Matrix-Free Delassus Operations: Scalable and Memory-Efficient Algorithms

Ajay Suresha Sathya[1], Louis Montaut[1], Yann de Mont-Marin[1], and Justin Carpentier[1]

***Abstract*—The Delassus matrix, closely related to the operational-space inertia matrix, is a fundamental quantity in robotics with applications in simulation, system identification, and control. Traditional approaches compute and store this matrix explicitly, either in sparse or dense form. In this work, we depart from this convention by treating the Delassus matrix as a matrix-free operator. We derive efficient algorithms with low computational complexity that multiply the Delassus matrix or its damped inverse by an input vector or matrix. Unlike approaches based on explicit matrix construction, our method achieves a linear memory footprint, making it scalable to problems with thousands of constraints and suitable for execution on resource-limited hardware. We implement these matrix-free operations on top of the open-source Pinocchio library and evaluate their performance against state-of-the-art methods that rely on explicit matrix computation. Our benchmarks demonstrate substantial speedups, ranging from 2x to over 400x, in contact-rich scenarios.**

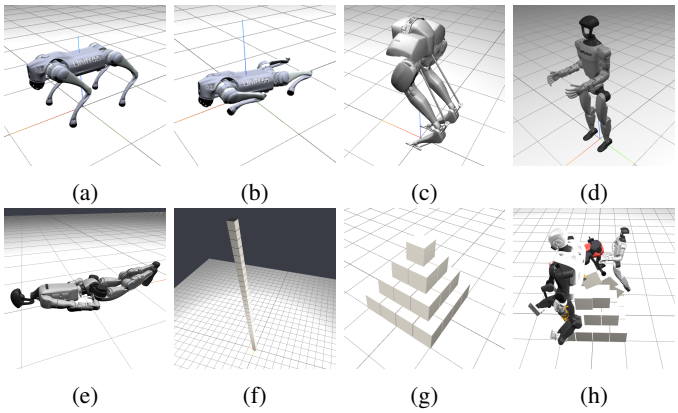


Fig. 1: Illustrating the contact scenarios benchmarked in Table I in Section VI.

I. Introduction

Matrix-free implementations of linear operators are highly beneficial, and often essential, when the explicit construction of a matrix or its inverse is computationally prohibitive or exceeds memory limits. Standard examples include iterative solvers for large sparse linear systems, such as the conjugate gradient method [1]; the L-BFGS method [2] for matrix-free Hessian approximation; automatic differentiation techniques where Jacobian-vector and Hessian-vector products are evaluated without explicitly forming the full matrices [3]; and factor-graph-based algorithms for localization and mapping [4], [5]. In the context of robotics and poly-articulated system simulation, the articulated-body algorithm (ABA) [6]–[8] is a classic example of a matrix-free algorithm. The joint-space inertia matrix (JSIM) is never explicitly formed or inverted using standard linear algebra routines. Instead, the ABA leverages the kinematic tree structure and articulated-body inertias to implicitly compute the inverse JSIM operator.

The Delassus matrix [9]–[11], also known as the inverse operational space inertia matrix, is a fundamental quantity in robot control and simulation. It is a linear operator mapping constraint forces to constraint accelerations (or impulses to velocity changes), defined as $\Lambda^{-1} := JM^{-1}J^T$, where M is the JSIM and J is the constraint Jacobian. Explicitly computing the Delassus matrix can require up to $O(n^3 + m^2n + mn^2)$ operations, where n is the number of degrees of freedom (DOFs) and m is the number of constraints. Typically, the Delassus matrix must also be factorized to solve for constraint forces or impulses, incurring an additional $O(m^3)$ cost. For the restricted case of kinematic trees with unary constraints (i.e., constraints involving a single link without loop-closures), existing efficient recursive algorithms [12], [13] can compute the Delassus matrix and its damped inverse explicitly in $O(n+m^2)$ operations. However, even these optimal complexity algorithms become computationally expensive for high-DOF robots, such as humanoids, in contact-rich scenarios. Furthermore, these recursive algorithms cannot handle the loop-closure constraints that arise in closed-chain mechanisms or during physical interaction with the environment.

Fortunately, many computationally intensive applications, such as robotic simulation or whole-body control, only require matrix-vector products with the Delassus matrix or its damped inverse. Prior work [14], [15] has explored matrix-free Delassus operators, but these often rely on maximal coordinates and do not exploit the inherent articulated structure of robots. Existing matrix-free methods for inverting the Delassus operator [15], [16] are typically iterative, such as those based on the Gauss-Seidel method, and can be slow to converge.

To address this, this letter introduces efficient matrix-free algorithms for evaluating the Delassus operator and its damped inverse. Our approach exploits the articulated structure of robots and is applicable to systems with kinematic loops and arbitrary n-ary constraints. Notably, the damped inverse Delassus operator is computed via a direct method, requiring no iterative solver. The primary contributions of this paper are:

1) **Matrix-free Delassus operator.** We propose a matrix-free algorithm for the Delassus operator with $O(n+m)$ time and memory complexity, leveraging the articulated-body algorithm (ABA).

[1]Inria and Département d'Informatique de l'École Normale Supérieure, PSL Research University in Paris, 75013 Paris, France.
emails: ajay.sathya@inria.fr louis.montaut@inria.fr yann.de-mont-marin@inria.fr justin.carpentier@inria.fr,

2) **Matrix-free damped Delassus inverse operator.** We propose a matrix-free algorithm that exactly computes the damped inverse of the Delassus operator for mechanisms with arbitrary kinematic loops. By utilizing the loop-constrained articulated-body algorithm (LCABA) [17] and the matrix inversion lemma [18], it inherits a worst-case time complexity of $O(n + m^2d + m^3)$ and memory complexity of $O(n+m^2)$, where d is the depth of the mechanism's spanning tree. In practice, it exhibits $O(n + m)$ complexity, as the worst case arises only in unrealistic scenarios where most loops are coupled via shared joints.
3) **Efficient open-source implementation.** We provide an efficient C++ implementation of the proposed algorithms in the widely used open-source PINOCCHIO library [19]. The code will be released upon conclusion of double-blind review.

II. Related work

The Delassus matrix represents the projection of the inverse joint-space inertia matrix onto the constraint space:

$$\Lambda^{-1} = J_c M^{-1} J_c^T, \tag{1}$$

where M is the joint-space inertia matrix (JSIM) and J_c is the constraint Jacobian.

As detailed in [20], the most straightforward method for computing the Delassus matrix involves three steps: i) explicitly forming M and J_c, ii) computing the Cholesky factorization $M = L^T L$, and finally iii) computing $\Lambda^{-1} = \left(J_c L^{-1}\right)\left(J_c L^{-1}\right)^T$. These operations result in a computational complexity of $O(n^3+n^2m+nm^2)$. For kinematic trees with unary constraints (excluding loop closures), Featherstone proposed the LTL-OSIM algorithm [21], which leverages the branching-induced sparsity pattern of M, J_c, and L to compute the Delassus matrix with a reduced complexity of $O(nd^2 + d^2m + dm^2)$, where d is the tree depth. The LTL-OSIM algorithm was extended in [22] to handle kinematic loops while continuing to exploit sparsity in M, J_c, and L. This extension formulates the computation of the Delassus matrix (and its damped inverse) as a sparse Cholesky decomposition of the Karush-Kuhn-Tucker (KKT) system [23] associated with Gauss's principle of least constraint [24]. It achieves a worst-case complexity of $O(nd^2 + n^2m + nm^2)$, achieved for systems with large loops constituting $\propto n$ joints.

In contrast to the joint-space methods above, *recursive* algorithms exploit the articulated structure of robots to achieve lower computational complexities. The first such algorithm, KJR (Kreutz, Jain, and Delgado) [25], reduced the complexity to $O(n + dm^2)$. A faster approach was proposed in [26] by leveraging the extended-force propagator (EFP). The EFP was further utilized in [27] to develop the EFP algorithm (EFPA), achieving a complexity of $O(n + dm + m^2)$. The most recent advancement is the PV-OSIMr algorithm [13], which attains the optimal complexity of $O(n + m^2)$.

The algorithms discussed above, whether joint-space or recursive, also require factorizing the Delassus matrix Λ^{-1} to solve for constraint forces or impulses during simulation, which adds a computational cost of $O(m^3)$. The recently proposed CABA-OSIMr algorithm [12] (restricted to kinematic trees with unary constraints) addresses this by utilizing the matrix inversion lemma [18] and the PV-OSIMr algorithm to directly compute the damped Delassus inverse $\Lambda_\mu = \left(\Lambda^{-1} + \mu I_m\right)^{-1}$ with an optimal $O(n + m^2)$ complexity. Computing the damped inverse is a standard technique for ensuring numerical stability near constraint singularities and is adopted in simulators such as MuJoCo [28], [29], Drake [30], and Simple [31].

While existing methods compute the Delassus matrix or its inverse explicitly, they incur at least $O(n + m^2)$ complexity even in the best-case scenario. However, contact simulation algorithms only require matrix-vector products with these operators, which can be evaluated more efficiently without forming the matrices. Prior research [14], [15] into matrix-free Delassus operators utilized maximal coordinates, which treat all joints as constraints and result in large, sparse systems. These methods generally do not exploit the articulated structure of the robot. Furthermore, existing matrix-free inverse operators [15], [16] often rely on iterative Gauss-Seidel-based methods, which are first-order, inexact, and sensitive to ill-conditioning. In contrast, the matrix-free operators introduced in this letter exploit the robot's articulated structure and avoids including even spanning-tree joint constraints in the Delassus operator. Our damped Delassus inverse operator is a direct, non-iterative, second-order algorithm that provides greater robustness to ill-conditioning.

III. Preliminaries

This section provides the necessary background on the Delassus matrix and relevant rigid-body dynamics algorithms.

A. Constrained dynamics and the Delassus matrix

Let $\mathbf{q} \in \mathcal{Q}$, $\boldsymbol{\nu} \in \mathcal{T}_{\mathbf{q}}\mathcal{Q} \simeq \mathbb{R}^n$ and $\dot{\boldsymbol{\nu}} \in \mathbb{R}^n$ denote generalized coordinates, velocities and accelerations, respectively. When the system is subject to constraints, the resulting accelerations and constraint forces $\boldsymbol{\lambda} \in \mathbb{R}^m$ due to applied torques $\boldsymbol{\tau}$ are obtained by solving the coupled system of linear equations:

$$M(\mathbf{q})\dot{\boldsymbol{\nu}} + \mathbf{c}(\mathbf{q}, \boldsymbol{\nu}) = S^\top \boldsymbol{\tau} + J_c(\mathbf{q})^\top \boldsymbol{\lambda}, \tag{2a}$$

$$J_c(\mathbf{q})\dot{\boldsymbol{\nu}} = \mathbf{a}_{\text{des}}, \tag{2b}$$

where $M \in \mathbb{S}^n_{++}$ is the joint-space inertia matrix (JSIM), $\mathbf{c} \in \mathbb{R}^n$ is the generalized forces due to Coriolis, centrifugal, and gravity effects, $S \in \{0,1\}^{n_a \times n}$ is a selection matrix for actuated joints, $J_c \in \mathbb{R}^{m \times n}$ is the constraint Jacobian, and $\mathbf{a}_{\text{des}} \in \mathbb{R}^m$ is the desired constraint acceleration (typically $-\dot{J}_c\boldsymbol{\nu}$). Since M is positive definite and invertible, solving for $\dot{\boldsymbol{\nu}}$ in (2a) and substituting it into (2b) yields:

$$\Lambda^{-1}(\mathbf{q})\boldsymbol{\lambda} = -\mathbf{g}(\mathbf{q}, \boldsymbol{\nu}, \boldsymbol{\tau}, \mathbf{a}_{\text{des}}), \tag{3}$$

where $\mathbf{g} = \mathbf{a}_{\text{des}} - \dot{J}_c\boldsymbol{\nu} - J_c M^{-1}(S\boldsymbol{\tau} - \mathbf{c})$. For brevity, variable dependencies are omitted hereafter. Λ^{-1} is positive definite if J_c has full row rank.

The Delassus matrix can be singular in the presence of redundant constraints or at kinematic singularities, and is typically ill-conditioned near singularities. A common practical remedy is using the damped inverse of the Delassus matrix, defined as:

$$\Lambda_R = \left(R^{-1} + J_c M^{-1} J_c^\top\right)^{-1}, \tag{4}$$

where $R \in \mathbb{S}^m_{++}$ is a diagonal matrix. Solving for constraint forces using Λ_R provides a regularized solution that is well-defined in singular cases. However, this introduces artificial compliance, each diagonal element of R can be interpreted as a stiffness parameter for the corresponding constraint. The proximal algorithm approach to contact simulation [31] leverages shifted regularization to satisfy constraints exactly (down to numerical tolerance) by iteratively removing this compliance while still benefiting from the numerical stability of the damped inverse.

Remark 1. While the above discussion focuses on traditional acceleration-level constraints, the Delassus matrix and its damped inverse can also be defined in the context of velocity-level constraints, such as those encountered in impulse-based contact dynamics [29], [31].

B. Loop-Constrained Articulated-Body Algorithm (LCABA)

This subsection reviews the Loop-Constrained Articulated-Body Algorithm (LCABA) [17], a low-complexity constrained dynamics algorithm supporting general kinematic loop constraints. This algorithm is adapted in Sec. V to derive the damped Delassus inverse operator.

A kinematic mechanism can be represented by a *connectivity graph*, an undirected graph where nodes and edges denote links and joints, respectively. A spanning tree of this graph is a subgraph containing all nodes and a subset of edges such that the subgraph is connected and acyclic. The edges omitted from the spanning tree correspond to *cut-joints*, which are imposed via loop-closure constraints. The constrained dynamics of the resulting system, comprising the spanning tree and the loop-closure constraints, can be formulated at the link level as a quadratic program (QP) using Gauss's principle of least constraint [17]:

$$\begin{aligned}
\underset{\dot{\boldsymbol{\nu}},\mathbf{a}}{\textbf{minimize}} \quad & \sum_{i\in\mathcal{S}} \left\{ \frac{1}{2}\mathbf{a}_i^\top H_i \mathbf{a}_i - \mathbf{f}_i^\top \mathbf{a}_i - \boldsymbol{\tau}_i^\top \dot{\boldsymbol{\nu}}_i \right\} && \text{(5a)}\\
\textbf{subject to} \quad & \mathbf{a}_i = \mathbf{a}_{\pi(i)} + S_i \dot{\boldsymbol{\nu}}_i + \mathbf{a}_{b,i}, && i \in \mathcal{S}, \ \text{(5b)}\\
& \sum_{i\in\mathcal{L}_e} K_e^i \mathbf{a}_i = \mathbf{k}_e, && e \in \mathcal{E}, \ \text{(5c)}\\
& \mathcal{K}_i \dot{\boldsymbol{\nu}}_i = \mathfrak{k}_i, && i \in \mathcal{J}, \ \text{(5d)}
\end{aligned}$$

where $\mathbf{a}_i \in \mathbb{M}^6$, $H_i \in \mathbb{I}^6 \simeq \mathbb{S}^6_{++}$, and $\mathbf{f}_i \in \mathbb{F}^6$ denote the i^{th} link's spatial acceleration, inertia, and resultant spatial force, respectively (see [20, Chap. 1] for notation details). The terms $\boldsymbol{\tau}_i \in \mathbb{R}^{n_i}$ and $\dot{\boldsymbol{\nu}}_i \in \mathbb{R}^{n_i}$ are the i^{th} joint's actuation torque and joint acceleration, respectively. Eqs. (5b) and (5c) impose constraints from spanning-tree joints and cut-joints, respectively. $\pi(i)$ indexes the parent link of the i^{th} link in the spanning tree, $S_i \in \mathbb{M}^{6\times n_i}$ is the i^{th} joint's motion subspace matrix, and $\mathbf{a}_{b,i} \in \mathbb{M}^6$ is the bias acceleration. $\mathcal{S}$ and $\mathcal{E}$ are ordered sets that index spanning-tree joints and cut-joints. For $i, j \in \mathcal{S}, i < j$ if $i = \pi(j)$. Note that $\mathcal{S} \cap \mathcal{E} = \emptyset$. Given e in $\mathcal{E}$, $\mathcal{L}_e$ is the set indexing links constrained by the e^{th} cut-joint, and for i in $\mathcal{L}_e$, $K_e^i \in \mathbb{R}^{m_e\times 6}$ is its motion constraint matrix on the i^{th} link, and $\mathbf{k}_e \in \mathbb{R}^{m_e}$ is the desired constraint acceleration. Finally, Eq. (5d) imposes additional joint-level constraints $\mathcal{K}_i \in \mathbb{R}^{m_i\times n_i}$ with $\mathfrak{k}_i \in \mathbb{R}^{m_i}$, where $\mathcal{J} \subseteq \mathcal{S}$ and hence $\mathcal{J} \cap \mathcal{E} = \emptyset$.

Remark 2. The joint-level constraints in Eq. (5d) can also be modeled as link-level motion constraints by noting that $\dot{\boldsymbol{\nu}}_i = S_i^+ \left(\mathbf{a}_i - \mathbf{a}_{\pi(i)} - \mathbf{a}_{b,i}\right)$, where S_i^+ is the Moore-Penrose pseudoinverse of S_i. However, treating joint-level constraints separately allows for more efficient computation.

Solving the QP above Eq. (5) is equivalent to solving the constrained dynamics in Eq. (2). Link-level cut-joint constraints in Eq. (5c) are typically converted to generalized coordinates in Eq. (2b) via the substitution $\mathbf{a}_i = J_i \dot{\boldsymbol{\nu}} + \dot{J}_i \boldsymbol{\nu}$, where $J_i \in \mathbb{R}^{6\times n_i}$ is the i^{th} link's kinematic Jacobian matrix. LCABA solves Eq. (5) via non-serial dynamic programming, yielding a recursive algorithm that exploits the mechanism's spanning-tree structure. For kinematic trees or mechanisms with only *external* loops (where every loop includes the ground link), LCABA reduces to the constrained articulated-body algorithm (CABA) [12], which runs in $O(n+m)$ time. For internal loops, the worst-case complexity is $O(n+dm^2+m^3)$, occurring only when loops are significantly coupled. In practice, loops are typically localized, and the algorithm exhibits $O(n+m)$ performance [17].

IV. Matrix-Free Delassus Operator

This section derives an efficient matrix-free algorithm to evaluate the Delassus operator $\Lambda^{-1} = J_c M^{-1} J_c^\top$ in $O(n+m)$ time and $O(n)$ memory, and presents it in an algorithmic form.

The Delassus operator is a linear operator that maps constraint forces $\boldsymbol{\lambda}$ to constraint accelerations $\mathbf{a}_c$,

$$\mathbf{a}_c = \Lambda^{-1}(\boldsymbol{\lambda}) = J_c M^{-1} J_c^\top(\boldsymbol{\lambda}). \tag{6}$$

This mapping can be viewed as the composition of three linear operators: $J_c \circ M^{-1} \circ J_c^\top$. Each constituent operator can be evaluated without explicitly forming its corresponding matrix. Accordingly, the evaluation of the Delassus operator is split into three steps:

1) Resolve the constraint forces to joint-space torques: $\boldsymbol{\tau}_c = J_c^\top(\boldsymbol{\lambda})$.
2) Solve for the resulting joint accelerations: $\dot{\boldsymbol{\nu}} = M^{-1}(\boldsymbol{\tau}_c)$.
3) Compute the accelerations in constraint space: $\mathbf{a}_c = J_c(\dot{\boldsymbol{\nu}})$.

A. Resolving constraint forces in joint-space

The matrix-free linear operator in the first step (Item 1), $J_c^\top(\boldsymbol{\lambda})$, is implemented using the link-level constraint formulation from Eq. (5). The constraint force vector $\boldsymbol{\lambda}$ is a concatenation of individual constraint forces $\boldsymbol{\lambda}^\top = \begin{bmatrix}\boldsymbol{\lambda}_\alpha^\top\end{bmatrix}^T$

for $\alpha \in \mathcal{E} \cup \mathcal{J}$. For a constraint $e \in \mathcal{E}$, the spatial force acting on link $i \in \mathcal{L}_e$ due to the constraint force $\boldsymbol{\lambda}_e$ is $K_e^{i\top}\boldsymbol{\lambda}_e$ [32]. Once these spatial forces are accumulated, the resulting joint torques are computed via a recursive backward sweep through the spanning tree. This procedure is identical to the backward force sweep in the Recursive Newton-Euler Algorithm (RNEA) [33]. The constraint force due to joint-level constraints $\mathcal{K}_i^\top \boldsymbol{\lambda}_i$ for $i \in \mathcal{J}$, is directly added to the corresponding joint torque.

Alg. 1 details the matrix-free implementation. It runs in $O(n+m)$ time and memory: line 4 accumulates spatial forces in $O(m)$ time (assuming a constant upper-bound for number of links per constraint $|\mathcal{L}_e|$), and the backward sweep (lines 8–10) executes in $O(n)$ time and memory.

Algorithm 1 Matrix-free evaluation of $J_c^\top(\boldsymbol{\lambda})$

Require: $\boldsymbol{\lambda}$; $\mathcal{S}$, π, S, $\mathcal{E}$, $\mathcal{L}$, K, $\mathcal{J}$, $\mathcal{K}$
1: Initialize $\mathbf{f}_i \leftarrow \mathbf{0}_6, \boldsymbol{\tau}_{ci} \leftarrow \mathbf{0}_{n_i}$ for $i \in \mathcal{S}$
2: **for** $e \in \mathcal{E}$ **do** ▷ Loop over constraints
3: **for** $i \in \mathcal{L}_e$ **do** ▷ Loop over links in e^{th} constraint
4: $\mathbf{f}_i \leftarrow \mathbf{f}_i + K_e^{i\top}\boldsymbol{\lambda}_e$ ▷ Contribution from e^{th} constraint force
5: **for** i in $\mathcal{J}$ **do** ▷ Add joint-level constraint torques
6: $\boldsymbol{\tau}_{ci} \leftarrow \boldsymbol{\tau}_{ci} + \mathcal{K}_i^\top \boldsymbol{\lambda}_i$
7: **for** $i \in \mathcal{S}_{\text{reversed}}$ **do** ▷ Backward sweep
8: $\boldsymbol{\tau}_{ci} \leftarrow \boldsymbol{\tau}_{ci} + S_i^\top \mathbf{f}_i$ ▷ Joint torque due to constraint forces
9: **if** $\pi(i) \neq 0$ **then**
10: $\mathbf{f}_{\pi(i)} \leftarrow \mathbf{f}_{\pi(i)} + \mathbf{f}_i$ ▷ Propagate wrench to parent
return $\boldsymbol{\tau}_c$

B. Solving for resulting joint-accelerations

The second operator (Item 2), $M^{-1}(\boldsymbol{\tau}_c)$, maps joint torques to joint accelerations. This is implemented in a matrix-free manner by adapting the standard articulated-body algorithm (ABA) [20] to omit the computation of bias accelerations and forces (the $\mathbf{c}$ term in Eq. (2a)). The procedure is detailed in Alg. 2. Each line performs $O(1)$ operations per link, resulting in $O(n)$ time and memory complexity.

Algorithm 2 Matrix-free evaluation of $M^{-1}(\boldsymbol{\tau}_c)$

Require: $\boldsymbol{\tau}_c$; $\mathcal{S}$, π, S, H
1: Initialize $\mathbf{f}_i \leftarrow \mathbf{0}_6, H_i^A = H_i$ for $i \in \mathcal{S}$
2: **for** $i \in \mathcal{S}_{\text{reversed}}$ **do** ▷ **Backward sweep**
3: $\mathbf{u}_i \leftarrow \boldsymbol{\tau}_{ci} - S_i^\top \mathbf{f}_i$ ▷ Net joint torque
4: $U_i = H_i^A S_i$; $D_i = S_i^\top U_i$; Compute D_i^{-1}
5: **if** $\pi(i) \neq 0$ **then**
6: $\mathbf{f}_{\pi(i)} \leftarrow \mathbf{f}_{\pi(i)} + \mathbf{f}_i + U_i D_i^{-1} \mathbf{u}_i$
7: $H_{\pi(i)}^A \leftarrow H_{\pi(i)}^A + H_i^A - U_i D_i^{-1} U_i^\top$
8: **for** $i \in \mathcal{S}$ **do** ▷ **Forward sweep**
9: $\dot{\boldsymbol{\nu}}_i \leftarrow D_i^{-1}\left(\mathbf{u}_i - U_i^\top \mathbf{a}_{\pi(i)}\right)$
10: $\mathbf{a}_i \leftarrow \mathbf{a}_{\pi(i)} + S_i \dot{\boldsymbol{\nu}}_i$
return $\dot{\boldsymbol{\nu}}$

C. Computing constraint accelerations

The final operator (Item 3), $J_c(\dot{\boldsymbol{\nu}})$, maps joint accelerations to constraint accelerations $\mathbf{a}_c$, which concatenates individual constraint accelerations $\mathbf{a}_c^\top = \begin{bmatrix}\mathbf{a}_{c\alpha}^\top\end{bmatrix}$ for $\alpha \in \mathcal{J} \cup \mathcal{E}$. This is implemented by propagating link accelerations via a forward kinematic sweep and transforming them into the constraint space using Eq. (5c). The procedure is detailed in Alg. 3. The forward sweep (lines 2–3) requires $O(n)$ time and memory, while computing constraint accelerations (line 6) requires $O(m)$ time. The total complexity is $O(n+m)$.

Algorithm 3 Matrix-free evaluation of $J_c(\dot{\boldsymbol{\nu}})$

Require: $\dot{\boldsymbol{\nu}}$; $\mathcal{S}$, π, S, $\mathcal{E}$, $\mathcal{L}$, K, $\mathcal{J}$, $\mathcal{K}$
1: Initialize $\mathbf{a}_{c\alpha} \leftarrow \mathbf{0}_{m_\alpha}$ for $\alpha \in \mathcal{E} \cup \mathcal{J}$, $\mathbf{a}_i \leftarrow \mathbf{0}_6$ for $i \in \mathcal{S}$
2: **for** $i \in \mathcal{S}$ **do** ▷ **Forward kinematic sweep**
3: $\mathbf{a}_i \leftarrow \mathbf{a}_{\pi(i)} + S_i \dot{\boldsymbol{\nu}}_i$
4: **for** $e \in \mathcal{E}$ **do** ▷ Loop over constraints
5: **for** $i \in \mathcal{L}_e$ **do** ▷ Loop over links in e^{th} constraint
6: $\mathbf{a}_{ce} \leftarrow \mathbf{a}_{ce} + K_e^i \mathbf{a}_i$
7: **for** $i \in \mathcal{J}$ **do** ▷ Extract joint constraint accelerations
8: $\mathbf{a}_{ci} = \mathcal{K}_i \dot{\boldsymbol{\nu}}_i$
return $\mathbf{a}_c$

D. Implementation Optimizations

A naive sequential execution of the modular operators involves four recursive sweeps: one backward in Alg. 1, two in Alg. 2, and one forward in Alg. 3. The implementation can be optimized to require only two sweeps in total as follows:

Merging backward sweeps. Alg. 1 converts spatial forces into joint torques $\boldsymbol{\tau}_c$, which Alg. 2 then utilizes. These can be merged by using the spatial forces $\mathbf{f}_i$ aggregated from constraints (Alg. 1, lines 2–4) directly as initial values for $\mathbf{f}_i$ in the ABA backward sweep (Alg. 2, line 2), bypassing the explicit computation of $\boldsymbol{\tau}_c$.

Merging forward sweeps. Alg. 3 re-propagates link accelerations $\mathbf{a}_i$ that are already computed during the forward sweep of Alg. 2. This redundancy is eliminated by using the $\mathbf{a}_i$ values from Alg. 2 directly for the constraint acceleration computation (Alg. 3, line 6).

V. Matrix-Free Damped Delassus Inverse Operator

This section presents our matrix-free algorithm for evaluating the damped inverse of the Delassus matrix. We begin by reviewing the matrix inversion lemma and its application to the damped Delassus inverse, followed by the derivation of the matrix-free operator and its algorithmic implementation.

A. Matrix inversion lemma

The matrix inversion lemma (MIL) [18], also known as the Sherman-Morrison-Woodbury formula, states that for invert-

ible matrices A and C:

$$(A+UCV)^{-1} = A^{-1} - A^{-1}U\left(C^{-1}+VA^{-1}U\right)^{-1}VA^{-1}. \tag{7}$$

Applying this lemma to the damped Delassus inverse, $\Lambda_R = \left(R^{-1} + J_c M^{-1} J_c^\top\right)^{-1}$, yields:

$$\Lambda_R = R - RJ_c\left(M + J_c^\top RJ_c\right)^{-1} J_c^\top R. \tag{8}$$

The term $M_R := M + J_c^\top RJ_c$ corresponds to a *modified JSIM*, representing the system inertia augmented by regularization terms from the constraints. This structural insight was previously exploited in the cABA-OSIM algorithm [12] to explicitly compute the damped Delassus inverse for kinematic trees with unary constraints in $O(n+m^2)$ operations.

In many applications, the explicit computation of Λ_R is unnecessary. We instead consider the damped inverse as a linear operator mapping constraint accelerations $\mathbf{a}_c$ to constraint forces $\boldsymbol{\lambda}$. Substituting Eq. (8) into $\boldsymbol{\lambda} = \Lambda_R \mathbf{a}_c$ leads to the operator form:

$$\boldsymbol{\lambda} = \left[R - RJ_c\underbrace{\left(M + J_c^\top RJ_c\right)}_{M_R}^{-1} J_c^\top R\right](\mathbf{a}_c). \tag{9}$$

B. Matrix-free evaluation of the damped Delassus inverse operator

Evaluating Eq. (9) requires solving the linear system M_R. Unlike the standard JSIM M, which can be solved via ABA, M_R includes the constraint regularization term. We therefore adapt the LCABA algorithm [17], which is designed to solve for this modified JSIM in a recursive, matrix-free manner.

The evaluation of the damped Delassus inverse operator is performed in four steps:

1) Compute the "compliant" constraint forces $\boldsymbol{\lambda}_R = R\mathbf{a}_c$ and resolve them into joint-space torques: $\boldsymbol{\tau}_R = J_c^\top(\boldsymbol{\lambda}_R)$.
2) Solve for the modified joint accelerations: $\dot{\boldsymbol{\nu}}_R = M_R^{-1}(\boldsymbol{\tau}_R)$.
3) Compute the resulting constraint-space accelerations: $\mathbf{a}_R = J_c(\dot{\boldsymbol{\nu}}_R)$.
4) Evaluate the final constraint forces: $\boldsymbol{\lambda} = \boldsymbol{\lambda}_R - R\mathbf{a}_R$.

C. Algorithmic derivation of the matrix-free damped Delassus inverse operator

Steps 1, 3, and 4 in the procedure above are straightforward to implement in a matrix-free manner. The operation $\boldsymbol{\lambda}_R = R\,\mathbf{a}_c$ in Step 1 and $\boldsymbol{\lambda} = \boldsymbol{\lambda}_R - R\mathbf{a}_R$ in Step 4 are computed as element-wise scaling since R is a diagonal matrix. The operators $J_c^\top(\boldsymbol{\lambda}_R)$ in Step 1 and $J_c(\dot{\boldsymbol{\nu}}_R)$ in Step 3 can readily re-use the matrix-free algorithms from Section IV (Alg 1 and Alg. 3, respectively).

The only remaining operation is Step 2, which involves solving the modified JSIM $M_R = M + J_c^\top RJ_c$. This is implemented in a matrix-free manner by adapting the LCABA algorithm [17], a generalization of the ABA algorithm handling such modified JSIMs. The aspects of LCABA that address bias forces and accelerations are omitted here, as they are not required. We list the adapted algorithm in Alg. 4 for completeness, where $\mathcal{S}^{\mathcal{E}}$ is the spanning-tree joint elimination ordering chosen by LCABA. The computations corresponding to the ABA algorithm are printed in brown color. For a detailed description of LCABA, we refer readers to [17].

Algorithm 4 Matrix-free evaluation of the expression $M_R^{-1}(\boldsymbol{\tau}_R)$ adapted from LCABA [17]

Require: $\boldsymbol{\tau}_R$; R, $\mathcal{S}$, $\mathcal{S}^{\mathcal{E}}$, π, $\mathcal{E}$, S, $\mathcal{L}$, K, $\mathcal{J}$, $\mathcal{K}$

Initialize $\mathbf{f}_i \leftarrow \mathbf{0}_6, H_{i,i} = H_i, \mathcal{N}_i \leftarrow \{\}$ for $i \in \mathcal{S}$

Process constraints: update inertia and neighbors

for $e \in \mathcal{E}$ **do**
  for $i \in \mathcal{L}_e$ **do** ▷ Loop over links in e^{th} constraint
    $H_{i,i} \leftarrow H_{i,i} + K_e^{i\top} R_e K_e^i$ ▷ Add constraint inertia
    for $j \in \mathcal{L}_e$ if $j \neq i$ **do**
      if $H_{i,j}$ is undefined **then**
        $H_{i,j} \leftarrow 0_{6\times 6}$
        $\mathcal{N}_i \leftarrow \mathcal{N}_i \cup \{j\}$; ▷ Update neighbor sets
      $H_{i,j} \leftarrow H_{i,j} + K_e^{i\top} R_e K_e^j$

for $i \in \mathcal{J}$ **do** ▷ Initialize inertias due to joint constraints
  $D_i \leftarrow \mathcal{K}_i^\top R_i \mathcal{K}_i$

Backward Sweep

for $i \in \mathcal{S}^{\mathcal{E}}$ **do**
  $\mathbf{u}_i \leftarrow \boldsymbol{\tau}_{Ri} - S_i^\top \mathbf{f}_i$ ▷ Net joint torque
  $U_i = H_{i,i}S_i$; $D_i \leftarrow D_i + S_i^\top U_i$; Compute D_i^{-1}
  $P_i = I_6 - U_i D_i^{-1} S_i^\top$ ▷ Projection matrix
  for $j \in \mathcal{N}_i \cup \{\pi(i)\}$ **do** ▷ Update connections
    $\mathcal{N}_j \leftarrow \mathcal{N}_j - \{i\}$ ▷ Remove i^{th} link from neighbors
    for $k \in \mathcal{N}_i \cup \{\pi(i)\}$ if $k \neq j$ and $k \notin \mathcal{N}_j$ **do**
      $H_{k,j} \leftarrow 0_{6\times 6}$
      $\mathcal{N}_j \leftarrow \mathcal{N}_j \cup \{k\}$ ▷ Update neighbor set
  for $j \in \mathcal{N}_i$ **do** ▷ Update coupling inertias and forces
    $H_{\pi(i),j} \leftarrow H_{\pi(i),j} + H_{i,j} P_i^\top$
    $\mathbf{f}_j \leftarrow \mathbf{f}_j + H_{i,j} S_i D_i^{-1} \mathbf{u}_i$
    $H_{j,j} \leftarrow H_{j,j} - H_{i,j} S_i D_i^{-1} S_i^\top H_{i,j}^\top$
    for $k \in \mathcal{N}_i$ if $k > j$ **do**
      $H_{k,j} \leftarrow H_{k,j} - H_{i,k} S_i D_i^{-1} S_i^\top H_{i,j}^\top$
      $H_{j,k} \leftarrow H_{k,j}^\top$ ▷ Re-symmetrize
  if $\pi(i) \in \mathcal{N}_i$ **then** ▷ Re-symmetrize if parent is neighbor
    $H^A_{\pi(i)} \leftarrow H^A_{\pi(i)} + \left(H_{i,\pi(i)} P_i^\top\right)^\top$
  if $\pi(i) \neq 0$ **then**
    $\mathbf{f}_{\pi(i)} \leftarrow \mathbf{f}_{\pi(i)} + \mathbf{f}_i - U_i D_i^{-1} \mathbf{u}_i$
    $H_{\pi(i),\pi(i)} \leftarrow H_{\pi(i),\pi(i)} + H_{i,i} - U_i D_i^{-1} U_i^\top$

Forward Sweep

for $i \in \mathcal{S}^{\mathcal{E}}_{\text{reversed}}$ **do**
  for $j \in \mathcal{N}_i$ **do** ▷ Account for neighbor accelerations
    $\mathbf{u}_i \leftarrow \mathbf{u}_i - (H_{i,j} S_i)^\top \mathbf{a}_j$
  $\dot{\boldsymbol{\nu}}_{Ri} \leftarrow D_i^{-1}\left(\mathbf{u}_i - U_i^\top \mathbf{a}_{\pi(i)}\right)$
  $\mathbf{a}_i \leftarrow \mathbf{a}_{\pi(i)} + S_i \dot{\boldsymbol{\nu}}_{Ri}$

return $\dot{\boldsymbol{\nu}}_R$

For kinematic trees with unary constraints, LCABA reduces to the constrainedABA [12] algorithm and is guaranteed to run in $O(n+m)$ time and memory. For systems with loops, the algorithm inherits the worst-case complexity of

$O(n + dm^2 + m^3)$ in time and $O(n + dm^2)$ in memory from LCABA, which can occur when the majority of the loops are coupled. However, in practice, loops are localized, and the algorithm typically runs in $O(n + m)$ time.

D. Optimized implementation

Similarly to the matrix-free Delassus operator in Section IV, the implementation can be optimized by merging the operators. The backward sweep in Step 1 can be merged with the LCABA backward sweep in Step 2 by directly initializing the spatial forces $\mathbf{f}_i$ in Alg. 4 with the spatial forces resulting from $\boldsymbol{\lambda}_R$ in Alg. 1, thus avoiding the explicit computation of joint torques $\boldsymbol{\tau}_R$. Similarly, the forward sweep in Step 3 can be eliminated by using the link accelerations $\mathbf{a}_i$ already computed during the LCABA forward sweep in Step 2 to directly evaluate the constraint accelerations $\mathbf{a}_R$.

VI. Benchmarks and Discussions

A. Implementation details

The proposed matrix-free Delassus operator and its damped inverse operator have been implemented in C++ within the widely-used open-source rigid-body dynamics library Pinocchio [19]. The operators are evaluated across various robotic platforms and contact scenarios with differing degrees of freedom (DoFs) and constraint dimensions. Realistic contact configurations are generated by simulating these platforms in the Simple physics simulator [31][1] and capturing contact configurations at specific time instances. Benchmarks were conducted on a laptop with an Intel® Core™ Ultra 7 165H CPU running Ubuntu 22.04 LTS. The implementation was compiled using Clang 21.1.1 with optimization flags `-O3 -DNDEBUG -march=native`.

The matrix-free algorithms are compared against state-of-the-art explicit matrix-computation methods based on the LTL-OSIM algorithm [21], [22], whose efficient implementation already exists in Pinocchio. Since our benchmarks include contact-induced internal loops, we do not compare against optimal $O(n + m^2)$ algorithms such as PV-OSIMr [13] and cABA-OSIM [12], as they are restricted to ground contacts. Extending these methods to internal loops is non-trivial and remains a subject for future research. Furthermore, we do not explicitly benchmark against maximal coordinate-based sparse solvers [15], as dedicated algorithms exploiting articulated structures are known to be generally more efficient [20]. However, a benchmark involving a stack of boxes is included, which will illustrate the performance gap; here, our implementation naturally reduces to a maximal coordinate approach since the system consists of independent free-floating objects without articulated joints.

B. Benchmarking on robots with multiple contacts

Benchmarking results are summarized in Table I, which compares the computation times (in microseconds) of the proposed matrix-free methods against the explicit LTL-OSIM baseline [22]. Google benchmarks[2] is employed to profile timings up to 100,000 iterations to obtain stable timings. To benchmark memory footprint, we report both the number of cache lines accessed (in thousands) and the absolute memory footprint (in kB). The former is profiled via Callgrind[3] by simulating a large cache[4] that can hold the entire simulation environment to count unique cache lines read/written, and the latter is computed by aggregating the sizes of all state variables involved in the implementation.

The benchmark scenarios (visualized in Fig. 1) consist of the quadruped robot Unitree Go2, Cassie (a bipedal robot with internal loops), the humanoid robot Unitree G1, a stack of boxes, a pyramid of boxes, and multiple robots falling on the pyramid. Robot DoFs, the number of 3D point contacts (n_c), and total constraint dimension (n_{con}) are also reported. Each pair of contacting surfaces are modeled with up to 4 contact points. The rows corresponding the scenarios where joint friction is enabled are shaded in gray, which increases the constraint count by the robot's DoF.

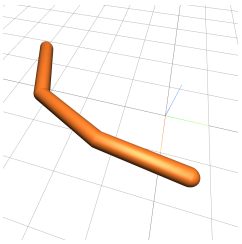
Across all scenarios, the matrix-free operators consistently outperform the explicit baseline. This is expected, as they evaluate matrix-vector products instead of constructing the full Delassus matrix. The performance gap widens as the number of constraints increases, demonstrating better scalability. Note that the Delassus matrix is dense in all the considered scenarios due to interconnectedness via contacts. In the largest scenarios, the matrix-free methods achieve a speedup of nearly $\sim$400$\times$ over the baseline.

Efficient operator re-evaluation: Many applications, such as ADMM-based frictional contact solvers, repeatedly apply the same operator to different vectors or updated damping factors. Our implementation is designed for efficient re-evaluation by storing invariant intermediate terms (e.g., articulated-body inertias) and re-using them during subsequent re-evaluations. The final six columns of Table I detail the costs for re-applying the Delassus operator (Λ^{-1}) and its damped inverse (Λ_R) to a new vector, as well as re-evaluating and applying the inverse for a new damping value (Λ'_R). For small problems, the explicit method is significantly faster for the forward operator re-evaluations as it re-uses the pre-computed dense matrix to perform a matrix-vector product. However, the matrix-free approach exhibits higher performance as the number of constraints grows. Similar trends are observed for inverse operator re-evaluation or when the damping factor is updated (Λ'_R), with the advantage of the explicit method in small problems considerably narrowing due to the increased cost of solving or refactorizing a linear system. For robots with many contacts or when joint friction is enabled, the matrix-free re-evaluation (Λ'_R) always outperforms the explicit baseline.

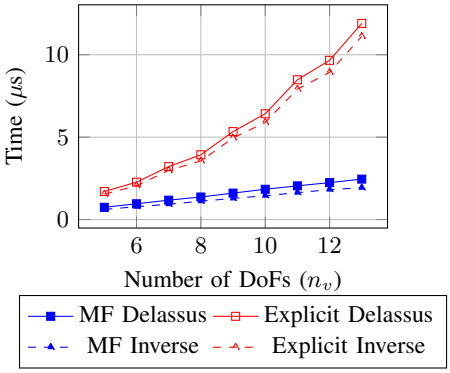
[1] https://github.com/Simple-Robotics/Simple

TABLE I: Computation time (in μs). Cache lines accessed (in thousands) and memory footprint (in kB) are reported in dedicated columns for the Delassus operator and its inverse on various robotic platforms and scenarios (Go2: $n_v = 18$, Cassie: $n_v = 32$, G1: $n_v = 35$, Boxes: $n_v = 180$, Pyramid+Robots: $n_v = 285$). The rows corresponding to the joint friction-enabled scenarios are shaded in grey. The proposed Matrix-Free method is compared against the LTL-OSIM baseline, and the final six columns detail the cost of re-evaluating the Delassus operator (Λ^{-1}), the damped inverse operator (Λ_R) and the damped inverse operator with a new damping factor ((Λ'_R)) on a new input vector.

Scenario	n_c	n_{con}	Delassus Operator (Λ^{-1})						Inverse Delassus Operator (Λ_R)						Re-evaluation timings in μs breakdown					
			Matrix-Free (ours)			Explicit (LTL-OSIM)			Matrix-Free (ours)			Explicit (LTL-OSIM)			Matrix-Free (ours)			Explicit (LTL-OSIM)		
			time	cache	mem	time	cache	mem	time	cache	mem	time	cache	mem	Λ^{-1}	Λ_R	Λ'_R	Λ^{-1}	Λ_R	Λ'_R
Go2 standing	4	12	**1.55**	**0.7**	**26**	3.11	1.9	33	**1.64**	1.5	**29**	3.65	**1.2**	32	0.369	0.389	1.112	**0.027**	**0.135**	**0.402**
Go2 standing	4	24	**1.61**	**0.7**	**28**	4.64	2.7	47	**1.63**	1.6	**32**	5.95	1.6	43	0.399	0.421	**1.171**	**0.052**	**0.305**	1.3
Go2 fallen	9	27	**1.74**	**0.7**	**38**	5.89	3.0	62	**2.42**	2.2	**42**	7.63	**1.9**	56	0.515	0.591	1.911	**0.083**	**0.371**	**1.671**
Go2 fallen	10	42	**1.77**	**0.7**	**45**	8.10	4.3	97	**2.44**	**2.3**	**49**	12.00	2.6	81	0.567	0.62	**1.88**	**0.147**	**0.601**	3.791
Cassie	8	24	**2.94**	**1.0**	**53**	9.27	2.9	84	**4.26**	**2.3**	**69**	10.40	4.6	88	0.77	0.854	3.444	**0.052**	**0.30**	**1.34**
Cassie	8	50	**3.07**	**1.3**	**57**	16.00	4.5	121	**4.51**	**2.4**	**72**	21.72	7.8	143	0.805	0.894	**3.604**	**0.20**	**0.718**	5.798
G1 standing	8	24	**3.66**	**1.3**	**91**	9.37	5.9	195	**3.58**	**3.1**	**110**	10.60	11.0	235	0.919	0.906	2.426	**0.051**	**0.297**	**1.327**
G1 standing	8	53	**3.96**	**1.3**	**94**	18.53	8.0	254	**3.83**	**3.1**	**117**	25.00	16.1	334	1.01	1.00	**2.7**	**0.226**	**0.796**	6.486
G1 fallen	23	69	**4.27**	**1.0**	**47**	25.98	2.0	75	**6.22**	**2.4**	**62**	37.88	3.0	80	1.43	1.46	**5.1**	**0.379**	**1.08**	11.48
G1 fallen	25	101	**4.41**	**1.0**	**50**	42.09	3.3	107	**7.33**	**2.4**	**65**	70.40	5.5	127	1.45	**1.62**	**6.06**	**0.89**	1.80	29
Stack of boxes	120	360	**9.47**	**1.9**	**241**	1214	52.6	1400	**31.38**	**6.7**	**269**	2482	110.7	1850	**5.22**	**7.58**	**30.58**	9.51	14.1	1257.1
Box pyramid	288	864	**14.88**	**3.4**	**755**	5925	347.5	9694	**65.20**	**15.9**	**826**	23092	941.6	15530	**10.7**	**15.5**	**64.7**	94.9	87.4	16822.4
Pyramid + Robots	108	324	**20.34**	**6.3**	**473**	1511	210.9	4533	**37.12**	**17.9**	**730**	2430	530.7	5630	**6.84**	**8.72**	**32.02**	7.36	12.6	906.6



(a) Three-link swimmer.



(b) Computation time vs. n_v.

Fig. 2: Benchmarking computational scaling on a swimmer robot with an increasing number of links. The matrix-free methods scale linearly, unlike the explicit methods.

C. Benchmarking the computational scaling

The computational scaling of the proposed matrix-free methods is further evaluated on a swimmer robot (see Fig. 2a) with an increasing number of links, as shown in Fig. 2b. A three-link swimmer has $n_v = 5$ DoFs due to its planar floating base and two revolute joints; each link makes two 3D point contacts with the ground, resulting in $n_c = 6$ and $n_{con} = 18$. As expected, the proposed matrix-free methods exhibit linear scaling with respect to the number of DoFs, whereas the explicit matrix methods scale superlinearly.

D. Discussions

The LTL-OSIM baseline is additionally slowed by the requirement to explicitly store and factorize the JSIM during the construction of the Delassus matrix. Consequently, recursive low-complexity algorithms [12], [13], [32] that avoid forming the JSIM have an advantage over the LTL-OSIM. However, these methods are currently restricted to ground contacts and cannot accommodate the internal loops common in contact-rich scenarios. Generalizing them to arbitrary loops is non-trivial and remains a subject for future research. Even with such extensions, these explicit methods are unlikely to outperform the proposed matrix-free approach for single evaluations because these algorithms do not utilize the early-elimination strategies that make LCABA-based operators efficient. Furthermore, they exhibit $O(m^2)$ complexity, whereas our matrix-free approach scales linearly.

For scenarios with few constraints relative to robot DoFs, existing explicit methods can be competitive or even faster when amortized over many evaluations (leveraging CPU vectorization) or when the matrix can be reused without re-factorization as seen in the previous benchmarks.

Unlike general sparse-solver approaches [15], our methods exploit the inherent poly-articulated structure to significantly reduce computational overhead. For instance, the G1 robot (30 links) can be modeled in maximal coordinates with 180 DoFs and $150+53 = 203$ constraints. This dimension is comparable to the stack-of-boxes scenario (180 DoFs, 360 constraints), where our method reduces to a maximal coordinate approach due to the lack of joint articulations. The matrix-free inverse operator requires only 7.33 μs for the G1 robot versus 31.38 μs for the boxes, demonstrating the efficiency gains achieved by leveraging the robot's articulated structure.

VII. Conclusion

This letter introduced efficient matrix-free algorithms for computing the Delassus operator and its damped inverse for articulated rigid-body systems with arbitrary kinematic loops and contact constraints. By leveraging recursive rigid-body dynamics, our algorithms achieve linear memory and time complexity in practice. Extensive benchmarking confirms that

[2]https://github.com/google/benchmark

[3]https://valgrind.org/docs/manual/cl-manual.html

[4]Used the command `valgrind --tool=callgrind --instr-atstart=no --cache-sim=yes --D1=33554432,8,64`

this reduced complexity translates to significant speedups compared to state-of-the-art explicit matrix methods, ranging from $2\times$ to nearly $400\times$ depending on the problem scale.

While explicit methods may remain competitive for systems with very few constraints relative to the number of degrees of freedom, our matrix-free approach is vastly superior in contact-rich scenarios. It is also more efficient when downstream applications require frequent updates to damping parameters, as it avoids explicit matrix re-factorizations required by explicit methods. The generality and efficiency of these algorithms facilitate their integration into physics engines for real-time control and simulation of complex robotic systems, particularly on resource-limited hardware. This integration will be the focus of our future work.

ACKNOWLEDGMENT

This work was supported by the European Union's Horizon Europe research and innovation programme through a Marie Skłodowska-Curie Postdoctoral Fellowship (101211945 — ExTRAORDiNary), the ARTIFACT project (GA no.101165695) and through the AGIMUS project (GA no.101070165), and by the French government under the management of Agence Nationale de la Recherche through the project INEXACT (ANR-22-CE33-0007-01), the "PR[AI]RIE-PSAI" AI Cluster (ANR-23-IACL-0008), under the France 2030 program with the references Organic Robotics Program (PEPR O2R), the PIQ program under the management of Agence de Programme du Numérique, Views and opinions expressed are those of the author(s) only and do not necessarily reflect those of the European Union or the European Commission. Neither the European Union nor the European Commission can be held responsible for them.

REFERENCES

[1] M. R. Hestenes and E. Stiefel, "Methods of conjugate gradients for solving linear systems," *Journal of research of the National Bureau of Standards*, vol. 49, no. 6, pp. 409–436, 1952.

[2] D. C. Liu and J. Nocedal, "On the limited memory bfgs method for large scale optimization," *Mathematical programming*, vol. 45, no. 1, pp. 503–528, 1989.

[3] A. Griewank and A. Walther, *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

[4] F. Dellaert, M. Kaess, *et al.*, "Factor graphs for robot perception," *Foundations and Trends® in Robotics*, vol. 6, no. 1-2, pp. 1–139, 2017.

[5] L. Carlone, A. Kim, T. Barfoot, D. Cremers, and F. Dellaert, "Slam handbook: From localization and mapping to spatial intelligence," 2025.

[6] R. Featherstone, "The calculation of robot dynamics using articulated-body inertias," *Int. J. Robot. Res.*, vol. 2, no. 1, pp. 13–30, 1983.

[7] A. Vereshchagin, "Computer simulation of the dynamics of complicated mechanisms of robot-manipulators," *Eng. Cybernet.*, vol. 12, pp. 65–70, 1974.

[8] H. Brandl, R. Johanni, and M. Otter, "A very efficient algorithm for the simulation of robots and similar multibody systems without inversion of the mass matrix," *IFAC Proceedings Volumes*, vol. 19, no. 14, pp. 95–100, 1986.

[9] É. Delassus, "Mémoire sur la théorie des liaisons finies unilatérales," in *Annales scientifiques de l'École normale supérieure*, vol. 34, 1917, pp. 95–179.

[10] B. Brogliato and B. Brogliato, *Nonsmooth mechanics*. Springer, 1999.

[11] C. Duriez, F. Dubois, A. Kheddar, and C. Andriot, "Realistic haptic rendering of interacting deformable objects in virtual environments," vol. 12, no. 1, pp. 36–47, 2005.

[12] A. S. Sathya and J. Carpentier, "Constrained articulated body dynamics algorithms," *IEEE Trans. Robot.*, vol. 41, pp. 430–449, 2025.

[13] A. S. Sathya, W. Decré, and J. Swevers, "Pv-osimr: A lowest order complexity algorithm for computing the delassus matrix," *IEEE Robot. Autom. Lett.*, vol. 9, no. 11, pp. 10 224–10 231, 2024.

[14] D. Baraff, "Linear-time dynamics using lagrange multipliers," in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, 1996, pp. 137–146.

[15] A. Tasora and M. Anitescu, "A matrix-free cone complementarity approach for solving large-scale, nonsmooth, rigid body dynamics," *Computer Methods in Applied Mechanics and Engineering*, vol. 200, no. 5-8, pp. 439–453, 2011.

[16] J. Hwangbo, J. Lee, and M. Hutter, "Per-contact iteration method for solving contact dynamics," *IEEE Robot. Autom. Lett.*, vol. 3, no. 2, pp. 895–902, 2018.

[17] A. S. Sathya and J. Carpentier, "Constrained articulated body algorithms for closed-loop mechanisms," *IEEE Trans. Robot.*, pp. 1–20, 2026.

[18] J. Sherman and W. J. Morrison, "Adjustment of an inverse matrix corresponding to a change in one element of a given matrix," *The Annals of Mathematical Statistics*, vol. 21, no. 1, pp. 124–127, 1950.

[19] J. Carpentier, G. Saurel, G. Buondonno, J. Mirabel, F. Lamiraux, O. Stasse, and N. Mansard, "The pinocchio c++ library: A fast and flexible implementation of rigid body dynamics algorithms and their analytical derivatives," in *2019 IEEE/SICE International Symposium on System Integration (SII)*. IEEE, 2019, pp. 614–619.

[20] R. Featherstone, *Rigid body dynamics algorithms*. Springer, 2014.

[21] ——, "Exploiting sparsity in operational-space dynamics," *Int. J. Robot. Res.*, vol. 29, no. 10, pp. 1353–1368, 2010.

[22] J. Carpentier, R. Budhiraja, and N. Mansard, "Proximal and sparse resolution of constrained dynamic equations," in *Proc. Robot., Sci. Syst.*, 2021.

[23] J. Nocedal and S. Wright, *Numerical optimization*. Springer Science & Business Media, 2006.

[24] C. F. Gauß, "Über ein neues allgemeines grundgesetz der mechanik." 1829.

[25] K. Kreutz-Delgado, A. Jain, and G. Rodriguez, "Recursive formulation of operational-space control," *Int. J. Robot. Res.*, vol. 11, no. 4, pp. 320–328, 1992.

[26] K.-S. Chang and O. Khatib, "Efficient recursive algorithm for the operational space inertia matrix of branching mechanisms," *Advanced Robotics*, vol. 14, no. 8, pp. 703–715, 2001.

[27] P. Wensing, R. Featherstone, and D. E. Orin, "A reduced-order recursive algorithm for the computation of the operational-space inertia matrix," in *Proc. IEEE Int. Conf. Robot. Autom.* IEEE, 2012, pp. 4911–4917.

[28] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *Proc. IEEE/RSJ Int. Conf. Int. Robots. Syst.* IEEE, 2012, pp. 5026–5033.

[29] E. Todorov, "Convex and analytically-invertible dynamics with contacts and constraints: Theory and implementation in mujoco," in *Proc. IEEE Int. Conf. Robot. Autom.* IEEE, 2014, pp. 6054–6061.

[30] R. Tedrake and the Drake Development Team, "Drake: Model-based design and verification for robotics," 2019. [Online]. Available: https://drake.mit.edu

[31] J. Carpentier, Q. L. Lidec, and L. Montaut, "From Compliant to Rigid Contact Simulation: a Unified and Efficient Approach," in *Proc. Robot., Sci. Syst.*, Delft, Netherlands, July 2024.

[32] A. S. Sathya, H. Bruyninckx, W. Decré, and G. Pipeleers, "Efficient constrained dynamics algorithms based on an equivalent lqr formulation using gauss' principle of least constraint," *IEEE Trans. Robot.*, vol. 40, pp. 729–749, 2024.

[33] J. Y. S. Luh, M. W. Walker, and R. P. C. Paul, "On-line computational scheme for mechanical manipulators," *Journal of Dynamic Systems, Measurement, and Control*, vol. 102, no. 2, pp. 69–76, 06 1980.