented.

1.1.1 Kinematics

In three dimensions, the motion of an unconstrained rigid body can be described by six parameters known as the degrees of freedom (DoFs). These six parameters correspond to the three linear coordinates of the body, giving its position, and the three angular coordinates, giving

its orientation. A body's position in space, $\mathbf{p} \in \mathbb{R}^3$, is described as a displacement within a global reference frame along three linearly independent axes x, y, z, such that

$$\mathbf{p} = [p_x, p_y, p_z]^\mathsf{T} \ . \tag{1.1}$$

The body's orientation, $\boldsymbol{\theta} \in \mathbb{R}^4$, is given by a quaternion, such that

$$\boldsymbol{\theta} = [\theta_w, \theta_x, \theta_y, \theta_z]^\mathsf{T} \ . \tag{1.2}$$

Observe that Equation 1.2 gives a 4-tuple of values, yet only 3 degree of freedoms (DoFs) are required to describe the angular motion of a body in 3D space. However, one degree of freedom is removed by the constraint that all rotation quaternions have unit length, such that $\|\boldsymbol{\theta}\| = 1$. Please see the seminal work by Shoemake (1985) for further details on using quaternions to animate orientations in graphics.

Combining the position and orientation, the rigid body's configuration in 3D space can be described by a vector $\mathbf{q} \in \mathbb{R}^7$ that concatenates the two:

$$\mathbf{q} = [\mathbf{p}^\mathsf{T} \ \boldsymbol{\theta}^\mathsf{T}]^\mathsf{T} \ . \tag{1.3}$$

In a physics simulation, the position and orientation change with time and are therefore functions of the simulation time t. Computing the first derivative of $\mathbf{p}(t)$ and $\boldsymbol{\theta}(t)$ with respect to t gives expressions for the linear and angular velocities, $\dot{\mathbf{p}} \in \mathbb{R}^3$ and $\boldsymbol{\omega} \in \mathbb{R}^3$ respectively, such that

$$\begin{aligned} \dot{\mathbf{p}}(t) &= \frac{\delta \mathbf{p}(t)}{dt} \\ \boldsymbol{\omega}(t) &= \mathbf{T}\frac{\delta \boldsymbol{\theta}(t)}{dt} \ . \end{aligned} \tag{1.4}$$

Note that computing the angular velocities from $\frac{d\boldsymbol{\theta}(t)}{dt}$ requires a kinematic mapping $\mathbf{T} \in \mathbb{R}^{3x4}$ due to the orientations using quaternions. A description of this mapping and an in-depth explanation can be found in Bender *et al.* (2014).

Both linear and angular velocities for a body may be combined to give the generalized velocity of the body as $\mathbf{v}(t) = [\dot{\mathbf{p}}(t)^\intercal \boldsymbol{\omega}(t)^\intercal]^\intercal$. Likewise, computing the second derivative with respect to time, and assuming that $\dot{\mathbf{T}} = 0$, gives the generalized acceleration:

$$\mathbf{a}(t) = [\ddot{\mathbf{p}}(t)^\intercal \ \dot{\boldsymbol{\omega}}(t)^\intercal]^\intercal \ . \tag{1.5}$$

For a system consisting of n rigid bodies, we can redefine the kinematic vectors by simply concatenating the quantities for each individual body into a global vector for each of the positions, velocities, and accelerations of the system, such that

$$\begin{aligned} \mathbf{q}(t) &= [\mathbf{q}_1{}^\intercal(t) \ \ldots \mathbf{q}_n{}^\intercal(t)]^\intercal \in \mathbb{R}^{7n} \\ \mathbf{v}(t) &= [\mathbf{v}_1{}^\intercal(t) \ \ldots \mathbf{v}_n{}^\intercal(t)]^\intercal \in \mathbb{R}^{6n} \\ \mathbf{a}(t) &= [\mathbf{a}_1{}^\intercal(t) \ \ldots \mathbf{a}_n{}^\intercal(t)]^\intercal \in \mathbb{R}^{6n} \ . \end{aligned} \tag{1.6}$$

1.1.2 Equations of motion

The Newton-Euler equations of motion determine the acceleration of free bodies based on their mass and forces acting on the bodies. This is a form of Newton's second law, which for a system of n bodies may be written as

$$\mathbf{f}(t) = \mathbf{M}\mathbf{a}(t) \ . \tag{1.7}$$

Here $\mathbf{a}(t) \in \mathbb{R}^{6n}$ contains the acceleration of all system bodies, $\mathbf{f}(t) \in \mathbb{R}^{6n}$ are the generalized forces, and $\mathbf{M} \in \mathbb{R}^{6n \times 6n}$ is the mass matrix that encodes their mass and inertia.

Physical simulations for computer graphics are typically performed using a discrete numerical integration scheme with time step h. To express Equation 1.7 in terms of velocities, we use the Taylor series approximation $\mathbf{v}(t+h) \approx \mathbf{v}(t) + h\mathbf{a}(t)$. For this approximation to be plausible, we want the time step h to be small. All of our examples use a time step $h = 1/60$ s, and since the errors introduced by the linear approximation is of the order $\boldsymbol{O}(h^2)$, we can therefore ignore it.

Defining velocities at the end of a time step as $\mathbf{v}^+ = \mathbf{v}(t+h)$, we drop the $\cdot(t)$ notation as a function of time and arrive at the following approximation of the accelerations

$$\mathbf{a} = \frac{\mathbf{v}^+ - \mathbf{v}}{h} \,. \tag{1.8}$$

Substituting Equation 1.8 into Equation 1.7 gives the velocity level equations of motion at a specific time step as

$$\mathbf{M}(\mathbf{v}^+ - \mathbf{v}) = h\mathbf{f} \,. \tag{1.9}$$

1.2 Kinematic constraints

Meaningful physics simulations involve interactions between bodies. For instance, rigid bodies should not interpenetrate, and rigid bodies generate friction when they slide against each other. These rules, or *constraints*, effectively restrict how bodies in a simulation are allowed to move relative to each other. They may be used to model articulations, such as hinges and ball-and-socket joints, but also complex mechanisms, such as chains, cables and even vehicles.

Each constraint equation ϕ is a scalar function that computes the violation, or error, of constraint for a specific constrained degree of freedom. It is typically computed as a function of body positions and orientation, but since the configuration of bodies is dynamic, constraint equations also depend on time t. More specifically, bilateral constraints are described by equalities of the form:

$$\phi(\mathbf{q}, t) = 0 \tag{1.10}$$

They are often used to model the behaviour of joints, such as hinges, ball-and-socket, and prismatics.

Whereas unilateral constraints take the form of an inequality:

$$\phi(\mathbf{q}, t) \geq 0 \tag{1.11}$$

Bender *et al.* (2014) present a constraint classification and detailed examples in their state-of-the-art paper. Equation 1.10 and Equation 1.11 are position level constraint equations. However, since we intend to use the velocity level formulation of the rigid body dynamics presented in Equation 1.9 to perform simulations, it is more convenient to derive a version of the constraint equations in terms of the velocities. We will explain how this is done for a system of m constraint equations:

$$\boldsymbol{\phi}(\mathbf{q}, t) = \begin{bmatrix} \phi_1(\mathbf{q}, t) & \dots & \phi_m(\mathbf{q}, t) \end{bmatrix}^{\mathsf{T}}$$

By D'Alembert's principle (Lanczos, 2012), the constraint equations will not be violated as long as no virtual work is done in the direction of the constraint gradients, $\frac{\partial \boldsymbol{\phi}}{\partial \mathbf{q}}$. Further note that the gradient may be expressed as a mapping of the body velocities by applying the chain rule, such that

$$\dot{\boldsymbol{\phi}}(\mathbf{q}, t) = \frac{d\boldsymbol{\phi}}{dt} = \frac{\partial \boldsymbol{\phi}}{\partial \mathbf{q}} \frac{d\mathbf{q}}{dt} \,. \tag{1.12}$$

Noting that $\mathbf{v} = \frac{\partial \mathbf{q}}{\partial t}$, the velocity level bilateral constraint equations can be stated as

$$\mathbf{J}\mathbf{v}^{+} = 0 \,, \tag{1.13}$$

where $\mathbf{J} = \frac{\partial \boldsymbol{\phi}}{\partial \mathbf{q}} = \begin{bmatrix} \frac{\partial \boldsymbol{\phi}_1}{\partial \mathbf{q}} & \frac{\partial \boldsymbol{\phi}_2}{\partial \mathbf{q}} & \dots & \frac{\partial \boldsymbol{\phi}_m}{\partial \mathbf{q}} \end{bmatrix} \in \mathbb{R}^{m \times 6n}$ is the *Jacobian matrix* containing gradient information at configuration $\mathbf{q}$ for all m constraint equations in the multibody system.

Although Equation 1.13 can be solved accurately, it is a linear approximation of the constraint behaviour at a particular configuration, and because numerical errors will inevitably accumulate, there is no guarantee that a numerical solution for Equation 1.13 will produce error free

simulations. To this end, *constraint stabilization* is used to correct constraint errors due to numerical drift at each time step. A Baumgarte-style stabilization is used to correct constraint errors (Cline & Pai, 2003), such that

$$\mathbf{J}\mathbf{v}^{+} = -\frac{\boldsymbol{\phi}}{h} \,. \tag{1.14}$$

Intuitively, Equation 1.14 gives a linear feedback term that attempts to "undo" any constraint error in the system left-over from the previous time step.

1.2.1 Constrained equations of motion

The method of Lagrange multipliers is used to enforce the constraints in the multibody simulation. This approach computes a multiplier λ that represents the magnitude of a force that acts in the direction of the constraint gradient $\mathbf{J}$. The constraint forces are computed in the space of generalized coordinates of the bodies as $\mathbf{J}^{\top}\boldsymbol{\lambda}$. Intuitively, the constraint forces will work to resolve constraint errors at the end of the time step, and since they are physical quantities, they can be included in the equations of motion as impulses

$$\mathbf{M}\mathbf{v}^{+} - h\mathbf{J}^{T}\boldsymbol{\lambda} = h\mathbf{f} + \mathbf{M}\mathbf{v} \,. \tag{1.15}$$

Rewriting Equation 1.15 and Equation 1.14 in block matrix form gives the constrained equations of motion

$$\begin{bmatrix} \mathbf{M} & -\mathbf{J}^{\top} \\ \mathbf{J} & 0 \end{bmatrix} \begin{bmatrix} \mathbf{v}^{+} \\ h\boldsymbol{\lambda}^{+} \end{bmatrix} = \begin{bmatrix} \mathbf{M}\mathbf{v} + h\mathbf{f} \\ -\frac{\boldsymbol{\phi}}{h} \end{bmatrix} \,. \tag{1.16}$$

The system in Equation 1.16 is potentially ill-conditioned, which occurs often during complex contact simulations. However, a common technique to improve the numerical properties of the linear system is adding positive coefficients along the diagonal of the lower-right block. These

coefficients act to regularize the linear system and permit the use of numerical methods that rely on having a full-rank linear system. Cline & Pai (2003) further demonstrate how these coefficients may be chosen to "soften" constraints, behaving much like a compliance for the constraint forces. Physics engines such as the Open Dynamics Engine (Smith *et al.*, 2005) also call this *constraint force mixing*, since constraint forces are added to the constraint equations.

This leads to a modified linear system

$$\begin{bmatrix} \mathbf{M} & -\mathbf{J}^\top \\ \mathbf{J} & \mathbf{C} \end{bmatrix} \begin{bmatrix} \mathbf{v}^+ \\ h\boldsymbol{\lambda}^+ \end{bmatrix} = \begin{bmatrix} \mathbf{M}\mathbf{v} + h\mathbf{f} \\ -\frac{\boldsymbol{\phi}}{h} \end{bmatrix}, \tag{1.17}$$

where $\mathbf{C} \in \mathbb{R}^{m \times m}$ is the diagonal matrix of positive coefficients. The linear system can be further simplified by using the Schur complement of the upper left block

$$\underbrace{(\mathbf{C} + \mathbf{J}\mathbf{M}^{-1}\mathbf{J}^\top)}_{\mathbf{A}} h\boldsymbol{\lambda}^+ = \underbrace{-(\frac{\boldsymbol{\phi}}{h} + h\mathbf{J}\mathbf{M}^{-1}\mathbf{f} + \mathbf{J}\mathbf{v})}_{\mathbf{b}}, \tag{1.18}$$

that results in a reduced linear system in which only the constraint impulses $h\boldsymbol{\lambda}^+ \in \mathbb{R}^m$ need to be solved. The generalized velocities can later be recovered by substituting the constraint impulses into the first line of Equation 1.17 and directly computing $\mathbf{v}^+ = \mathbf{M}^{-1}(\mathbf{M}\mathbf{v} + h\mathbf{f} + h\mathbf{J}^\top\boldsymbol{\lambda}^+)$. Note that the block diagonal form of $\mathbf{M}$ makes it trivial to invert, and furthermore since $\mathbf{C}$ is a diagonal matrix with only positive values, the matrix $\mathbf{A}$ is positive definite and symmetric.

1.2.2 Linear complementarity problem

Since our simulations involve contact, some of the constraints in Equation 1.18 are used to model non-interpenetration and friction between bodies. These special type of constraints require that we introduce *feasibility* and *complementarity* conditions on the values of $\boldsymbol{\lambda}^+$. Feasibility means that some variables have upper and lower bounds on the range of acceptable values, i.e. $\boldsymbol{\lambda}_{\text{lo}} \leq \boldsymbol{\lambda}^+ \leq \boldsymbol{\lambda}_{\text{hi}}$. Whereas complementarity means that velocity must be considered when solving for certain constraint impulses, which requires introducing a vector of residual velocities $\mathbf{w} \in \mathbb{R}^m$. The "slack" variable $\mathbf{w}$ is further decomposed into non-negative components, i.e. $\mathbf{0} \leq \mathbf{w}_+ \perp \mathbf{w}_- \geq \mathbf{0}$, giving the linear system

$$\mathbf{A}\boldsymbol{\lambda}^+ - \mathbf{b} = \mathbf{w}_+ - \mathbf{w}_- \ . \tag{1.19}$$

Finally, combining feasibility and complementarity conditions gives the mixed linear complementarity problem (MLCP)

$$\mathbf{A}\boldsymbol{\lambda}^{+} - \mathbf{b} = \mathbf{w} = \mathbf{w}_{+} - \mathbf{w}_{-} \tag{1.20a}$$

$$\begin{cases} \mathbf{0} \le \mathbf{w}_{+} \perp (\boldsymbol{\lambda}^{+} - \boldsymbol{\lambda}_{\mathrm{lo}}) \ge \mathbf{0} \\ \mathbf{0} \le \mathbf{w}_{-} \perp (\boldsymbol{\lambda}_{\mathrm{hi}} - \boldsymbol{\lambda}^{+}) \ge \mathbf{0} \end{cases}, \tag{1.20b}$$

where $\boldsymbol{\lambda}_{\mathrm{hi}}$ and $\boldsymbol{\lambda}_{\mathrm{lo}}$ are upper and lower bounds on the constraint forces, respectively.

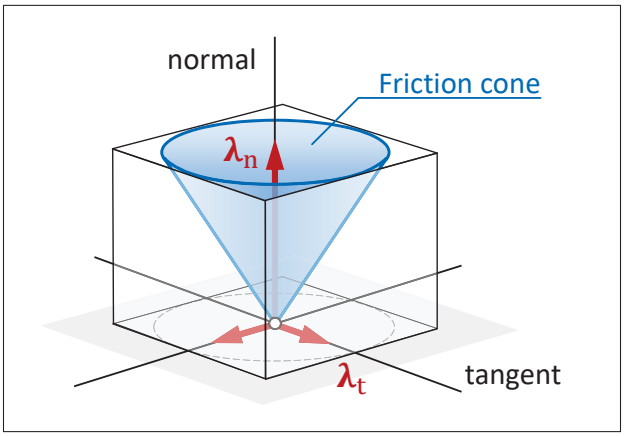

Figure 1.1 Coulomb friction cone

Frictional contact in rigid body simulations is often modeled using a Coulomb cone (see Figure 1.1), and constraint forces are physically valid only if they lie in the boundaries defined by the cone. For instance, in the case of non-interpenetration acting perpendicular to the contact surface, the constraint forces can only push bodies apart, and so $\boldsymbol{\lambda}_{lo} = 0$ and $\boldsymbol{\lambda}_{\mathrm{hi}} = \infty$. Whereas for the lower and upper limits of the friction forces that act tangentially to the contact surface, limits are defined by the coefficient of friction times the normal forces, such that

$$\begin{aligned} \boldsymbol{\lambda}_{\mathrm{hi}} &= \mu\boldsymbol{\lambda}_{\mathrm{N}} \\ \boldsymbol{\lambda}_{\mathrm{lo}} &= -\mu\boldsymbol{\lambda}_{\mathrm{N}} \ , \end{aligned} \tag{1.21}$$

where μ is the static coefficient of friction and λ_N is the corresponding non-interpenetration force that is perpendicular to the collision surface.

1.3 Numerical Methods

Solving the MLCPs from Section 1.2.2 requires special numerical methods due to the feasibility and complimentarity conditions, and indeed our focus is on improving the efficiency of solvers for these types of problems. There are three classes of solvers for MLCPs covered: (i) iterative methods, (ii) pivoting based methods, and (iii) hybrid approaches that combine aspects of the first two types. Each of these is briefly discussed in the sections below, including the algorithm for which an extension is proposed in Chapter 2.

1.3.1 Iterative methods

Physics engines such as Bullet (Coumans, 2019) and Box2D (Catto, 2018) are popular among developers of interactive and real-time applications since they use iterative algorithms to solve for the constraint impulses. These algorithms have the benefit that they can produce an approximate solution quickly, and the algorithm may be terminated after only a small number of iterations.

Fixed-point algorithms, such as the Projected Gauss-Seidel (PGS) method (Erleben, 2007) and Sequential impulses (Guendelman, Bridson & Fedkiw, 2003), are popular due to their ease of implementation. However, previous work has noted that fixed-point algorithms converge slowly when the lead matrix is ill-conditioned, as is the case for many of our target examples, or when the solution is poorly initialized (Erleben, 2004). Poorly conditioned linear systems are typical in our target applications due to large mass ratios between bodies that are coupled by constraints. However, ill-conditioning may also be caused by numerical degeneracy when the regularization coefficients in $\mathbf{C}$ are very small.

Erleben (2007) developed a specialized PGS algorithm to treat large stacks, which are notorious for producing degenerate linear systems. The projected Jacobi method is a related algorithm and it has been popularized due to ease of parallelization, but is known for its poor convergence when

compared to PGS. Recent work has used Chebyshev polynomials (Wang, 2015) and a Nesterov momentum term (Mazhar, Heyn, Negrut & Tasora, 2015) to accelerate the convergence of this class of algorithms. Tonge, Benevolenski & Voroshilov (2012) introduced a mass splitting scheme for large scale parallel simulations on the GPU, whereas Fratarcangeli, Tibaldo & Pellacini (2016) applied a graph coloring scheme to parallelize PGS solvers and dramatically improve performance and convergence.

1.3.2 Pivoting methods

Pivoting methods try to find a partitioning of the system presented in Section 1.2 into *free* (basic) and *tight* (non-basic) variables. Unlike iterative methods, they do not make monotonic progress toward a solution and therefore terminating the algorithm early with an approximate solution is often not possible.

The free label indicates that a variable is within bounds, and thus its value is unknown, whereas the tight label indicates the value of the variable is equal to the bound, and is therefore assumed to be known. The labeling for all variables is referred to as the *index set*. If the index set changes between iterations or pivoting steps, the system matrix needs to be updated and its factorization must be recomputed if a direct linear solver is being used. Therefore, complex simulations requiring a large number of pivoting steps may be intractable for interactive applications. Unlike iterative methods, pivoting methods with a direct solver are able to find an exact solution to the linear complementarity problem (LCP).

One of the most well known pivoting methods is Lemke's algorithm (Cottle, Pang & Stone, 1993), which is a simplex algorithm wherein a single variable is treated at each iteration. The algorithm is guaranteed to converge if a solution exists to the LCP, but requires many steps for complex problems. An efficient version of Lemke's algorithm was developed by Lloyd (2005) and the algorithm achieves nearly linear complexity under the assumption of a fixed size problem, although typically a complexity of $\boldsymbol{O}(m^3)$ is expected. Dantzig's algorithm (Cottle & Dantzig, 1968) has been previously used by interactive computer graphics work (Baraff, 1994), and the

algorithm is offered as an alternative solver in the Open Dynamics Engine (ODE) toolkit (Smith *et al.*, 2005).

Baraff (1994) notes that an incremental factorization can be used for the inner pivoting loop, and thus an overall complexity close to $\boldsymbol{O}(m^2)$ is achievable. They hint that low-rank modifications of the LU factorization of the lead matrix are possible, but provide few details or analysis. However, in our work, we perform an in-depth analysis regarding performance and accuracy for a variety of complex 3D simulations, and motivate our approach using empirical evidence. Furthermore, our approach applies only a series of low-rank downdates to the Cholesky factorizaton (rather than both updating and downdating), which we found improves numerical stability.

Murty & Yu (1988) proposed a Bard-type pivoting method that preserves complementarity between variables at all times and changes the index sets for a single pair of complementary variables at each step (single pivoting). This algorithm is proven to converge for LCPs with positive definite lead matrices, although convergence is slow. Keller's method (Keller, 1973) preserves complementarity and nonnegativity of the LCP solution variables and is said to be more efficient for LCPs with large positive semi-definite lead matrices (Júdice, 1994).

Júdice & Pires (1994) introduced a block version of Murty's method which is much more efficient for large problems since it allows changing many index sets in a single algorithm iteration. This block version is proven to converge as it switches to single pivoting (Murty's method) if the number of variables out of bounds does not decrease in a user-defined number of iterations. However, in practice, the algorithm mostly performs block pivots and switches to single pivoting only for degenerate problems. An extensive survey of the most common pivoting methods can be found in Júdice (1994).

The BPP algorithm is used by the Vortex dynamics engine (CM Labs Simulations, 2018) due to its improved accuracy compared to iterative methods, and improved efficiency compared to single pivoting approaches. In Chapter 2, we present the BPP algorithm in further detail and propose an extension of the algorithm using low-rank adjustments to the Cholesky factorization of the lead matrix.

1.3.3 Hybrid methods

Direct linear solvers, involving a Cholesky or LU decomposition of the system matrix, are heavily utilized by pivoting methods when exact solutions are required. However, direct solvers have received much less attention from the computer graphics community. This is due mainly to the higher computational overhead associated with these solvers compared to iterative techniques. Lacoursière (2007) proposed a splitting method that iterates between a direct block pivoting solve for the combined sub-set of articulation and non-interpenetration constraints, followed by an iterative solve for non-interpenetration and friction constraints. This method guarantees accurate solutions for stiff joint and contact normal forces while compromising the accuracy of the friction forces in order to reach better performance than direct solvers. However, Enzenhoefer, Andrews, Teichmann & Kövecses (2018) noted problems with this approach for simulations where accurate friction forces are required.

In their article on subspace minimization, Silcowitz-Hansen, Niebe & Erleben (2010) formulate contact simulation as a non-linear complementarity problem (NLCP) to reduce the resources needed to resolve contact forces and to reduce artefacts, such as viscosity or overly "soft" contact responses caused by slow convergence. Briefly, they find an approximate value for a set of active contact constraints with PGS in order to find the active constraints and then use the conjugate gradient method to solve the active constraints. In the second phase, they update the linear system by excluding constraints at each iteration that reach a feasibility or complementarity limit.

1.3.4 Low-rank matrix updates

Our work relies on efficient low-rank modifications of a Cholesky factorization, specifically a row deletion corresponding to a rank-1 downdate. In other words, we change the lead matrix $\mathbf{A}$ by a modification of the form $\tilde{\mathbf{A}} = \mathbf{A} - \mathbf{u}\mathbf{u}^{\mathsf{T}}$, where $\mathbf{u}$ is computed such that it corresponds to a row removal operation (i.e., when a variable is pivoted from the free to tight index set).

Davis & Hager (2005) provide a thorough treatment of low-rank modifications for both dense and sparse Cholesky factorizations, and indeed we used theirs as a starting point for our work. Seeger (2004) also demonstrated that low-rank modifications to a Cholesky factorization can improve stability versus modifications to the system matrix using the Sherman-Morrison-Woodbury formula.

A method for low-rank modifications of the Cholesky factorization of a 3D mesh has recently been proposed Herholz & Alexa (2018). They re-use the factorization of the full mesh to efficiently perform operations on sub-meshes. Similar to their work, we use the so-called *left-looking algorithm* to compute our Cholesky factorizations, although in our case the domain is the constraints of a multibody system rather than a mesh operator.

CHAPTER 2

EFFICIENT BLOCK PIVOTING

In this chapter, we begin by providing algorithmic details of the BPP method proposed by Júdice & Pires (1994), which are covered in Section 2.1. We then explain how the Cholesky factorization used by the baseline BPP method may be modified using low-rank downdates in Section 2.2.1, and revise the baseline BPP algorithm to improve efficiency with a skyline data structure in Section 2.2.

2.1 Block principal pivoting

Recall that the BPP algorithm combines a fast block strategy with Murty's single principal pivoting algorithm (Murty & Yu, 1988), with the main difference being that BPP exchanges multiple variables simultaneously during changes to the index set. At each iteration in the algorithm, the constraint variables are labeled as belonging to either the free ($\mathbb{F}$) or tight ($\mathbb{T}$) index set. Tight variables have reached a lower or upper bound, and therefore their value is defined by the feasibility conditions, whereas free variables have not exceeded these bounds, and therefore $\mathbf{w}_i = 0, \forall\ i \in \mathbb{F}$. For instance, when a frictional contact is "sticking", the constraint impulse is within bounds, and thus free. However, once the contact begins sliding, the constraint impulse is determined by the limits of the friction model, and thus the index set changes to tight.

A *pivoting step* is the process of proposing a candidate solution and determining if the index set is correct. Based on an assumption of the index set, a candidate solution of the linear system in Equation 1.18 is computed. If the assumption is correct, all variables satisfy feasibility and complementarity conditions. If not, the index set of all variables i which do not satisfy the

conditions are changed, such that

$$i \in \begin{cases} \mathbb{F}, & \text{if } \boldsymbol{\lambda}_{\text{lo}} < \boldsymbol{\lambda}_i < \boldsymbol{\lambda}_{\text{hi}} \text{ and } \mathbf{w}_i = 0 \\ \mathbb{T}_{\text{lo}}, & \text{if } \boldsymbol{\lambda}_i \leq \boldsymbol{\lambda}_{\text{lo}} \text{ and } \mathbf{w}_i > 0 \\ \mathbb{T}_{\text{hi}}, & \text{if } \boldsymbol{\lambda}_i \geq \boldsymbol{\lambda}_{\text{hi}} \text{ and } \mathbf{w}_i < 0 \end{cases} . \tag{2.1}$$

The algorithm terminates with success if the index sets between two consecutive pivoting steps do not change.

Note that in Equation 2.1 the tight set is decomposed into both lower ($\mathbb{T}_{\text{lo}}$) and upper ($\mathbb{T}_{\text{hi}}$) bounded variables, and together these form the complete tight set, such that $\mathbb{T} = \mathbb{T}_{\text{lo}} \cup \mathbb{T}_{\text{hi}}$. The system matrix and right-hand side vector in Equation 1.18 can be regrouped according to the index set, such that

$$\begin{bmatrix} \mathbf{A}_{\mathbb{F}\mathbb{F}} & \mathbf{A}_{\mathbb{F}\mathbb{T}} \\ \mathbf{A}_{\mathbb{T}\mathbb{F}} & \mathbf{A}_{\mathbb{T}\mathbb{T}} \end{bmatrix} \begin{bmatrix} \boldsymbol{\lambda}_{\mathbb{F}} \\ \boldsymbol{\lambda}_{\mathbb{T}} \end{bmatrix} - \begin{bmatrix} \mathbf{b}_{\mathbb{F}} \\ \mathbf{b}_{\mathbb{T}} \end{bmatrix} = \begin{bmatrix} \mathbf{w}_{\mathbb{F}} \\ \mathbf{w}_{\mathbb{T}} \end{bmatrix} . \tag{2.2}$$

Observe that for tight variables the value of $\boldsymbol{\lambda}_{\mathbb{T}}$ is known, and that for free variables the residual velocity $\mathbf{w}_{\mathbb{F}} = 0$, by definition of the LCP. This means that we only need to solve for $\boldsymbol{\lambda}_{\mathbb{F}}$ at each step in the algorithm

$$\mathbf{A}_{\mathbb{F}\mathbb{F}}\boldsymbol{\lambda}_{\mathbb{F}} = \mathbf{b}_{\mathbb{F}} - \mathbf{A}_{\mathbb{F}\mathbb{T}}\boldsymbol{\lambda}_{\mathbb{T}} . \tag{2.3}$$

Assuming that the lead matrix is positive definite, which is true for the multi-body systems in our target simulations, a Cholesky factorization may be used to solve the linear system in Equation 2.3 where $\mathbf{A}_{\mathbb{F}\mathbb{F}} = \mathbf{L}_{\mathbb{F}\mathbb{F}}\mathbf{L}_{\mathbb{F}\mathbb{F}}^{\mathsf{T}}$ and $\mathbf{L}_{\mathbb{F}\mathbb{F}}$ is a lower triangle matrix. Pseudo-code for the BPP algorithm using a Cholesky factorization is given in Algorithm 2.1. Observe that the matrix $\mathbf{A}_{\mathbb{F}\mathbb{F}}$ must be refactored whenever the index sets change (i.e., at each step of the algorithm). This can be costly, since the Cholesky factorization can have a worst-case complexity of $\boldsymbol{O}(m^3)$.

Algorithm 2.1 Block principal pivoting (Júdice & Pires, 1994)

Block principal pivoting algorithm

Input: $\mathbf{A}, \mathbf{b}, \boldsymbol{\lambda}_{\text{hi}}, \boldsymbol{\lambda}_{\text{lo}}$
Output: $\mathbf{w}, \boldsymbol{\lambda}$

initialize $\mathbb{F}$ and $\mathbb{T}$
$k \leftarrow 0$
do
  $\mathbf{L}_{\mathbb{FF}} \leftarrow \textsc{Cholesky}(\mathbf{A}_{\mathbb{FF}})$
  $\forall\, i \in \mathbb{T}_{\text{lo}},\ \boldsymbol{\lambda}_i \leftarrow \boldsymbol{\lambda}_{\text{lo},i}$
  $\forall\, i \in \mathbb{T}_{\text{hi}},\ \boldsymbol{\lambda}_i \leftarrow \boldsymbol{\lambda}_{\text{hi},i}$
  solve $\mathbf{L}_{\mathbb{FF}}\mathbf{L}_{\mathbb{FF}}{}^{\mathsf{T}}\boldsymbol{\lambda}_{\mathbb{F}} = \mathbf{b}_{\mathbb{F}} - \mathbf{A}_{\mathbb{FT}}\boldsymbol{\lambda}_{\mathbb{T}}$
  $\mathbf{w} = \mathbf{A}\boldsymbol{\lambda} - \mathbf{b}$
  update $\mathbb{F}, \mathbb{T}$ according to Eq. 2.1
  $k \leftarrow k + 1$
while ($k <$ *max steps*) *and (index sets* $\mathbb{F}, \mathbb{T}$ *changed)*

However, if the tight set is small, a more efficient approach would be to modify the original factorization $\mathbf{A} = \mathbf{L}\mathbf{L}^{\mathsf{T}}$ using a series of low-rank updates to obtain $\mathbf{L}_{\mathbb{FF}}$, and this is precisely the approach we propose.

2.2 Modified BPP algorithm

Pseudo-code for our proposed modification to the BPP algorithm is provided in Algorithm 2.2. The primary contribution is that the Cholesky factorization is computed only once per simulation step, rather than once per iteration of the pivoting algorithm. Instead, at each pivoting step, we copy the initial factorization and apply a sequence of low-rank downdates to the matrix $\mathbf{L}$, one for each tight variable in the system. Constraint impulses $\boldsymbol{\lambda}_{\mathbb{T}}$ are known and may be transferred to the right-hand side of Equation 2.3. We therefore need to remove the rows and columns in $\mathbf{L}$ associated with these variables.

Algorithm 2.2 Efficient block principal pivoting

Efficient BPP algorithm

Input: $\mathbf{A}$, $\mathbf{b}$, $\boldsymbol{\lambda}_{\text{hi}}, \boldsymbol{\lambda}_{\text{lo}}$
Output: $\mathbf{w}$, $\boldsymbol{\lambda}$

procedure `SolveLCP` ($\mathbf{A}, \mathbf{b}, \boldsymbol{\lambda}_{hi}, \boldsymbol{\lambda}_{lo}$)
  $\mathbf{L}_0 \leftarrow$ `Cholesky`($\mathbf{A}$)
  $\mathbf{s} \leftarrow$ `Skyline`($\mathbf{L}_0$)
  initialize index sets $\mathbb{F}$ and $\mathbb{T}$
  $k = 0$
  do
    $\mathbf{L} = \mathbf{L}_0$
    for each $i \in \mathbb{T}$ **do**
      $\boldsymbol{\sigma} = \mathbf{L}_{i+1\ldots m,i}$
      $\mathbf{L} \leftarrow$ `Downdate`($i, \mathbf{L}, \boldsymbol{\sigma}, \mathbf{s}$)
    end – for
    $\forall\ i \in \mathbb{T}_{\text{lo}},\ \boldsymbol{\lambda}_i \leftarrow \boldsymbol{\lambda}_{\text{lo},i}$
    $\forall\ i \in \mathbb{T}_{\text{hi}},\ \boldsymbol{\lambda}_i \leftarrow \boldsymbol{\lambda}_{\text{hi},i}$
    $\boldsymbol{\lambda}_{\mathbb{F}} \leftarrow$ `CholSolve`($\mathbf{L}, \mathbf{b}_{\mathbb{F}} - \mathbf{A}_{\mathbb{F}\mathbb{T}}\boldsymbol{\lambda}_{\mathbb{T}}, \mathbb{F}, \mathbf{s}$)
    $\mathbf{w} = \mathbf{A}\boldsymbol{\lambda} - \mathbf{b}$
    update $\mathbb{F}, \mathbb{T}$ according to Eq. 2.1
    $k = k + 1$
  while ($k <$ *max steps*) *and (index sets* $\mathbb{F}, \mathbb{T}$ *changed)*
end – procedure

2.2.1 Downdating the Cholesky factorization

In a left-looking Cholesky factorization, a change to the ith column of the factorization affects only the sub-block $\tilde{\mathbf{L}}$, formed by the rows and columns with indices in the range $i + 1 \ldots m$ (see Figure 2.1). When a constraint variable pivots to the tight set $\mathbb{T}$, a downdate of the original factorization $\mathbf{L}$ can be performed by modifying only the entries in $\tilde{\mathbf{L}}$. In order to remove the row and the column corresponding to the ith variable, we downdate using the vector σ_i, and this process is repeated for each variable $i \in \mathbb{T}$ sequentially. The cost of recomputing the Cholesky factorization can therefore be reduced by copying the original factorization $\mathbf{L}$ and updating in-place the sub-matrix $\tilde{\mathbf{L}}$, eventually giving $\mathbf{L}_{\mathbb{F}\mathbb{F}}$ for indices in $\mathbb{F}$.

Algorithm 2.3 Modified downdating procedure for a $\mathbf{L}\mathbf{L}^\intercal$ factorization

Downdating procedure

Input: A valid decomposition $\mathbf{L}$
The index of the variable whose contribution is to be removed i
Output: The updated $\mathbf{L}$ decomposition

1 **procedure** `Downdate` $(\mathbf{L}, i)$
2 $\quad \beta \leftarrow 1$
3 $\quad \boldsymbol{\sigma}_i \leftarrow \mathbf{L}_{i+1\ldots m,i}$
4 $\quad$ **for** $j \leftarrow 1 \ldots (m-i)$ **do**
5 $\quad\quad k \leftarrow i + j$
6 $\quad\quad x \leftarrow \boldsymbol{\sigma}_j$
7 $\quad\quad y \leftarrow \mathbf{L}_{k,k}$
8 $\quad\quad z \leftarrow y^2 + \frac{x^2}{\beta}$
9 $\quad\quad \gamma \leftarrow \beta\, y^2 + x^2$
10 $\quad\quad \mathbf{L}_{k,k} \leftarrow \sqrt{z}$
11 $\quad\quad \beta \leftarrow \beta + \frac{x^2}{y^2}$
12 $\quad\quad$ **if** $m - k > 0$ **then**
13 $\quad\quad\quad j_1 \leftarrow j + 1,\ j_2 \leftarrow m - i$
14 $\quad\quad\quad k_1 \leftarrow k + 1,\ k_2 \leftarrow m$
15 $\quad\quad\quad \boldsymbol{\sigma}_{j_1\ldots j_2} \leftarrow \boldsymbol{\sigma}_{j_1\ldots j_2} - \frac{x}{y}\mathbf{L}_{k_1\ldots k_2,k}$
16 $\quad\quad\quad \mathbf{L}_{k_1\ldots k_2,k} \leftarrow \mathbf{L}_{k_1\ldots k_2,k} + \frac{\sqrt{z}\,x}{\gamma}\boldsymbol{\sigma}_{j_1\ldots j_2}$
17 $\quad\quad$ **end**
18 $\quad$ **end**
19 $\quad$ **return** $\mathbf{L}$
20 **end – procedure**

The computation of the downdate vector is based on the dense row deletion algorithm given by Davis & Hager (2005), where vector $\boldsymbol{\sigma}_i = \mathbf{L}_{i+1\ldots m,i}$ consists of the rows below the diagonal of the ith column. Vector $\boldsymbol{\sigma}_i$ is then used to perform a rank-1 downdate of the lower right sub-block of the Cholesky factorization. Note that $\mathbf{L}$ is not resized during this process. Matrix elements are modified in place when the downdate is performed, and during the Cholesky solve in line 14 in Algorithm 2.2; rows and columns corresponding to tight variables are skipped. The DownDate routine in Algorithm 2.3 gives pseudo-code to downdate $\mathbf{L}$ for a single variable at index i. This routine is similar to low-rank modification routines which are commonplace in

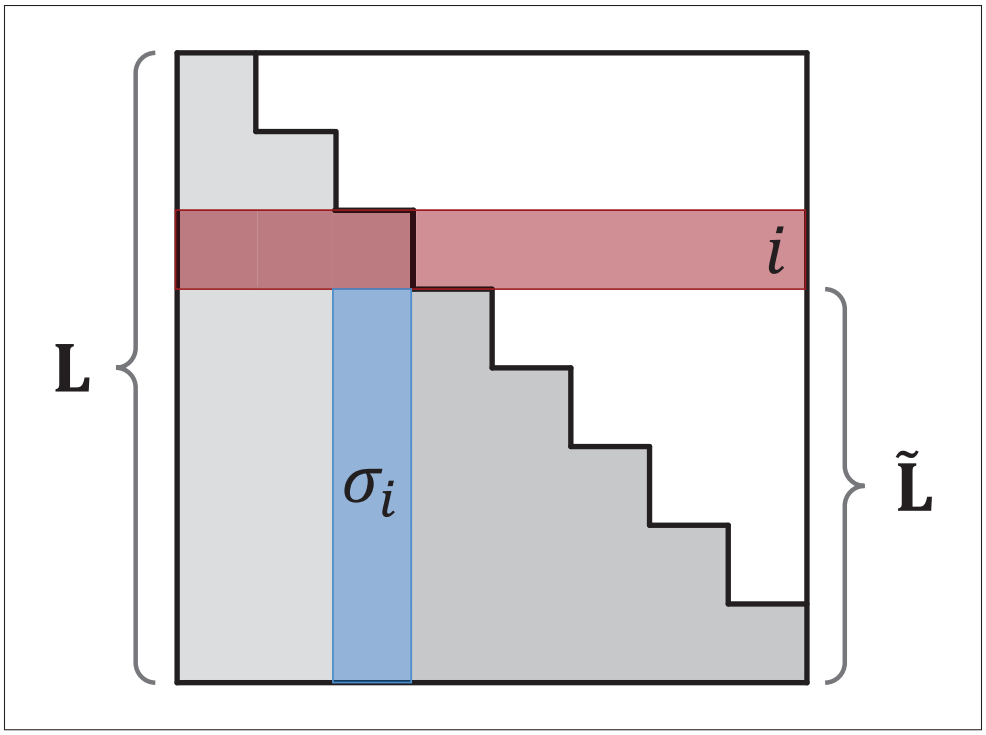

Figure 2.1 Effect of pivoting the *i*th column of a Cholesky factorization

linear algebra frameworks, e.g. Eigen (2018). However, we note that an important difference with our implementation is that only the sub-matrix corresponding to lower right block in Figure 2.1 is modified.

2.2.2 Analysis of index sets

In order to justify the use of a downdating scheme for our target simulations, we evaluated the index set behaviour across many time steps for several typical scenarios. Figure 2.2 presents a screenshot from each example simulated using our efficient block pivoting method: a mobile crane lifting a concrete pipe (top left); a winch spooling a chain (top center); a light armored vehicle driving over obstacles (top right); a knuckle boom crane on a ship lifting a subsea module (bottom right); a forklift lifting a crate and driving around cones (bottom center); a forwarder picking up a log (bottom right). Each one is modeled as a multibody system with hundreds of constraints, stiff contacts, and large mass ratios.

Figure 2.2 Examples simulated using our efficient block pivoting method

The evaluation shown in Figure 2.3 suggests that the size of $\mathbb{T}$ is often small compared to the total number of variables, even for complex scenarios. This indicates that, beginning from an initial factorization, only a small number of low-rank modifications are required to obtain $\mathbf{L}_{\mathbb{FF}}$.

As a next step, we performed a number of experiments on randomly generated symmetric positive-definite matrices of various sizes using a dense matrix storage in order to determine the threshold for which the downdating scheme becomes inefficient. We computed the Cholesky factorization $\mathbf{L}$ for the full matrix and randomly select variables to be pivoted into the tight set according to a pre-defined percentage of tight variables with respect to all variables. We then compared the time required to downdate $\mathbf{L}$ versus the time spent to recompute the factorization $\mathbf{A}_{\mathbb{FF}} = \mathbf{L}_{\mathbb{FF}}\mathbf{L}_{\mathbb{FF}}{}^{\mathsf{T}}$.

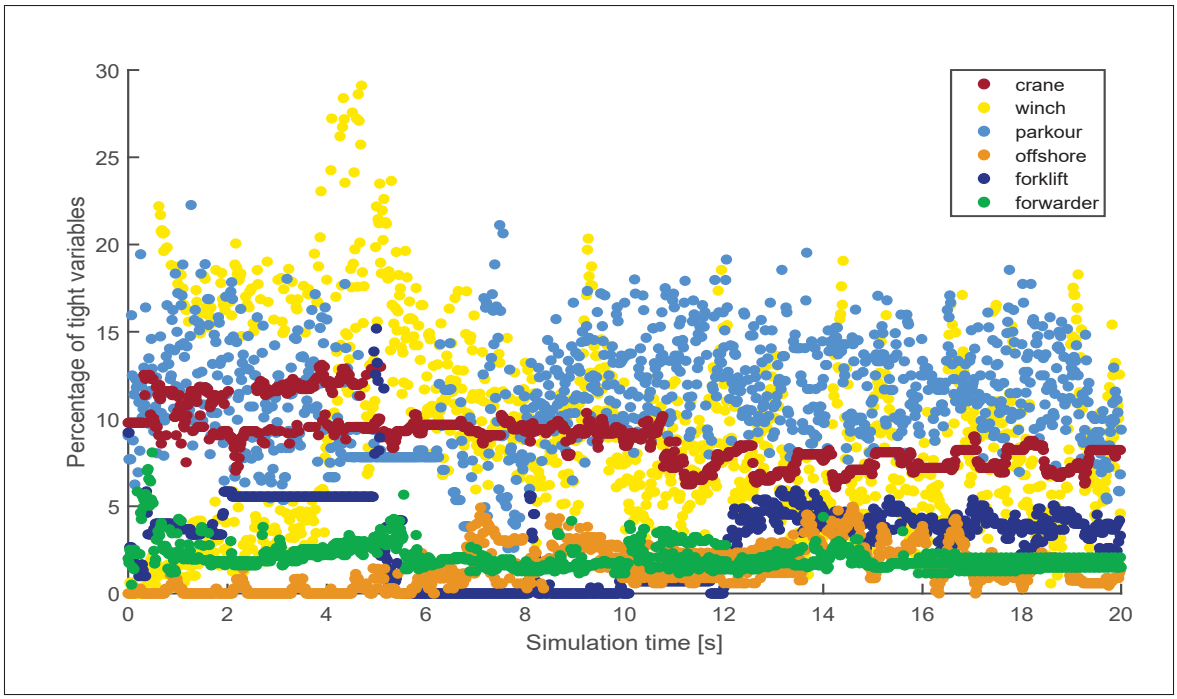

Figure 2.3 Percentage of variables in matrix $\mathbf{A}$ that pivot into the tight set $\mathbb{T}$

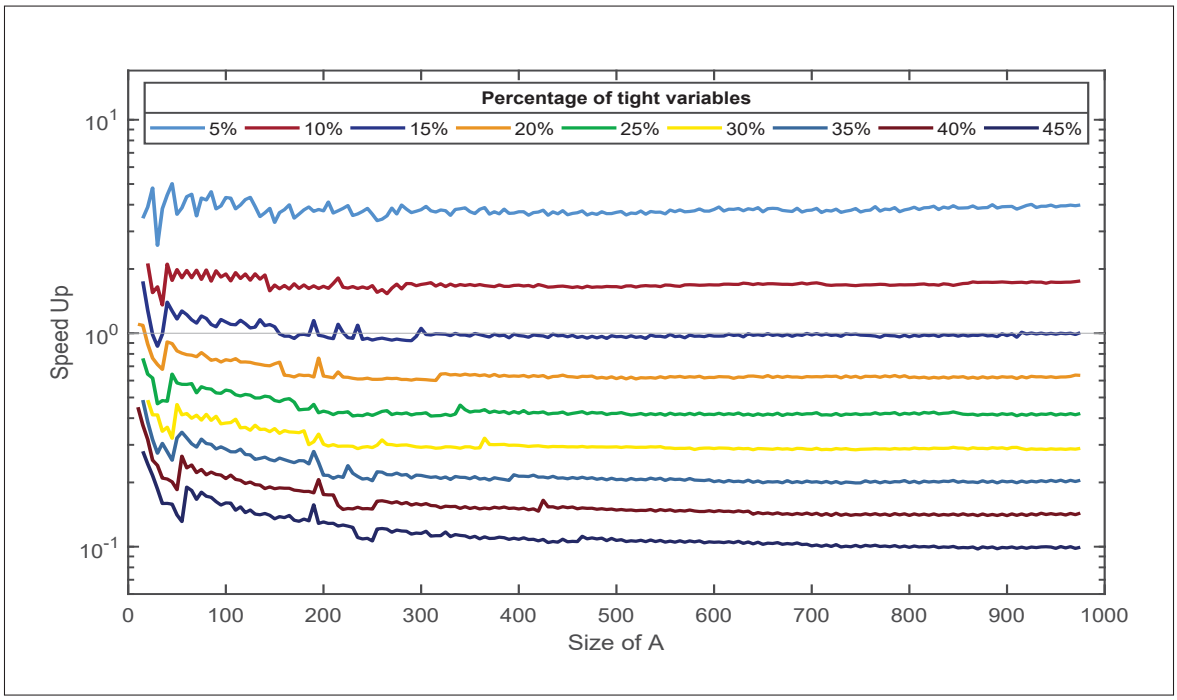

Figure 2.4 Speed-up using low-rank downdates

Figure 2.4 illustrates the speed gain obtained by using low-rank downdates instead of recomputing the Cholesky factorization of $\mathbf{A}_{\mathbb{FF}}$ with respect to the size of $\mathbf{A}$. Each curve corresponds to a different percentage of tight variables. It demonstrates that a speed-up can be realized when the percentage of tight variables is less than 15%. Revisiting Figure 2.3, we observe that this is the case for most of the simulation frames in our target examples. We also note that the speed-up factor remains nearly constant for a fixed percentage of tight variable even if the overall matrix size increases or decreases. Encouraged by these results, we proceeded to develop a modified version of the BPP algorithm incorporating low-rank downdates.

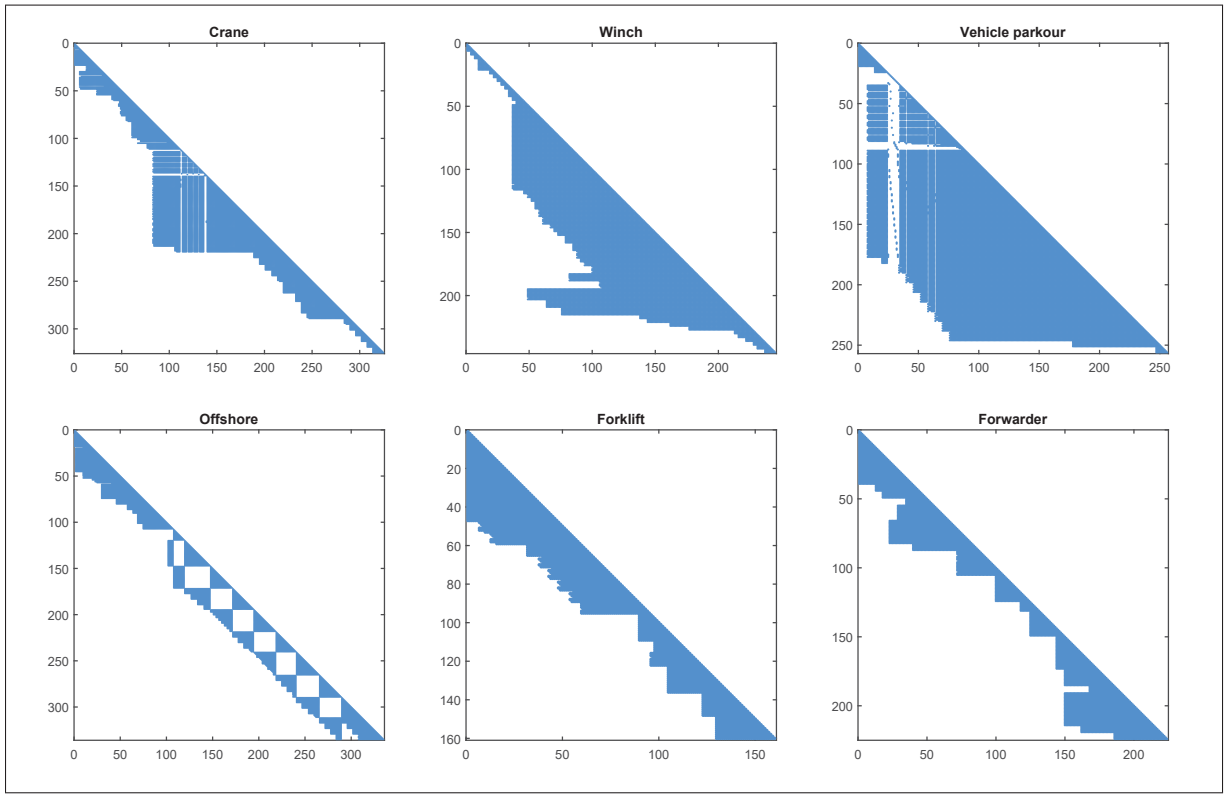

Figure 2.5 Fill-in pattern of the system matrices

2.2.3 Skyline coding

We further improve the efficiency of the block pivoting algorithm exploiting matrix sparsity using a *skyline* coding of the factorization. As a pre-process, constraint variables are reordered

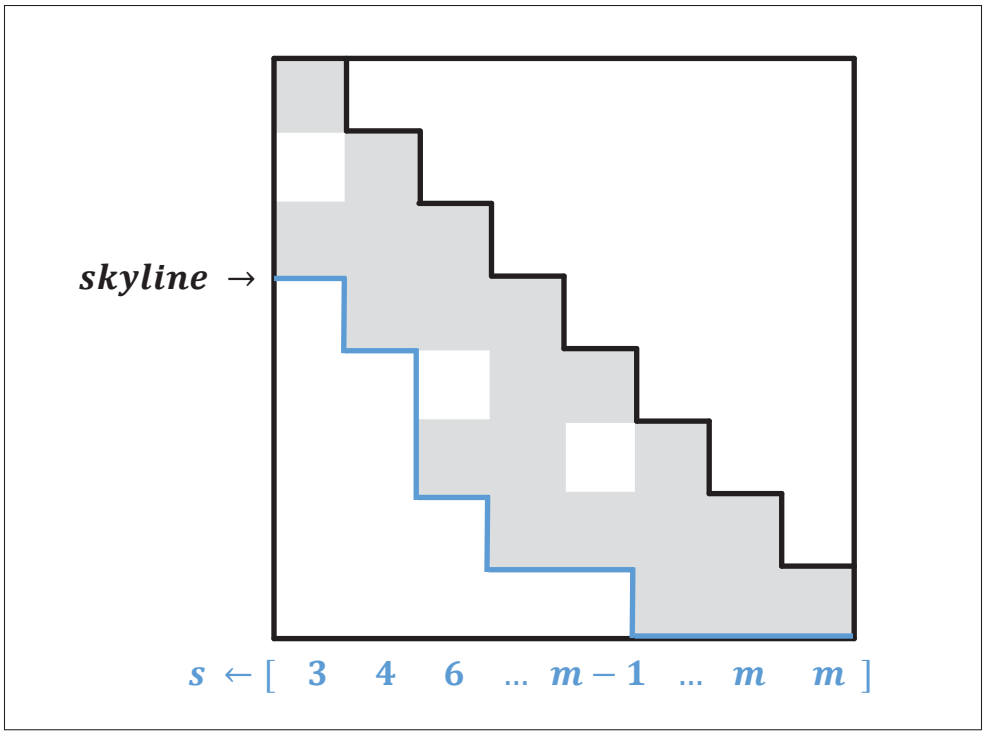

Figure 2.6 Visualization of the skyline data structure

according to a traversal of the constraint graph using the *reverse Cuthill-McKee (RCM)* algorithm (Cuthill & McKee, 1969). This reduces the envelope of the system matrix and, thus, reduces fill-in for $\mathbf{L}$ giving an overall reduced bandwidth. Figure 2.5 shows reordering constraint variables of the system matrices creates a fill-in pattern with smaller bandwidth. The plots show the system matrices for the largest connected island in each simulation. Furthermore, we note that since we only remove, but never add rows and columns, the bandwidth of $\mathbf{A}_{\mathbb{FF}}$ never increases relative to $\mathbf{A}$. The bandwidth of the matrix may therefore be efficiently encoded using a skyline data structure $\mathbf{s}$, which stores the bandwidth at each column of $\mathbf{L}$ relative to the diagonal. In other words, each $\mathbf{s}_i$ stores the row index of the last non-zero entry of column $\mathbf{L}_i$ (see Figure 2.6).

We use the skyline to limit the amount of work done in the DownDate routine, since it helps to avoid unnecessary updates on zero sub-blocks of the Cholesky factorization. We also use the

Algorithm 2.4 Direct solver using an $\mathbf{LL}^\intercal$ Cholesky decomposition

Direct Cholesky solver

Input: A valid decomposition $\mathbf{L}$
The impulse vector of the free variable $\mathbf{b}_\mathbb{F}$
The free index set $\mathbb{F}$
Output: The updated impulse vector of the free set $\boldsymbol{\lambda}_\mathbb{F}$

1 **procedure** `CholSolve` ($\mathbf{L}, \mathbf{b}_\mathbb{F}, \mathbb{F}$)
2 $\quad \boldsymbol{\lambda}_\mathbb{F} \leftarrow \mathbf{b}_\mathbb{F}$
3 $\quad$ **for each** $i \in \mathbb{F}$ **do** ▷ forward substitution $\mathbf{L_y} = \mathbf{b}_\mathbb{F}$
4 $\quad\quad \boldsymbol{\lambda}_i \leftarrow \frac{\boldsymbol{\lambda}_i}{\mathbf{L}_{i,i}}$
5 $\quad\quad \mathbb{F}^\star \leftarrow \{x \in \mathbb{F} \mid x > i\}$ ▷ set of indices in $\mathbb{F}$ following i
6 $\quad\quad$ **while** $j \leftarrow \min(\mathbb{F}^\star)$ **do** ▷ smallest index in $\mathbb{F}^\star$
7 $\quad\quad\quad \boldsymbol{\lambda}_j \leftarrow \boldsymbol{\lambda}_j - \mathbf{L}_{j,i}\boldsymbol{\lambda}_i$
8 $\quad\quad\quad \mathbb{F}^\star \leftarrow \mathbb{F}^\star \setminus \{j\}$ ▷ exits on $\mathbb{F}^\star = \varnothing$
9 $\quad\quad$ **end – while**
10 $\quad$ **end – for**
11 $\quad$ **for each** $i \in \mathbb{F}$ **do** ▷ backward substitution $\mathbf{L}^\intercal \boldsymbol{\lambda}_\mathbb{F} = \mathbf{y}$
12 $\quad\quad \mathbb{F}^\star \leftarrow \{x \in \mathbb{F} \mid x > i\}$ ▷ set of indices in $\mathbb{F}$ following i
13 $\quad\quad$ **while** $j \leftarrow \min(\mathbb{F}^\star)$ **do** ▷ smallest index in $\mathbb{F}^\star$
14 $\quad\quad\quad \boldsymbol{\lambda}_i \leftarrow \boldsymbol{\lambda}_i - \mathbf{L}_{j,i}\boldsymbol{\lambda}_j$
15 $\quad\quad\quad \mathbb{F}^\star \leftarrow \mathbb{F}^\star \setminus \{j\}$ ▷ exits on $\mathbb{F}^\star = \varnothing$
16 $\quad\quad$ **end – while**
17 $\quad\quad \boldsymbol{\lambda}_i \leftarrow \frac{\boldsymbol{\lambda}_i}{\mathbf{L}_{i,i}}$
18 $\quad$ **end – for**
19 $\quad$ **return** $\boldsymbol{\lambda}_\mathbb{F}$
20 **end – procedure**

skyline to reduce the amount of work during the forward/backward substitution of the Cholesky solve (see CholSolve in Algorithm 2.4).

2.3 Results

Our implementation uses the Vortex physics engine (CM Labs Simulations, 2018) for collision detection, computing constraint information, and the dynamics integration. The solver is implemented in C++ using double precision and the Eigen linear algebra library (Eigen, 2018)

Table 2.1 Simulation parameters used in our experiments

Friction coefficient	$\mu = 1.0$
Gravity [m/s^2]	$g = 9.81$
Constraint compliance [N/m]	10^{-6} to 10^{-10}
Step size [s]	$h = 1/60$
Simulation duration [s]	$t = 20$
Max solver iterations	$k = 35$

for matrix multiplications and storage. We choose a column-wise dense matrix storage format to have full control over the matrix traversal and skyline coding in our modified BPP algorithm. The memory for the $m \times m$ matrix is allocated, but for the skyline version of our downdating algorithm we only iterate over elements within the envelope. We initialize all index sets as free when starting all versions of the BPP algorithm.

2.3.1 Examples

Figure 2.2 shows the six examples we use to evaluate our modified BPP algorithm. We summarize the simulation and modeling parameters for these examples in Table 2.1. Additional information for each selected example, such as the maxima for mass ratio, number of constraints, and condition number for each example are presented in the Table 2.2. Note that unconnected systems are split into multiple independent partitions, or "island", and an MLCP is formulated and solved for each such island. The supplementary video[1] also shows a side-by-side comparison of our modified BPP algorithm versus simulating with the full Cholesky decomposition. The results are qualitatively identical. Below, we highlight characteristics of each example.

2.3.1.1 Forklift

A 3,000 kg warehouse forklift picks up a 100 kg crate, accelerates then makes a 360 degree turn around a traffic cone while frictional contact ensures that the crate remains stable on the fork.

[1] https://youtu.be/JSstcHdz3FU

The wheels start skidding during the turn which leads to a increase of variables in the tight set to up to 5 % of all variables.

2.3.1.2 Mobile crane

A 30,000 kg mobile crane hoists a 2,000 kg concrete pipe segment with a 70 kg steel cable attached to a 70 kg hook. Contacts are created when the hook touches the pipe, which causes the solve time to momentarily spike.

2.3.1.3 Vehicle parkour

A simulation involving a 15,000 kg light armored vehicle (LAV) traversing various obstacles (stairs, small rectangular bumps, hills). The power train of the vehicle is modeled using multiple unilateral constraints connecting the vehicle shafts, wheels, differentials, and engine, which creates a highly coupled mechanical system (see sparsity pattern in Figure 2.5).

2.3.1.4 Offshore

A 50,000 kg offshore knuckle boom crane is rigidly attached to a 5,500,000 kg ship hoisting a 15,000 kg subsea module with lightweight steel cable. Each cable segment has mass < 1 kg. Frictional sliding (tight) occurs between the ship hull and the subsea module, as well as cable bodies.

2.3.1.5 Forwarder

A forwarder with a 16-link 1,900 kg gripper arm is used to grasp a 400 kg wooden log and lift it in the air by raising its boom. This scenario is challenging since a stable grasp without sliding is required, and a non-negligible friction impulse is acting between log and claw due to arm rotation. The hydraulics, hydraulics cables, and the vehicle are solved independently from the main mechanical components of the gripper arm.

Table 2.2 Mass ratio, constraint count, and condition numbers for our examples

Scenario	Mass ratio	Number of constraints	Condition number
Crane	6,300:1	430	10^9
Winch	400:1	370	10^8
Vehicle parkour	20,000:1	260	10^{19}
Offshore	4,400,000:1	350	10^9
Forklift	2,700:1	300	10^8
Forwarder	6,500:1	250	10^{11}

2.3.1.6 Winch

A chain, consisting of 50×1 kg links modeled using capsules and connected by spherical joints, is attached to a 400 kg winch and wound at a constant angular velocity. As the chain is dragged along the ground, many of the frictional constraints transition to a sliding mode (tight).

2.3.2 Performance comparison

Table 2.3 shows the average solve time per frame for the original BPP algorithm performing a full Cholesky factorization of $\mathbf{A}_{\mathbb{FF}}$, alongside our modified BPP algorithm performing low-rank downdates to $\mathbf{L}$ with and without taking into account the skyline. Table 2.4 shows numerical error introduced by downdating $\mathbf{L}$ versus computing $\mathbf{L}_{\mathbb{FF}}$. The two rows show the error when downdating is performed without (first row) and with (second row) skyline information.

Figure 2.7 shows the CPU time required to solve Equation 1.20 at each time step for the examples used in our experiments. From left to right, top row: mobile crane and winch with chain. Middle row: vehicle parkour and offshore crane. Bottom row: forklift and logging forwarder. All simulations were performed on an Intel Core i7-5820K (3.3 GHz) with 16 GB of RAM. Our modified BPP algorithm with low-rank downdates and skyline gives the best performance for all test cases, followed by the algorithm using low-rank downdates without skyline information (except for the winch simulation). For simplicity, we do not parallelize the computations of

Table 2.3 Average dynamics solve time and speed-up for the test scenarios

Scenario	**Full Cholesky**	**Downdating**	**Downdating Skyline**
Crane	12.4 ms	4.7 ms (2.7 ×)	3.4 ms (3.6 ×)
Winch	2.0 ms	2.4 ms (0.9 ×)	1.6 ms (1.3 ×)
Vehicle parkour	3.0 ms	2.2 ms (1.4 ×)	2.1 ms (1.4 ×)
Offshore	6.7 ms	4.0 ms (1.7 ×)	2.9 ms (2.3 ×)
Forklift	1.2 ms	1.0 ms (1.3 ×)	0.8 ms (1.5 ×)
Forwarder	3.1 ms	2.5 ms (1.2 ×)	2.0 ms (1.6 ×)

Table 2.4 Numerical error introduced by downdating

Scenario	**Min**	**Max**	**Mean**	**Median**
Crane	6.74e-12	3.00e-10	4.77e-11	3.80e-11
	6.76e-12	2.50e-10	4.84e-11	3.88e-11
Winch	0.00e+00	7.29e-11	6.60e-13	1.31e-13
	0.00e+00	7.34e-11	6.07e-13	1.28e-13
Vehicle parkour	3.17e-13	1.81e-09	4.93e-12	7.18e-13
	2.71e-13	7.99e-10	1.85e-12	7.36e-13
Offshore	0.00e+00	2.00e-08	1.50e-09	6.55e-11
	0.00e+00	1.00e-08	1.30e-09	5.55e-11
Forklift	0.00e+00	5.74e-11	1.50e-12	1.46e-12
	0.00e+00	1.17e-11	1.35e-12	1.36e-12
Forwarder	1.11e-10	7.00e-08	4.04e-09	3.22e-09
	1.11e-10	7.00e-08	4.04e-09	3.22e-09

multiple unconnected islands, i.e. the presented solve times are the sum of the dynamics solve times of all islands. Note that Table 2.3 and Table 2.4 exclude frames for which there are no tight variables, i.e. no pivoting, since the BPP algorithms using full Cholesky or downdating are equivalent in this case.

The most significant speed-up is experienced for the mobile crane, in particular when the crane hook is in contact with the pipe in the beginning of the simulation (0-5 s). The offshore simulation, which resembles the mobile crane example, also achieves a significant performance improvement. In practice, one would only perform low-rank downdates if the number of tight

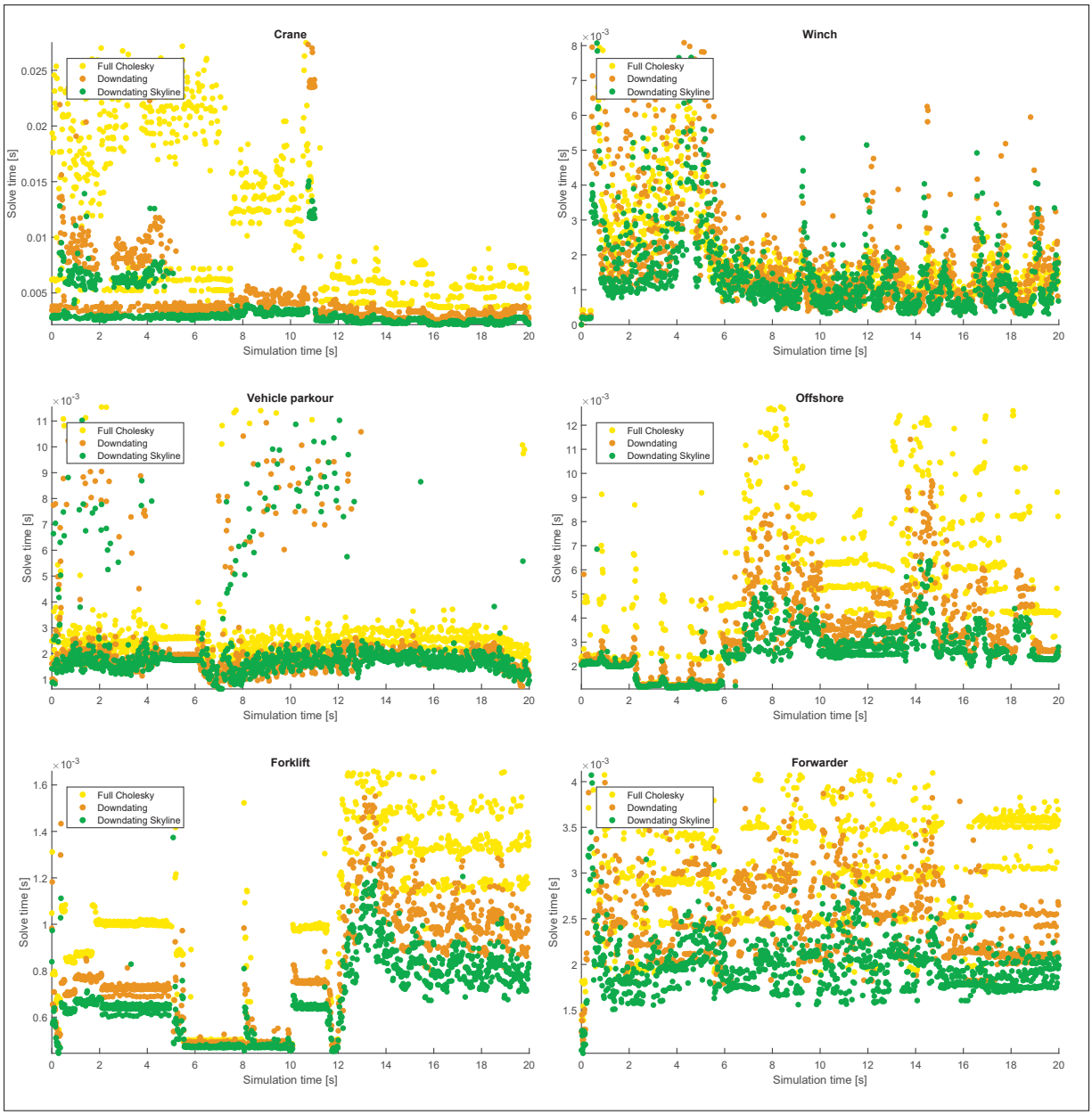

Figure 2.7 Time to solve the constrained dynamics equations for our examples

variables is sufficiently small in order to guarantee to be always at least as fast as the original algorithm applying the full Cholesky factorization. In our tests, we obtain a slow-down for the winch example since we always use the downdating strategy.

Algorithm 2.5 Downdating an $\mathbf{LL}^\intercal$ factorization with skyline

Downdating procedure with skyline

Input: A valid decomposition $\mathbf{L}$
The index of the variable whose contribution is to be removed i
The matrix skyline vector $\mathbf{s}$
Output: The updated $\mathbf{L}$ decomposition

1 **procedure** `Downdate` ($\mathbf{L}, i, \mathbf{s}$)
2 $\quad \beta \leftarrow 1$
3 $\quad \boldsymbol{\sigma} \leftarrow \mathbf{L}_{i+1\ldots m,i}$
4 $\quad$ **for** $j \leftarrow 1 \ldots (m-i)$ **do**
5 $\quad\quad k \leftarrow i + j$
6 $\quad\quad x \leftarrow \boldsymbol{\sigma}_j$
7 $\quad\quad y \leftarrow \mathbf{L}_{k,k}$
8 $\quad\quad z \leftarrow y^2 + \frac{x^2}{\beta}$
9 $\quad\quad \gamma \leftarrow \beta\, y^2 + x^2$
10 $\quad\quad \mathbf{L}_{k,k} \leftarrow \sqrt{z}$
11 $\quad\quad \beta \leftarrow \beta + \frac{x^2}{y^2}$
12 $\quad\quad$ **if** $(\mathbf{s}_k - k) > 0$ **then** ▷ skyline check
13 $\quad\quad\quad j_1 \leftarrow j + 1,\ j_2 \leftarrow j + (\mathbf{s}_k - k)$
14 $\quad\quad\quad k_1 \leftarrow k + 1,\ k_2 \leftarrow k + (\mathbf{s}_k - k)$
15 $\quad\quad\quad \boldsymbol{\sigma}_{j_1\ldots j_2} \leftarrow \boldsymbol{\sigma}_{j_1\ldots j_2} - \frac{x}{y}\mathbf{L}_{k_1\ldots k_2,k}$
16 $\quad\quad\quad \mathbf{L}_{k_1\ldots k_2,k} \leftarrow \mathbf{L}_{k_1\ldots k_2,k} + \frac{\sqrt{z}\,x}{\gamma}\boldsymbol{\sigma}_{j_1\ldots j_2}$
17 $\quad\quad$ **end**
18 $\quad$ **end**
19 $\quad$ **return** $\mathbf{L}$
20 **end – procedure**

Algorithm 2.6 Modified $\mathbf{LL}^\intercal$ Cholesky decomposition with skyline

Direct Cholesky solver with skyline

Input: A valid decomposition $\mathbf{L}$
The right hand side vector of the free variable $\mathbf{b}_\mathbb{F}$
The free index set $\mathbb{F}$
The matrix skyline vector $\mathbf{s}$
Output: The updated impulse vector of the free set $\boldsymbol{\lambda}_\mathbb{F}$

```
1  procedure CholSolve(L, b_F, F)
2      λ_F ← b_F
3      for each i ∈ F do                          ▷ forward substitution L_y = b_F
4          λ_i ← λ_i / L_{i,i}
5          F* ← {x ∈ F | x > i ∧ x < s_i}          ▷ skyline check
6          while j ← min(F*) do                   ▷ smallest index in F*
7              λ_j ← λ_j − L_{j,i} λ_i
8              F* ← F* \ { j }                    ▷ exits on F* = ∅
9          end – while
10     end – for
11     for each i ∈ F do                          ▷ backward substitution L^T λ_F = y
12         F* ← {x ∈ F | x > i ∧ x < s_i}          ▷ skyline check
13         while j ← min(F*) do                   ▷ smallest index in F*
14             λ_i ← λ_i − L_{j,i} λ_j
15             F* ← F* \ { j }                    ▷ exits on F* = ∅
16         end – while
17         λ_i ← λ_i / L_{i,i}
18     end – for
19     return λ_F
20 end – procedure
```

CHAPTER 3

FACTORIZATION CACHING

The results from the downdating method presented in Chapter 2 demonstrate that the approach is viable for complex problems where only a small percentage of variables are pivoted at each frame. We also observed that from one pivoting step to the next (line 16, Alg. 2.2), often only a few variables will pivot. Yet our algorithm computes a downdated factorization starting from the original factorization $\mathbf{L}_0$. This means we recompute the same intermediate factorizations several times each frame. Therefore, we considered if it was possible to accelerate the computation by caching the downdated decomposition $\mathbf{L}$ from a pivot to the next. In other words, for a tight set $\mathbb{T}$, would saving the intermediate state of $\mathbf{L}$ after downdating each variable reduce the amount of computation and thus further speed up the BPP algorithm? In this chapter, we provide details on a caching algorithm and results of experiments obtained using the test scenarios.

There are two new operations required to realize caching for the downdating algorithm. First, we need to determine which Cholesky factorization in the cache most closely matches the current index set. Second, we need to save intermediary factorizations after every downdate operation. The pseudo-code of the modified version of `SolveLCP` procedure is presented in Algorithm 3.1. Essentially, when the tight set changes, the cache is searched for an intermediary factorization that will minimize the amount of computation required to obtain the factorization corresponding to the new index set. The steps to find the right factorization are encapsulated in the method `FindClosestEntry` (see Algorithm 3.2) whose role is further described in Section 3.2.

3.1 Caching data structure

We identify each factorization by the set of tight variables $\mathbb{T}$. The cache is a map of key-value pairs $\{\mathbb{K}, \mathbf{L}\}$ where the keys are unique index sets corresponding to a signature of tight variables, and the value is a downdated factorization that has already been encountered during a previous pivoting step. Because the original decomposition, $\mathbf{L}_0$, includes all variables, we start by initializing the cache with the tuple $\{\varnothing, \mathbf{L}_0\}$. Each time $\mathbf{L}$ is downdated, we get a factorized

Algorithm 3.1 Modified pivoting algorithm using caching

Efficient BPP algorithm with caching

Input: A positive definite matrix $\mathbf{A}$, $\mathbf{b}$, $\boldsymbol{\lambda}_{\text{hi}}, \boldsymbol{\lambda}_{\text{lo}}, \mathbb{C}$;
Output: $\mathbf{w}$, $\boldsymbol{\lambda}$;

```
1  procedure SolveLCP(A, b, λ_hi, λ_lo)
2      L ← Cholesky(A)
3      𝕂 ← ∅                                  ▷ original decomposition has no tight variable
4      ℂ ← { {𝕂, L} }                         ▷ save decomposition to the cache; e.i. L_0
5      initialize index sets 𝔽 and 𝕋
6      k ← 0
7      do
8          {𝕂, L} ← FindClosestEntry(ℂ, 𝕋)
9          for each i ∈ 𝕋 \ 𝕂 do              ▷ run through set in ascending order
10             L ← Downdate(i, L)
11             𝕂 ← 𝕂 ∪ { i }
12             ℂ ← ℂ ∪ { {𝕂, L} }
13         end – for
14         ∀ i ∈ 𝕋_lo,  λ_i ← λ_lo,i
15         ∀ i ∈ 𝕋_hi,  λ_i ← λ_hi,i
16         λ_𝔽 ← CholSolve(L, b_𝔽 − A_𝔽𝕋 λ_𝕋, 𝔽)
17         w = Aλ − b
18         update 𝔽, 𝕋 according to Eq. 2.1
19         k = k + 1
20     while (k < max steps) and (index sets 𝔽, 𝕋 changed)
21 end – procedure
```

matrix corresponding to tight variables being pivoted, and at each pivot the newly downdated matrix with tight index key is stored in the cache map.

We demonstrate how the cache data structure is built with an example. At the beginning of the algorithm, the cache is initialized with the factorization $\mathbf{L}_0$ and a key that corresponds to the empty set $\mathbb{K} = \varnothing$, meaning that all variables are free. However, consider that after the first pivoting step, the tight index set becomes $\mathbb{T} = \{1, 2, 5\}$. Our downdating algorithm from Chapter 2 then incrementally downdates the initial factorization. First, the key-value pair $\{\{1\}, \mathbf{L}_{\{1\}}\}$ is added to the cache. Note that we use the subscript to denote variables that

Algorithm 3.2 Procedure used to find the cache entry

```
1  procedure FindClosestEntry(ℂ, 𝕋)
2      𝕂 ← ∅
3      L ← { y | {x, y} ∈ ℂ ∧ x = 𝕂 }                    ▷ i.e. L_0
4      for each i ∈ 𝕋 do                                  ▷ run through set in ascending order
5          𝕂* ← 𝕂 ∪ { i }                                 ▷ incrementally add tight indices
6          if ∃{x, y} | {x, y} ∈ ℂ ∧ x = 𝕂* then
7              𝕂 ← 𝕂*
8              L ← y
9          else
10             return {𝕂, L}
11         end
12     end – for
13     return {𝕂, L}
14 end – procedure
```

have been pivoted in the factorization. When the next variable is pivoted, the key-value pair $\{\{1,2\}, \mathbf{L}_{\{1,2\}}\}$ is added to the cache, and finally $\{\{1,2,5\}, \mathbf{L}_{\{1,2,5\}}\}$ for the last pivot. Now the cache contains four entries– one for the initial factorization and three for each of the intermediate pivots that are computed by incremental downdating.

3.2 Cached efficient BPP

Rather than downdating $\mathbf{L}$ from $\mathbf{L}_0$, we compute it by starting with the factorization sharing the greatest amount of consecutive tight indices. In order to find the closest entry in the cache to the current tight index set $\mathbb{T}$, we build the key incrementally. We start with a key representing no tight indices, $\mathbb{K} = \emptyset$, which corresponds to the original decomposition L_0. Then, we create a tentative key $\mathbb{K}^\star$ by adding the first tight index in $\mathbb{T}$. If that key exists in the cache, we update the key $\mathbb{K} \leftarrow \mathbb{K}^\star$ and keep track of the corresponding factorization. We then add the next index and repeat this process until either $\mathbb{K}^\star$ is not found in $\mathbf{C}$, or until all the indices in $\mathbb{T}$ have been inserted in $\mathbb{K}$. In the case of the latter, this means that the exact index set already exists in the cache, and no downdating is required. The pseudo-code to find the cache entry is presented

in the method FindClosestEntry of Algorithm 3.2. Once the closest cache entry has been determined, the method returns the closest factorization and its corresponding tight index key.

The cached algorithm then proceeds to downdate the factorization as presented in Algorithm 3.1, using the factorization returned by FindClosestEntry and starting from the first index in $\mathbb{T}$ that isn't in $\mathbb{K}$. As explained in Chapter 2, the downdating operation consists of incrementally modifying the factorization one variable at a time. Once a variable is effectively removed and the matrix is downdated, we save the result in the cache.

Expanding on the example in Section 3.1, if the tight index set now becomes $\mathbb{T} = \{1, 2, 6, 7\}$, we would compute the downdated factorization $\mathbf{L}_{\{1,2,6\}}$ by starting with $\mathbf{L}_{\{1,2\}}$ and pivoting the variable at index 6, then adding the key-value pair $\{\{1, 2, 6\}, \mathbf{L}_{\{1,2,6\}}\}$ to the cache, and repeating the process until all indices in $\mathbb{T}$ have been considered.

Recall that indices are sorted by ascending order, and while it is true that we can permute the index of the constraints in $\mathbf{A}$ the sequence of variables used for downdating must be processed in order because of the nature of a left-looking Cholesky decomposition. In essence, each value in a column i is computed from the values of the previous $i - 1$ columns. This means the order of the indices are extremely important. Furthermore, we can only start from a key $\mathbb{K} \subset \mathbb{T}$ since pivoting a column i modifies entries of the proceeding the $n - i + 1$ columns. For instance, if the tight set becomes $\mathbb{T} = \{2, 6, 7\}$ then there would be no way to use the entries computed for the sequence $\{1, 2, 6, 7\}$ since the first variable index is different.

3.3 Experimentation

We compared the performance of the cached efficient BPP algorithm to the version presented in Chapter 2 using selected frames from the test scenarios. For this comparison, skyline optimization was not used. Speed up due to the caching algorithm as a function of time for our examples is presented in Figure 3.1. We notice that the cached algorithm is several times slower. Note that gaps in the plots of the Stack of boxes example correspond to time steps where no variables are tight (i.e. free falling bodies). Furthermore, the Forklift test used only 22 frames of

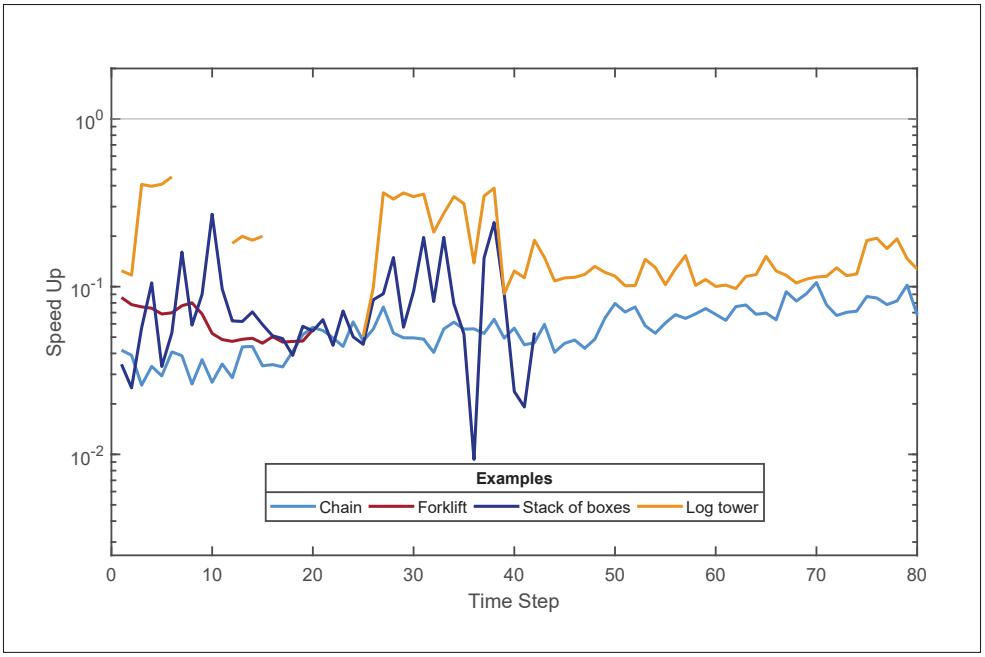

Figure 3.1 Caching algorithm performance

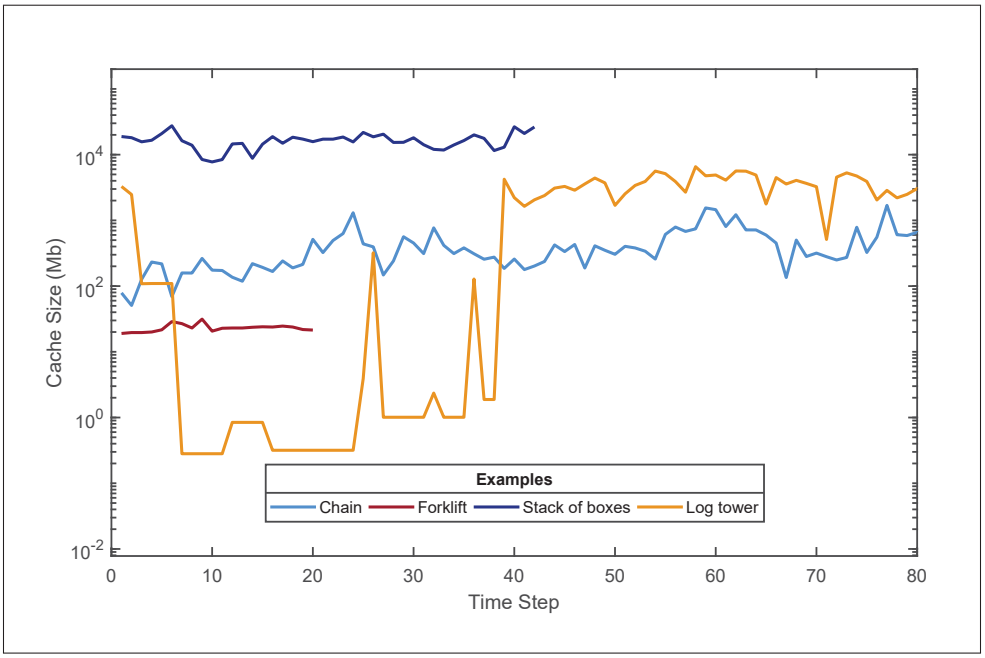

Figure 3.2 Caching algorithm memory usage

simulation, and the Log tower example was cut short because the application ran out of memory. The size of the cache's data structure in RAM as a function of time is shown in Figure 3.2. Again, a limited amount of data was available for the log tower and forklift examples, but we can see the amount of memory needed can vary wildly from time step to time step. From Table 3.1, observe that even seemingly simple examples, such as the Stack of boxes, generate many more contacts and constraints than the Forklift. Simulations involving unstructured piles of bodies therefore require a significantly greater number of cache entries.

Table 3.1 Caching algorithm statistics

Examples	Mean size of $\mathbf{A}$ (# of variables m)	Average size of $\mathbb{C}$ (# of entries)
Chain	528.9	272.5
Forklift	162.0	115.6
Log tower	829.9	3100.5
Stack of boxes	333.9	1351.9

3.4 Discussion

Although it was anticipated that the the amount of memory needed to run the caching algorithm was going to be very large, the hypothesis was that the memory footprint could be sacrificed for performance. After all, the algorithm is meant to run on dedicated high-end simulation machines. However, Figure 3.1 clearly show that caching $\mathbf{L}$ after every pivot is not a good strategy to boost performance. This is because caching this information has a cost: the matrix $\mathbf{A}$ is typically very large which means we need a large amount of random-access memory (RAM), which implies copying a large number of entries. Furthermore, it means that only a small number of whole matrices can be contained in the CPU cache resulting in diminished performance due to cache thrashing. The speed up potential is therefore hampered by the overhead required to store and copy all the matrices, even when the cache memory is pre-allocated. We expect a modified version of the algorithm using the skyline would partially mitigate this problem if the factorization has a very narrow bandwidth since this would reduce the amount of data copied for each matrix, but not completely eliminate the problem.

Also, we observe that based on the stack of boxes example in Figure 3.1, the size of the cache can vary wildly. While the boxes are in free fall, no variables are tight and thus the cache contains only the original factorization. Every time a box makes contact with the floor or another object, we observe witness spikes in the size of the cache due to the new contacts being introduced and the complexity of the simulation increases. An improved caching algorithm should take these performance "spikes" into consideration.

Finally, observe from Table 3.1 that the number of constraints m in a simulation does not necessarily correlate with the size of the cache. Another interesting feature is to see how large the cache grows for the Log tower and the Stack of boxes examples. For disorganized piles of objects, the amount of redundant constraints is expected to be extremely large, which causes variables related to contact constraints to frequently jump from the free to tight and back several times and further causes large growth of the cache. This is a well known problem for these types of scenarios and makes them challenging for direct solvers.

CONCLUSION

Training simulations developed by CM Labs require highly accurate solutions of complex scenarios at real-time frame rates. Their simulations are characterized by stiff multi-body systems with large mass ratios. Pivoting algorithms using direct methods show great potential for solving this category of systems, but struggle to achieve real-time frame rates for extremely complex scenarios.

In order to accelerate the solver and reduce overhead associated with the computation of dynamics, we render the BPP algorithm more efficient by updating the Cholesky factorization with incremental downdates. We have demonstrated that our approach, which uses a sequence of rank-one downdates, doesn't impact the accuracy of the simulation and results in performance improvements of 1.5-3× speed-up in solving the dynamics. We also identified by an empirical process a threshold where, if the number of variable becomes too large, there is no performance gain using our algorithm and the solver can safely fall back to the original BPP algorithm. We have demonstrated the utility of the modified algorithm with complex examples, and for many simulation frames the examples match the threshold for applying our algorithm.

We also note that the lead matrix's bandwidth reduction resulting from a RCM (Cuthill & McKee, 1969) permutation of variables is transferable to the Cholesky factorization. Therefore, performance is further improved by using a skyline data structure to eliminate unnecessary computations.

Further observing that within a time step only a small number of constraint variables will continue to change after the first few iterations of the algorithm, we inferred that a significant amount of computational effort is spent recomputing factorizations from previous iterations, and we hypothesized that the solver could be further accelerated by caching and reusing intermediate factorizations. Unfortunately, this requires swapping large amounts of data in memory, and the resulting algorithm ended up being much slower.

4.1 Future work

Other work has used elimination trees of sparse matrices to reduce the amount of computation needed to get the factorization (Herholz & Alexa, 2018). We briefly explored the possibility of using such a data structure in the context of a pivoting solver, but after some preliminary analysis realized it would not be compatible with the systems we target. This is because the lead matrix in our target systems are reordered using RCM, which does not necessarily reduce matrix fill-in. The reordering step narrows the lead matrix bandwidth, which results in changes to the structure of the underlying elimination tree. For example, by analyzing one of our target examples, Figure 4.1 demonstrates the effect of applying RCM versus a minimum degree reordering (MDR) algorithm. RCM results in a narrow, degenerate tree with very low branching factor. This would offer no benefit when computing and reusing Cholesky factorizations.

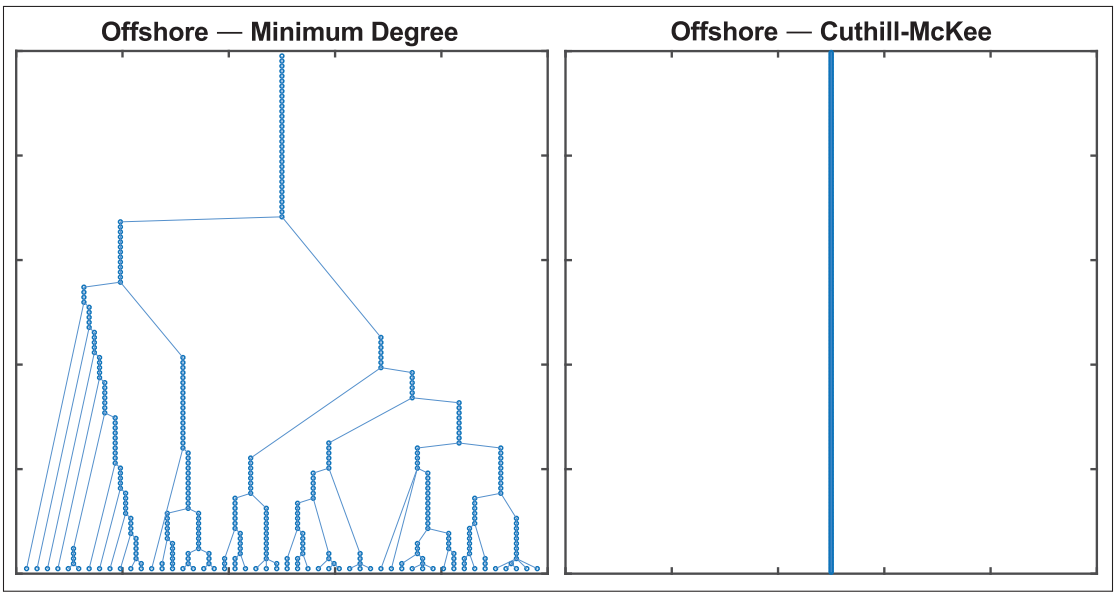

Figure 4.1 Eliminations trees for different permutations of the lead matrix

An avenue of future research would be to explore if elimination trees could offer an acceleration for our target systems with different permutations. However, this could introduce additional overhead due to sparsity analysis and the creation of the elimination tree. Also, due to the size

of our system (100s to 1000s of variables), sparse data structures may offer no benefit, and such a strategy may only benefit very large scale simulations. A comparison of matrix reordering methods for multibody simulation by Torres-Moreno, Blanco, López-Martínez & Giménez-Fernández (2013) found that simple reordering strategies, specifically the column approximate minimum degree method (COLAMD), outperformed graph partitioning algorithms for medium sized systems (i.e., 50 to 2000 variables). This indicates that more sophisticated fill-in reducing algorithms may hinder, rather than improve, performance.

Finally, we hypothesize that an efficient implicit integration technique could be realized by using our downdating approach, where changes to the lead matrix $\mathbf{J}\mathbf{M}^{-1}\mathbf{J}^{\mathsf{T}}$ due to perturbations of the constraint Jacobian $\mathbf{J}$ could be computed efficiently by reusing a Cholesky factorization at the beginning of the time step or even the previous time step.

REFERENCES

Andrews, S., Teichmann, M. & Kry, P. G. (2017). Geometric Stiffness for Real-Time Constrained Multibody Dynamics. *Computer Graphics Forum*, 36(2), 235–246. doi: 10.1111/cgf.13122.

Baraff, D. (1994). Fast Contact Force Computation for Nonpenetrating Rigid Bodies. *Computer Graphics Proceedings, Annual Conference Series*, 23-24, 23–34. doi: 10.1145/192161.192168.

Bender, J., Erleben, K. & Trinkle, J. (2014). Interactive Simulation of Rigid Body Dynamics in Computer Graphics. *Computer Graphics Forum*, 33(1), 246–270. doi: 10.1111/cgf.12272.

Catto, E. (2018). Box2D Physics Engine. [Online] https://box2d.org/.

Cline, M. B. & Pai, D. K. (2003). Post-Stabilization for Rigid Body Simulation with Contact and Constraints. *2003 IEEE International Conference on Robotics and Automation*, 3, 3744-3751 vol.3. doi: 10.1109/ROBOT.2003.1242171.

CM Labs Simulations. (2018). Vortex Studio. [Online] http://www.cm-labs.com/.

Cottle, R. W. & Dantzig, G. B. (1968). Complementary Pivot Theory of Mathematical Programming. *Linear Algebra and its Applications*, 1(1), 103–125. doi: 10.1016/0024-3795(68)90052-9.

Cottle, R., Pang, J. & Stone, R. (1993). *The Linear Complementarity Problem*. SIAM. doi: 10.1137/1.9780898719000.

Coumans, E. (2019). Bullet Physics Library. [Online] https://pybullet.org/.

Cuthill, E. & McKee, J. (1969). Reducing the Bandwidth of Sparse Symmetric Matrices. *Proceedings of the 1969 24th National Conference*, pp. 157–172. doi: 10.1145/800195.805928.

Davis, T. A. & Hager, W. W. (2005). Row Modifications of a sparse Cholesky Factorization. *SIAM Journal on Matrix Analysis and Applications*, 26(3), 621–639. doi: 10.1137/S089547980343641X.

Eigen. (2018). Eigen C++ Library for Linear Algebra, Vector and Matrix Multiplications. [Online] http://eigen.tuxfamily.org/.

Enzenhoefer, A., Andrews, S., Teichmann, M. & Kövecses, J. (2018). Comparison of Mixed Linear Complementarity Problem Solvers for Multibody Simulations with Contact. *Workshop on Virtual Reality Interaction and Physical Simulation*. doi: 10.2312/vriphys.20181063.

Enzenhoefer, A., Lefebvre, N. & Andrews, S. (2019). Efficient Block Pivoting for Multibody Simulations with Contact. *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pp. 2:1–2:9. doi: 10.1145/3306131.3317019.

Erleben, K. (2004). *Stable, Robust, and Versatile Multibody Dynamics Animation*. (Ph.D. thesis, University of Copenhagen, Copenhagen).

Erleben, K. (2007). Velocity-Based Shock Propagation for Multibody Dynamics Animation. *ACM Transactions on Graphics*, 26(2), 1-20. doi: 10.1145/1243980.1243986.

Fratarcangeli, M., Tibaldo, V. & Pellacini, F. (2016). Vivace: A Practical Gauss-seidel Method for Stable Soft Body Dynamics. *ACM Transactions on Graphics*, 35(6), 214:1–214:9. doi: 10.1145/2980179.2982437.

Guendelman, E., Bridson, R. & Fedkiw, R. (2003). Nonconvex Rigid Bodies with Stacking. *ACM Transactions on Graphics*, 22(3), 871–878. doi: 10.1145/882262.882358.

Havok. (2019). Havok Physics. [Online] https://www.havok.com/.

Herholz, P. & Alexa, M. (2018). Factor Once: Reusing Cholesky Factorizations on Sub-meshes. *SIGGRAPH Asia 2018 Technical Papers*, pp. 230:1–230:9. doi: 10.1145/3272127.3275107.

Júdice, J. J. (1994). Algorithms for Linear Complementarity Problems. In Spedicato, E. G. (Ed.), *Algorithms for Continuous Optimization* (pp. 435–474). Springer. doi: 10.1007/978-94-009-0369-2_15.

Júdice, J. J. & Pires, F. M. (1994). A Block Principal Pivoting Algorithm for Large-Scale Strictly Monotone Linear Complementarity Problems. *Computers & operations research*, 21(5), 587–596. doi: 10.1016/0305-0548(94)90106-6.

Keller, E. L. (1973). The General Quadratic Optimization Problem. *Mathematical Programming*, 5(1), 311–337. doi: 10.1007/BF01580136.

Lacoursière, C. (2007). A Parallel Block Iterative Method for Interactive Contacting Rigid Multibody Simulations on Multicore PCs. *Applied Parallel Computing. State of the Art in Scientific Computing*, pp. 956–965. doi: 10.1007/978-3-540-75755-9_113.

Lanczos, C. (2012). *The Variational Principles of Mechanics*. Courier Corporation.

Lloyd, J. E. (2005). Fast Implementation of Lemke's Algorithm for Rigid Body Contact Simulation. *IEEE International Conference on Robotics and Automation*, pp. 4538-4543. doi: 10.1109/ROBOT.2005.1570819.

Mazhar, H., Heyn, T., Negrut, D. & Tasora, A. (2015). Using Nesterov's Method to Accelerate Multibody Dynamics with Friction and Contact. *ACM Transactions on Graphics*, 34(3), 32:1–32:14. doi: 10.1145/2735627.

Murty, K. G. & Yu, F.-T. (1988). *Linear Complementarity, Linear and Nonlinear Programming*. Heldermann.

Seeger, M. (2004). *Low Rank Updates for the Cholesky Decomposition*. University of California at Berkeley, [technical report].

Shoemake, K. (1985). Animating Rotation with Quaternion Curves. *Computer Graphics*, (SIGGRAPH '85), 245–254. doi: 10.1145/325334.325242.

Silcowitz-Hansen, M., Niebe, S. & Erleben, K. (2010, 05). Projected Gauss-Seidel Subspace Minimization Method for Interactive Rigid Body Dynamics - Improving Animation Quality using a Projected Gauss-Seidel Subspace Minimization Method. *International Conference on Computer Graphics Theory and Applications*. doi: 10.1007/978-3-642-25382-9_15.

Smith, R. et al. (2005). Open Dynamics Engine. [Online] https://www.ode.org/.

Tonge, R., Benevolenski, F. & Voroshilov, A. (2012). Mass Splitting for Jitter-Free Parallel Rigid Body Simulation. *ACM Transactions on Graphics*, 31(4), 1–8. doi: 10.1145/2185520.2185601.

Torres-Moreno, J.-L., Blanco, J.-L., López-Martínez, J. & Giménez-Fernández, A. (2013). A Comparison of Algorithms for Sparse Matrix Factoring and Variable Reordering aimed at Real-Time Multibody Dynamic Simulation. *ECCOMAS Multibody Dynamics, European Community on Computational Methods in Applied Sciences*, 167–168.

Tournier, M., Nesme, M., Gilles, B. & Faure, F. (2015). Stable Constrained Dynamics. *ACM Transactions on Graphics*, 34(4), 132:1–132:10. doi: 10.1145/2766969.

Wang, H. (2015). A Chebyshev Semi-Iterative Approach for Accelerating Projective and Position-Based Dynamics. *ACM Transactions on Graphics*, 34(6), 246. doi: 10.1145/2816795.2818063.