ACM SIGGRAPH / Eurographics Symposium on Computer Animation 2020
J. Bender and T. Popa
(Guest Editors)

Volume 39 (2020), Number 8

Linear Time Stable PD Controllers for Physics-based Character Animation

Zhiqi Yin[†] KangKang Yin [‡]

Simon Fraser University

Abstract

In physics-based character animation, Proportional-Derivative (PD) controllers are commonly used for tracking reference motions in motor control tasks. Stable PD (SPD) controllers significantly improve the numerical stability of traditional PD controllers and support large gains and large integration time steps during simulation [TLT11]. For an articulated rigid body system with n degrees of freedom, all SPD implementations to date, however, use an $O(n^3)$ dense matrix factorization based method. In this paper, we propose a linear time algorithm for SPD computation, which is based on Featherstone's forward dynamics formulation for articulated rigid body systems in generalized coordinates [Fea14]. We demonstrate the performance advantage of our algorithm by comparing with both the conventional dense matrix factorization based method and an alternative sparse matrix factorization based method. We show that the proposed algorithm provides superior stability when controlling complex models at large time steps. We further demonstrate that our algorithm can improve the learning speed and quality of a Deep Reinforcement Learning (DRL) system for physics-based character animation.

CCS Concepts

• ***Computing methodologies*** → ***Animation; Physical simulation;***

1. Introduction

Physics-based character animation has made significant progresses in recent years. High quality controllers and skills can now be automatically learned to generate motions that are indistinguishable from motion capture data in real-time [YTL18, PALvdP18, PRL*19, BCHF19]. Physics-based characters are yet to be widely adopted by the industry, however, as simulation times for controllable characters still remain as one of the bottlenecks. For example, state-of-the-art Deep Reinforcement Learning (DRL) based algorithms still require hours to days to learn motor skills successfully. Game engines still cannot afford to simulate multiple controllable characters on mobile phones. In the state-of-the-art method DReCon [BCHF19], lower simulation frequency with visible noise had to be used, as higher frequencies lead to significant performance impacts and increases in training time due to the high cost of physics simulation. Algorithms that can accelerate both policy training time and simulation run time for physics-based characters are desirable.

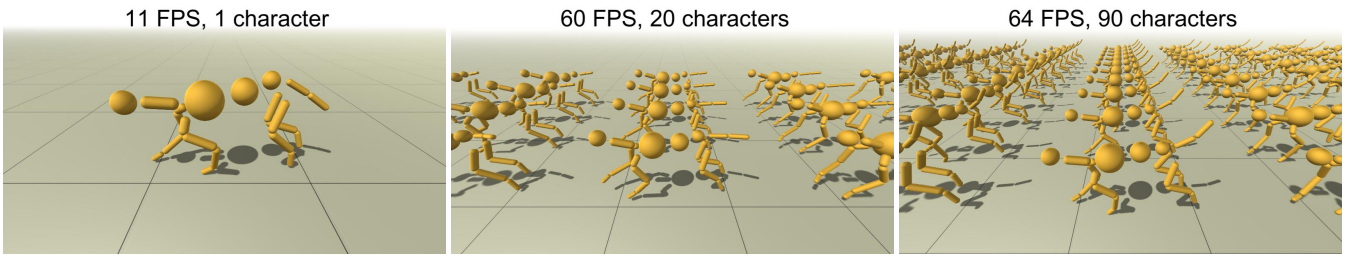
In physics-based character animation, Proportional-Derivative (PD) controllers are commonly used for joint actuation, especially for tracking-based motor control methods where kinematic reference motions are available [ZH02, YLvdP07]. Its major advantage is its simplicity, and its major disadvantage is its instability. Extremely small simulation time steps are required to avoid numerical instability for high gain PD controllers. Therefore it is necessary to address its instability for better simulation accuracy and efficiency.

Stable Proportional-Derivative (SPD) controllers proposed by Tan *et al.* greatly improve the numerical stability of traditional PD controllers by employing the idea of implicit integration [TLT11]. Instead of computing control forces based on the current state, SPD formulates PD controls using the state at the next simulation time step. In practice, SPD formulation allows for fairly high gains at large time steps.

SPD controllers are usually implemented for articulated rigid body systems in generalized coordinates [LGH*18, Cou15]. These implementations solve an $n \times n$ linear system in $O(n^3)$ time based on dense matrix factorization, where n is the total number of Degrees of Freedom (DoFs) in the articulation. However, theoretical time complexity for SPD computation can be easily reduced to $O(nd^2)$ by sparse matrix factorization, where d is the maximum number of DoFs among all branches of the articulation. In the worst case where the kinematic tree becomes a chain, the time complexity falls back to $O(n^3)$. We show in our experiments that cubic time SPD computations significantly slow down when controlling complex articulated characters with large DoFs.

In this paper, we propose a fast and practical SPD computa-

† zhiqi_yin@sfu.ca
‡ kkyin@sfu.ca



(a) *PD* **(b)** *Cubic-time SPD* **(c)** *Our linear-time SPD*

Figure 1: *Performance comparison using a dog model tracking a canter motion with PD controls (a), conventional cubic-time SPD controls (b), and our proposed linear-time SPD controls (c). On the same desktop, we find the largest time steps that we can use to achieve stable tracking for each method. For SPD methods, we also find the maximum number of characters that we can simulate simultaneously in real-time using one CPU thread.*

tion algorithm for articulations parameterized in generalized coordinates. In particular, we derive a Modified Articulated-Body Algorithm (MABA) based on Featherstone's Articulated-Body Algorithm (ABA) for forward dynamics [Fea14]. The proposed algorithm computes SPD controls in worst case $O(n)$ time. As shown in Figure 1, our linear-time algorithm enables simulation and control of many more characters than PD and cubic-time SPD. In our experiments, we demonstrate the performance advantage of MABA over the conventional dense matrix factorization based SPD implementation, as well as its sparse matrix factorization based alternative. We report the simulation FPS (frames per second) of an entire motion tracking system, and then the extra time required for SPD computation. We show that our algorithm provides superior stability for controlling complex character models at large time steps. We further demonstrate that MABA improves the training speed and quality of a DRL system for learning physics-based skills.

2. Background and Related Work

2.1. SPD Formulation for a Single DoF

Standard PD controllers calculate forces based on position and velocity errors at the current time step. At time step n, we denote the position variable as q^n, the target position as $\bar{q}^n$, the velocity as $\dot{q}^n$, and the target velocity as $\bar{\dot{q}}^n$. Then the PD control force τ^n can be calculated as:

$$\tau^n = -k_p(q^n - \bar{q}^n) - k_d(\dot{q}^n - \bar{\dot{q}}^n) \tag{1}$$

where k_p is the stiffness parameter and k_d is the damping parameter. If no velocity tracking is required, Equation 1 can be simplified to:

$$\tau^n = -k_p(q^n - \bar{q}^n) - k_d\dot{q}^n \tag{2}$$

For notation simplicity, hereafter we will formulate different variations of SPD controllers based on the PD control in Equation 2.

SPD computes control forces using state at the next time step instead of the current state [TLT11]:

$$\tau^n = -k_p(q^{n+1} - \bar{q}^{n+1}) - k_d\dot{q}^{n+1} \tag{3}$$

Since future position q^{n+1} and velocity $\dot{q}^{n+1}$ at the next time step are unknown, they are approximated by the first order Taylor expansion:

$$q^{n+1} = q^n + \Delta t\dot{q}^n$$

$$\dot{q}^{n+1} = \dot{q}^n + \Delta t\ddot{q}^n$$

Equation 3 can then be reformulated as:

$$\tau^n = -k_p(q^n + \Delta t\dot{q}^n - \bar{q}^{n+1}) - k_d(\dot{q}^n + \Delta t\ddot{q}^n) \tag{4}$$

Equation 4 is the most popular SPD formulation. Other SPD formulations do exist. For example, the PD formulation in [LYWG13, LPY16, LH17] uses positions at the current time step but velocities at the next time step. Such explicit-proportional implicit-derivative formulation still achieves better stability than the conventional PD. In this paper, we focus on designing and analyzing practical algorithms for SPD computation formulated as Equation 4.

2.2. SPD for Articulations

For an articulated rigid body system with n DoFs parameterized in generalized coordinates, we use n dimensional vectors $\boldsymbol{\tau}$, $\boldsymbol{q}$, $\dot{\boldsymbol{q}}$, $\ddot{\boldsymbol{q}}$ to denote the generalized force, position, velocity, and acceleration of the system. The equation of motion for the articulation can then be expressed as:

$$\boldsymbol{M}\ddot{\boldsymbol{q}} = \boldsymbol{\tau} - \boldsymbol{C} \tag{5}$$

where $\boldsymbol{M}$ is the generalized inertia matrix, and $\boldsymbol{C}$ is the bias force term including centrifugal, Coriolis, and external forces.

The SPD formula for an articulation parameterized in generalized coordinates is a linear equation relating the acceleration $\ddot{\boldsymbol{q}}$ to the force $\boldsymbol{\tau}$, similar to Equation 4:

$$\boldsymbol{\tau} = -\boldsymbol{K}_p(\boldsymbol{q} + \Delta t\dot{\boldsymbol{q}} - \bar{\boldsymbol{q}}) - \boldsymbol{K}_d(\dot{\boldsymbol{q}} + \Delta t\ddot{\boldsymbol{q}}) \tag{6}$$

Here $\boldsymbol{K}_p$ and $\boldsymbol{K}_d$ are the diagonal stiffness and damping matrices. The position $\boldsymbol{q}$ and velocity $\dot{\boldsymbol{q}}$ can be obtained from the simulation state. The acceleration $\ddot{\boldsymbol{q}}$ is unknown and needs to be solved for by a forward dynamics algorithm that obeys Equation 5. Therefore, we substitute $\boldsymbol{\tau}$ in Equation 6 to Equation 5 to solve for the acceleration $\ddot{\boldsymbol{q}}$ first:

$$(\boldsymbol{M} + \boldsymbol{K}_d\Delta t)\ddot{\boldsymbol{q}} = -\boldsymbol{C} - \boldsymbol{K}_p(\boldsymbol{q} + \Delta t\dot{\boldsymbol{q}} - \bar{\boldsymbol{q}}) - \boldsymbol{K}_d\dot{\boldsymbol{q}} \tag{7}$$

Then $\boldsymbol{\tau}$ can be calculated by substituting $\ddot{\boldsymbol{q}}$ into either Equation 5 or 6.

2.3. Solving SPD by Matrix Factorization

The key step in solving SPD is to solve the linear system in Equation 7. Conventionally, it is solved directly by numerical methods which result in $O(n^3)$ running time in general. We briefly review these implementations in this section.

The most straight forward SPD implementation uses direct matrix inversion to solve the system [LGH*18, LPLL19, PRL*19]. This approach takes $O(n^3)$ time and may encounter numerical stability issues. A better approach is to apply Cholesky factorization LDL^T or LL^T on $\boldsymbol{M}+\boldsymbol{K}_d\Delta t$. Such factorization is applicable because the inertia matrix $\boldsymbol{M}$ is symmetric positive definite, and $\boldsymbol{K}_d\Delta t$ is a diagonal matrix with non-negative elements. This approach still takes $O(n^3)$ time to compute because it treats $\boldsymbol{M}+\boldsymbol{K}_d\Delta t$ as a dense matrix. We shall refer to this approach as **dense factorization** (DF) hereafter. Due to its simplicity, the DF method is widely adopted by both the research community and the industry [Cou15, PALvdP18, PBYVDP17, PBVdP16, YK19].

To reduce the time complexity of the DF method, our first thought is to examine the inertia matrix $\boldsymbol{M}$. Featherstone's dynamics formulation of articulated rigid body systems results in branch-induced sparsity of $\boldsymbol{M}$ [Fea14]. More specifically, for tree-like articulations that contain no loops, $\boldsymbol{M}_{ij}$ is non zero if and only if node i is an ancestor or a decedent of node j. We thus can employ a topology dependent sparse LDL^T or LL^T factorization algorithm [Fea14] for $\boldsymbol{M}+\boldsymbol{K}_d\Delta t$, because again the additional diagonal matrix $\boldsymbol{K}_d\Delta t$ does not change the sparsity of the inertia matrix $\boldsymbol{M}$. We shall refer to this approach as **sparse factorization** (SF) hereafter. The time complexity of the SF method is $O(nd^2)$, where d is the maximum number of DoFs among all branches of the articulation tree. In the worst case where the tree becomes a chain, SF degenerates into DF and requires $O(n^3)$ time as well, as the inertia matrix is not sparse anymore. To the best of our knowledge, we are the first to implement the sparse factorization method for SPD computation.

2.4. Articulated-Body Forward Dynamics Algorithm

Forward dynamics is the problem of solving accelerations from the equation of motion indicated by Equation 5. From Equations 5 and 7 we can see that both forward dynamics and SPD control involve solving for the accelerations from a linear system. The cubic time SPD computation can significantly slow down the performance for systems with large DoFs, if linear time forward dynamics algorithms are used for simulation [Fea14, Bar96]. One well-known linear time forward dynamics algorithm from the robotics literature is Featherstone's Articulated-Body Algorithm (ABA) [Fea14]. This algorithm is sometimes abbreviated as ABM (Articulated Body Method) [Kok04], but we follow Featherstone's original acronym.

ABA solves for the accelerations recursively in linear time without explicitly solving the $n \times n$ closed-form linear system. Its efficiency comes from the use of recurrence relations. A similar and more well known case is the Newton-Euler inverse dynamics algorithm: the recursive Newton-Euler is much faster than its non-recursive predecessor. ABA calculates the forward dynamics of a kinematic tree by three passes over the tree: an outward pass (root to leaves) to calculate velocity and bias terms; an inward pass to calculate articulated-body inertias and bias forces; and a second outward pass to calculate the accelerations.

ABA does not directly handle constraints such as collisions and joint limits. Additional mechanisms, either ABA-based or independent, are required to impose penalty, impulse, or constraint forces to enforce such constraints. For example, a modified version of ABA is used to compute acceleration constraint matrices in [Kok04]. Another ABA-based procedure is used for computing and propagating impulse responses through articulated bodies [Mir96]. In our work, we solely modify ABA for the purpose of SPD computation, which is complementary to other works that integrate other types of constraints into ABA-based algorithms.We will discuss more ABA details in Section 3.1, and derive our linear time SPD implementation, which is based on ABA, in Section 3.2 .

3. Modified Articulated-Body Algorithm

Since SPD computation has a similar structure as forward dynamics, we propose to solve SPD in linear time by adapting the Articulated-Body Algorithm. The fundamental intuition is that solving SPD accelerations is simply computing forward dynamics while enforcing the SPD constraints on the control forces indicated by Equation 6. We name such a linear time SPD algorithm as Modified Articulated-Body Algorithm (MABA). We will derive MABA by enforcing the SPD control force constraints where necessary in the ABA derivation. MABA shares the same algorithm structure as ABA, and therefore can be computed with the same set of tree traversals as in ABA.

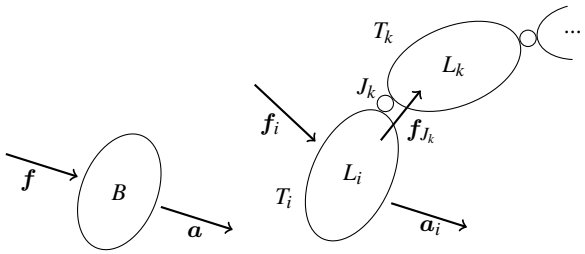
3.1. ABA Preliminaries

In this section, we review a few key concepts and the three tree traversal passes of ABA, starting with necessary symbol definitions. We refer readers to [Fea14] for more detailed explanations. However, ABA was developed by roboticists and its original derivation as presented in [Fea14] is hard to understand for a typical graphics audience [Bar96, Mir96]. We encourage readers who do not have any knowledge on spatial notations to first follow these excellent tutorials [Fea10a, Fea10b]. Interested readers are further referred to an ABA derivation from basic principles of rigid body dynamics [Mir96], which should be easier to understand.

Let T be a kinematic tree with m links L_1 to L_m, where L_1 is the root. The subtree rooted at L_i is denoted as T_i. The set of link indices of L_i's children is denoted as $\mu(i)$. The index of L_i's parent link is denoted as $\lambda(i)$. The inbound joint of L_i is denoted as J_i.

For each link L_i, we denote its 6D spatial motion vector as $\boldsymbol{v}_i$, which includes both the linear and angular motion of the rigid body. Similarly, we denote its 6D spatial force vector as $\boldsymbol{f}_i$, which includes both the linear force and the angular torque. Spatial acceleration of L_i is denoted as $\boldsymbol{a}_i$. Generalized joint variables for joint J_i are denoted as $\boldsymbol{q}_i$, $\dot{\boldsymbol{q}}_i$, $\ddot{\boldsymbol{q}}_i$ and $\boldsymbol{\tau}_i$. The motion subspace matrix $\boldsymbol{S}_i$ relates the generalized coordinates to spatial quantities as follows:

$$\boldsymbol{v}_{J_i} = \boldsymbol{S}_i\dot{\boldsymbol{q}}_i \tag{8}$$



(a) *a single body* **(b)** *an articulated body*

Figure 2: *Illustration of a single body and an articulated body.*

$$\boldsymbol{\tau}_i = \boldsymbol{S}_i^T \boldsymbol{f}_{J_i} \tag{9}$$

where $\boldsymbol{v}_{J_i}$ and $\boldsymbol{f}_{J_i}$ are the spatial velocity and spatial force of joint J_i.

Figure 2 illustrates the concept of a single body system and an articulated-body system. When applying a spatial force $\boldsymbol{f}$ on a single rigid body B as shown in Figure 2a, the acceleration $\boldsymbol{a}$ of body B is determined by the Newton-Euler equation:

$$\boldsymbol{f} = \boldsymbol{I}\boldsymbol{a} + \boldsymbol{p} \tag{10}$$

where $\boldsymbol{I}$ is a 6×6 single-body spatial inertia matrix including both the mass and moment of inertia, and $\boldsymbol{p}$ is a bias force term including the centrifugal and Coriolis forces.

An articulated body is a system of rigid bodies articulated by joints as shown in Figure 2b. When spatial force $\boldsymbol{f}_i$ is applied on link L_i, we cannot compute the acceleration $\boldsymbol{a}_i$ from Equation 10 anymore because of the unknown joint force $\boldsymbol{f}_{J_k}$ from the subtree T_k. However, $\boldsymbol{f}_i$ and $\boldsymbol{a}_i$ still satisfy a linear equation

$$\boldsymbol{f}_i = \boldsymbol{I}_i^A \boldsymbol{a}_i + \boldsymbol{p}_i^A \tag{11}$$

where unknowns $\boldsymbol{I}_i^A$ and $\boldsymbol{p}_i^A$ depend on the whole structure of the articulated body. $\boldsymbol{I}_i^A$ and $\boldsymbol{p}_i^A$ are termed the articulated-body inertia and bias force of T_i. L_i is called the *handle* of the articulated-body system, as Equation 11 describes the acceleration response of body L_i jointed with all bodies in T_i. $\boldsymbol{I}_i^A$ and $\boldsymbol{p}_i^A$ have the following two properties that enable the ABA implementation by three tree traversals:

1. $\boldsymbol{I}_i^A$ and $\boldsymbol{p}_i^A$ can be computed recursively from the leaves to the root. Specifically, $\boldsymbol{I}_i^A$ and $\boldsymbol{p}_i^A$ can be computed from $\boldsymbol{I}_j^A$ and $\boldsymbol{p}_j^A$ where $j \in \mu(i)$.
2. Once we know $\boldsymbol{I}_i^A$ and $\boldsymbol{p}_i^A$ for each i, all joint and link accelerations can be computed recursively from the root to the leaves.

ABA requires three passes of tree traversal of the articulation tree. These tree traversals solve for the auxiliary variables and final accelerations following the topological order of the tree:

- **Pass 1** (top down): Compute auxiliaries including joint and link velocities, and single-body centrifugal, Coriolis, and external forces, from the root to the leaves.
- **Pass 2** (bottom up): Compute articulated-body inertias and bias forces based on auxiliaries computed in Pass 1, from the leaves to the root.
- **Pass 3** (top down): Compute joint and link accelerations based on the articulated-body inertias and bias forces computed in Pass 2, from the root to the leaves.

3.2. MABA Derivation

Our MABA derivation is similar to the standard ABA derivation, which makes three traversal passes of the kinematic tree. In particular, MABA shares the exact same Pass 1 as ABA, for which we omit the details in this paper and refer the interested readers to [Fea14]. Hereafter we assume quantities computed by Pass 1 are known, including joint and link velocities, and single-body bias forces. We will first derive Pass 3 then Pass 2, since derivation for Pass 2 requires a relationship between joint accelerations and link accelerations derived in Pass 3. For notation simplicity and ease of comprehension, we omit necessary coordinate transformations in this section. For ease of reimplementation though, we list the complete set of equations with proper coordinate transformations in Appendix A.

3.2.1. MABA Pass 3

Pass 3 of MABA does a similar job as Pass 3 of ABA, with additional SPD constraints taken into account when accumulating accelerations from the root to the leaves. The root link acceleration serves as the base case of the recursion. For fixed-base articulations, the root acceleration is set to zero. For floating-base articulations, we treat the root link as connected to a fixed base by a virtual 6-DoF joint. Now we derive recurrent formulas to solve

- (Objective 1): $\boldsymbol{a}_i$ based on $\boldsymbol{a}_{\lambda(i)}$ and $\ddot{\boldsymbol{q}}_i$, and
- (Objective 2): $\ddot{\boldsymbol{q}}_i$ based on $\boldsymbol{a}_{\lambda(i)}$.

For Objective 1, we consider the articulated body $T_{\lambda(i)}$ shown in Figure 3. The joint velocity $\boldsymbol{v}_{J_i}$ is the relative spatial velocity of L_i with respect to $L_{\lambda(i)}$. From Equation 8 we have

$$\boldsymbol{v}_{J_i} = \boldsymbol{v}_i - \boldsymbol{v}_{\lambda(i)} = \boldsymbol{S}_i \dot{\boldsymbol{q}}_i. \tag{12}$$

By differentiating the above equation, we get

$$\boldsymbol{a}_i - \boldsymbol{a}_{\lambda(i)} = \boldsymbol{S}_i \ddot{\boldsymbol{q}}_i + \dot{\boldsymbol{S}}_i \dot{\boldsymbol{q}}_i. \tag{13}$$

So the link acceleration $\boldsymbol{a}_i$ can be computed from the parent link acceleration $\boldsymbol{a}_{\lambda(i)}$ and the inbound joint acceleration $\ddot{\boldsymbol{q}}_i$, achieving Objective 1 of Pass 3.

For Objective 2, we take the SPD constraints into consideration. After Pass 2, the articulated-body inertia $\boldsymbol{I}_i^A$ and the bias force $\boldsymbol{p}_i^A$ for T_i are knowns that satisfy

$$\boldsymbol{f}_i = \boldsymbol{I}_i^A \boldsymbol{a}_i + \boldsymbol{p}_i^A \tag{14}$$

where $\boldsymbol{f}_i$ is the net external force acting on articulated body T_i through handle L_i. Such force comes only from joint J_i, therefore,

$$\boldsymbol{f}_i = \boldsymbol{f}_{J_i} \tag{15}$$

Now we expand Equation 9 by Equation 15, 14 and 13:

$$\begin{aligned} \boldsymbol{\tau}_i &= \boldsymbol{S}_i^T \boldsymbol{f}_{J_i} \\ &= \boldsymbol{S}_i^T \left(\boldsymbol{I}_i^A \boldsymbol{a}_i + \boldsymbol{p}_i^A \right) \\ &= \boldsymbol{S}_i^T \left(\boldsymbol{I}_i^A \left(\boldsymbol{a}_{\lambda(i)} + \boldsymbol{S}_i \ddot{\boldsymbol{q}}_i + \dot{\boldsymbol{S}}_i \dot{\boldsymbol{q}}_i \right) + \boldsymbol{p}_i^A \right) \end{aligned} \tag{16}$$

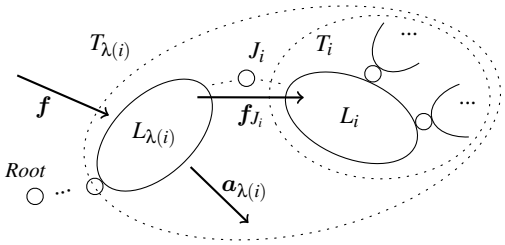


Figure 3: *Articulated-body $T_{\lambda(i)}$ with handle $L_{\lambda(i)}$.*

Equation 16 is a linear equation relating the generalized joint force τ_i and the generalized joint acceleration $\ddot{q}_i$. Here we must enforce that τ_i equals the SPD control force. Following Equation 4, the SPD constraint for a single joint J_i can be written as

$$\tau_i = -K_{pi}(q_i + \Delta t\dot{q}_i - \bar{q}_i) - K_{di}(\dot{q}_i + \Delta t\ddot{q}_i) \tag{17}$$

Combining Equation 16 and 17, we get

$$\ddot{q}_i = (S_i^T I_i^A S_i + K_i)^{-1}\left(Q_i - S_i^T\left(I_i^A(a_{\lambda(i)} + \dot{S}_i\dot{q}_i) + p_i^A\right)\right) \tag{18}$$

where we define

$$K_i = K_{di}\Delta t, \tag{19}$$

$$Q_i = -K_{pi}(q_i + \Delta t\dot{q}_i - \bar{q}_i) - K_{di}\dot{q}_i \tag{20}$$

Equation 18 achieves Objective 2, as joint acceleration $\ddot{q}_i$ can be computed from the parent link acceleration $a_{\lambda(i)}$. Since we have enforced the SPD constraint in the derivation, accelerations computed by this algorithm strictly follow the SPD control. Now we have completed the derivation of MABA Pass 3.

3.2.2. MABA Pass 2

Pass 2 of MABA recursively accumulates articulated-body inertias and bias forces from the leaves to the root. All leaf links of the kinematic tree form the recursion base cases. As leaf links have no children, their articulated-body inertia and bias force equal to their single-body counterparts. Next we need to derive the recurrent formulas to solve for a link's articulated-body inertia and bias force from those of its children. We first consider the case shown in Figure 3 where $L_{\lambda(i)}$ has only one child L_i. Later we will generalize the computation for links with multiple children.

To compute $I_{\lambda(i)}^A$ and $p_{\lambda(i)}^A$ from I_i^A and p_i^A, the strategy is to derive an equation of the form

$$f = Aa_{\lambda(i)} + b \tag{21}$$

where f is the net spatial force acting on $L_{\lambda(i)}$ external to the articulated body $T_{\lambda(i)}$. Then the coefficients A and b will correspond to the desired quantities $I_{\lambda(i)}^A$ and $p_{\lambda(i)}^A$.

We first consider the spatial forces acting on the single body $L_{\lambda(i)}$. Apart from f, there is also a joint reaction force $-f_{J_i}$ acting on $L_{\lambda(i)}$. Then from the Newton-Euler equation similar to Equation 10:

$$f - f_{J_i} = I_{\lambda(i)}a_{\lambda(i)} + p_{\lambda(i)} \tag{22}$$

where $I_{\lambda(i)}$ and $p_{\lambda(i)}$ are the single-body spatial inertia and bias force terms. Expanding Equation 22 with Equations 15, 14, 13 and 18, we get:

$$\begin{aligned} f &= I_{\lambda(i)}a_{\lambda(i)} + p_{\lambda(i)} + f_{J_i} \\ &= I_{\lambda(i)}a_{\lambda(i)} + p_{\lambda(i)} + I_i^A a_i + p_i^A \\ &= I_{\lambda(i)}a_{\lambda(i)} + p_{\lambda(i)} + I_i^A(a_{\lambda(i)} + S_i\ddot{q}_i + \dot{S}_i\dot{q}_i) + p_i^A \\ &= I_{\lambda(i)}a_{\lambda(i)} + p_{\lambda(i)} + I_i^A(a_{\lambda(i)} + S_i(S_i^T I_i^A S_i + K_i)^{-1} \\ &\quad \left(Q_i - S_i^T\left(I_i^A(a_{\lambda(i)} + \dot{S}_i\dot{q}_i) + p_i^A\right)\right) + \dot{S}_i\dot{q}_i) + p_i^A \end{aligned} \tag{23}$$

By rearranging terms in the above equation, we can achieve the desired form in Equation 21. If we define

$$I_i^a = I_i^A - I_i^A S_i(S_i^T I_i^A S_i + K_i)^{-1} S_i^T I_i^A, \tag{24}$$

$$p_i^a = p_i^A + I_i^a\dot{S}_i\dot{q}_i + I_i^A S_i(S_i^T I_i^A S_i + K_i)^{-1}(Q_i - S_i^T p_i^A) \tag{25}$$

then we can get the formulas for $I_{\lambda(i)}^A$ and $p_{\lambda(i)}^A$ as follows:

$$I_{\lambda(i)}^A = I_{\lambda(i)} + I_i^a, \tag{26}$$

$$p_{\lambda(i)}^A = p_{\lambda(i)} + p_i^a \tag{27}$$

Similarly, Equations 26 and 27 can be generalized for links with multiple children. For an arbitrary link L_i:

$$I_i^A = I_i + \sum_{j\in\mu(i)} I_j^a, \tag{28}$$

$$p_i^A = p_i + \sum_{j\in\mu(i)} p_j^a. \tag{29}$$

We therefore use Equations 28 and 29 to compute the articulated-body inertias and bias forces from the leaves to the root in MABA Pass 2. Again, a complete set of equations for the whole algorithm is given in Appendix A.

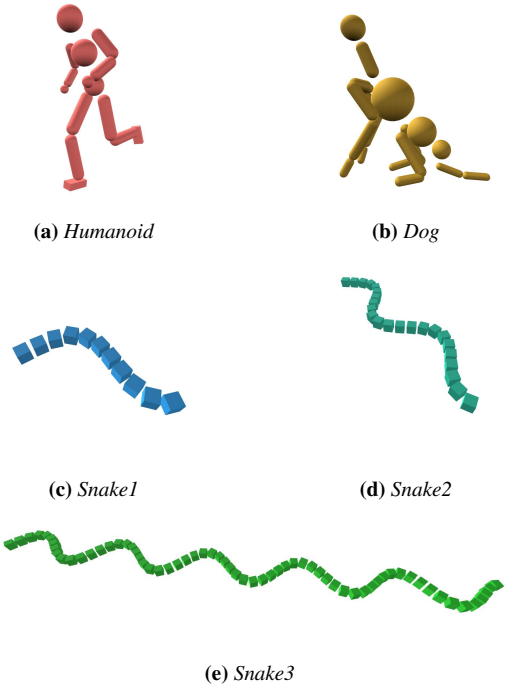
3.3. Algorithm Complexity

MABA and ABA share most of their essential computations. The key difference is that MABA requires the extra computation of K_i and Q_i by Equation 19 and 20, which takes at most $O(n)$ time. Since ABA runs in worst case $O(n)$ time, we conclude that MABA runs in worst case $O(n)$ time as well.

3.4. Practical Implementation

MABA can be directly implemented by modifying essentially just two lines of the original ABA equations, for which more details are given in Appendix A. This makes it easy to embed SPD control directly into simulation systems that already use ABA for forward dynamics, such as PhysX [NVI19], Bullet [Cou15] and DART [LGH*18]. In such an embedded implementation, forward dynamics accelerations are computed directly under constraints imposed by SPD, without the actual control forces explicitly calculated. Such implementation is simple to code on top of ABA, and incurs negligible cost as we will show in our experiments.

As MABA is simply ABA with SPD constraints satisfied, our



(a) *Humanoid* **(b)** *Dog* **(c)** *Snake1* **(d)** *Snake2* **(e)** *Snake3*

Figure 4: *Character models used in our experiments.*

embedded implementation of MABA naturally works with additional constraint solvers, either ABA-based or independent. Simulation engines with independent constraint solvers can call MABA instead of ABA to solve forward dynamics with SPD controls [NVI19]. ABA-based solvers just need to incorporate our minor modifications into their algorithm to use SPD controllers [Kok04]. For our experiments done on PhysX, we do not need to do anything other than replacing the original ABA code with our MABA code to incorporate contact and ground reaction forces into the simulation, as PhysX handles these constraints independently of ABA.

4. Experiments

In this section we validate the accuracy, stability, and performance of MABA in motion tracking tasks and Deep Reinforcement Learning (DRL) tasks. We also compare MABA with the dense factorization (DF) method and the sparse factorization (SF) method. The DF method is implemented using the dense LL^T factorization provided by the Eigen library [GJ*10]. The SF implementation strictly follows the pseudo-code for sparse LL^T factorization presented in [Fea14].

Our experiments are performed on a Dell Precision 7920 Tower workstation with an Intel Xeon Gold 6128 CPU (3.4 GHz, 12 threads) and a GeForce GTX 1080 Ti GPU. NVIDIA PhysX (version 2019.8) is used as our physics simulation engine. Five simulated virtual character models of different complexity are studied in our experiments, including a humanoid, a dog, and three snakes of different length. We visualize these models in Figure 4 and list their important model parameters in Table 1. The humanoid model

Model	Humanoid	Dog	Snake1	Snake2	Snake3
n	34	72	36	72	195
d	13	24	36	72	195

Table 1: *Model parameters: n is the degrees of freedom; and d is the maximum number of DoFs from the root to the leaves.*

and motions are obtained from Peng *et al.* [PALvdP18]. The dog model and motions are obtained from Zhang *et al.* [ZSKS18].

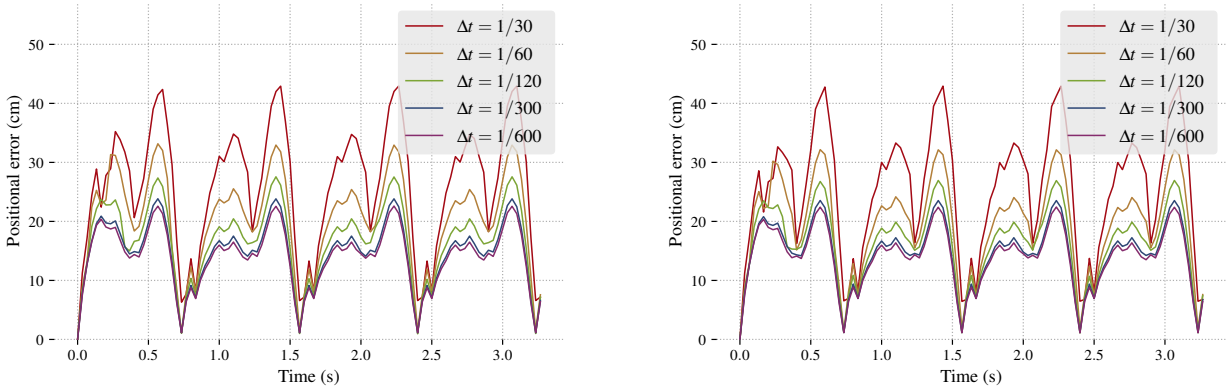
4.1. Accuracy and Stability

We investigate the accuracy and stability of MABA through motion tracking tasks on the humanoid and dog models. More specifically, we use SPD controllers to track both the root and internal joints in predefined motion trajectories. As the root is controlled by external "hand-of-God" forces, such tasks are quasi-static in nature and used purely for accessing the tracking performance. Tracking accuracy is measured by end-effector to root errors between the reference and simulated vectors. Here we use a fixed set of SPD parameters in our experiment: $k_p = 20000, k_d = 2000$ for the root, and $k_p = 75000, k_d = 4000$ for all the internal joints. We show the tracking errors using different simulation time steps in Figure 5 and 6. Figure 5 corresponds to the tracking errors measured by the right ankle to root vector of the humanoid model during a running motion. Figure 6 corresponds to the tracking errors measured by the right front toe to root vector of the dog model during a cantering motion. We only compare the DF method and MABA here, as SF only differs from DF in terms of efficiency, but not accuracy nor stability.

Figure 5 shows that MABA and the dense factorization method produce near identical accuracy curves on the humanoid model. Both controllers are stable for time steps up to $\Delta t = 1/30\ s$. Figure 6 demonstrates that MABA achieves significantly better accuracy and stability than DF on the dog model. MABA is stable for all tested time steps, while DF diverges for $\Delta t \geq 1/60\ s$. For $\Delta t = 1/120\ s$, although DF does not fail, it produces visibly larger tracking errors than MABA. For small time steps $\Delta t \leq 1/300\ s$, the two implementations become comparable. Based on these experiments, we conclude that MABA delivers better stability for complex models at large time steps. Even though different implementations solve the same SPD formulation, MABA generally produces less numerical errors than matrix factorization due to its computational simplicity. Therefore MABA degrades more gracefully than matrix factorization based methods for large integration time steps and complex models, which tend to amplify the accumulated numerical errors.

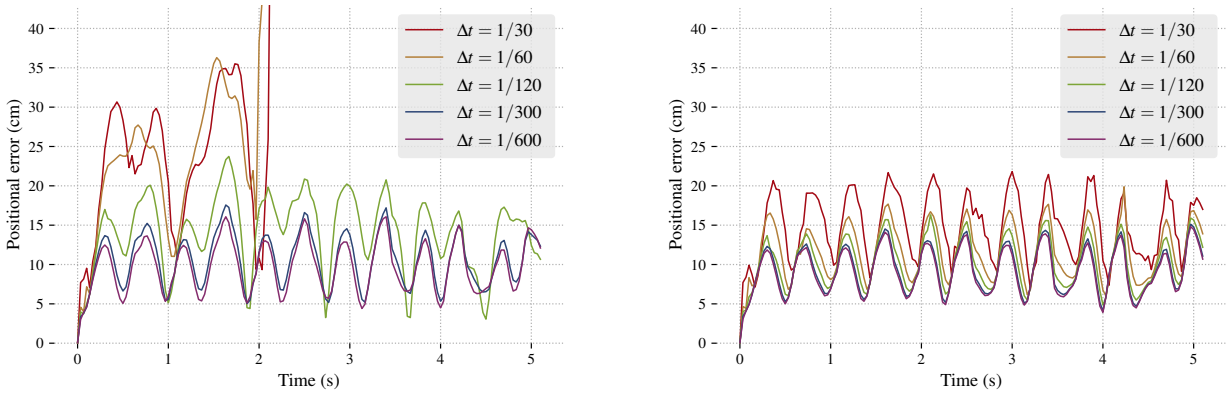
4.2. Simulation Performance

We compare the runtime performance of different SPD implementations on four character models of different complexity using the same motion tracking tasks described in Section 4.1. Performance is measured by the number of simulated Frames Per Second (FPS) on a single CPU thread. We set $\Delta t = 1/30\ s$ for the humanoid and snake models. We use $\Delta t = 1/240\ s$ for the dog model, as



(a) *Dense factorization method* **(b)** *MABA*

Figure 5: *SPD tracking errors measured at the right ankle of the humanoid model in a running motion. The smaller the simulation time step is, the smaller the tracking errors are. MABA produces comparable results as the dense factorization method.*



(a) *Dense factorization method* **(b)** *MABA*

Figure 6: *SPD tracking errors measured at the right front toe of the dog model in a cantering motion. The smaller the simulation time step is, the smaller the tracking errors are. MABA produces better results than the dense factorization method.*

factorization-based methods cannot achieve stable simulation when using large time steps as shown in Figure 6.

We record the simulation FPS for different combinations of models, motions and SPD implementations (DF, SF, and MABA). Table 2 shows that MABA is significantly more efficient than DF, which is currently the only option in both research and industry to the best of our knowledge. MABA is also always faster than SF. SF achieves better performance than DF when controlling models with low DoFs, or models with shallow tree structures. However, SF is less efficient than DF for models with high DoFs or deep tree structures. Our snake models correspond to the worst case scenario for SF, as their kinematic trees degenerate to chains so the time complexity becomes $O(n^3)$ just as in DF. In fact, SF runs slower than DF due to necessary overhead incurred for sparse factorization.

Iterative methods, such as Preconditioned Conjugate Gradient (PCG), can also be used to solve this problem. However, as our matrix is neither very large nor sparse, PCG does not provide any performance gain over Cholesky factorization. As shown in Table 2, PCG results in comparable speed to DF. There is a better mechanism proposed in [WO82] that uses PCG to achieve $O(n^2)$ performance. Our proposed MABA is much simpler and faster, however, so we will not include PCG-based methods in further comparative studies.

We also report the percentages of extra time required for SPD computation over total simulation time for different methods. Table 3 shows that matrix factorization-based methods can dominate the simulation process for complex models while MABA solves SPD with consistently negligible cost. Therefore, we recommend MABA for SPD computation wherever possible. We note that for

Model	Motion	DF	PCG	SF	MABA
Humanoid	walk	13,752	14,108	17,083	20,224
	run	14,109	14,409	17,048	20,523
	cartwheel	13,729	14,436	16,962	20,213
	backflip	13,827	14,537	16,943	19,387
Dog	pace	6,572	4,838	8,764	11,804
	trot	6,551	4,682	8,969	11,540
	canter	6,498	4,553	8,738	11,700
Snake1	slither	11,897	11,935	12,259	16,444
Snake2	slither	5,099	5,069	3,979	8,334
Snake3	slither	1,036	944	417	3,036

Table 2: *Simulation FPS (frames per second) of different SPD implementations: DF (dense factorization), PCG (preconditioned conjugate gradient with Jacobi preconditioning), SF (sparse factorization), and MABA.*

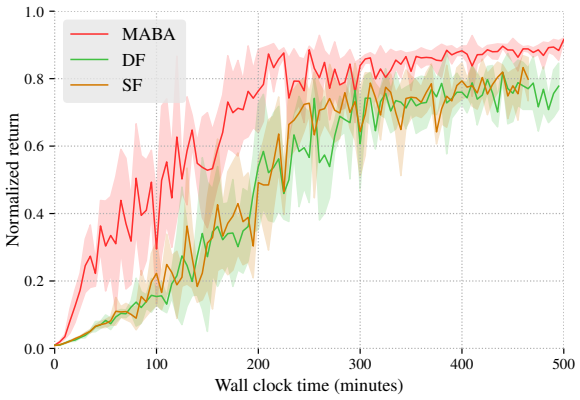
Model	DF(%)	SF(%)	MABA(%)
Humanoid	31.1	18.2	2.1
Dog	44.3	25.7	3.6
Snake1	28.0	28.3	1.7
Snake2	39.9	53.9	2.5
Snake3	64.4	86.3	3.7

Table 3: *Percentage of extra time required by SPD computation over total simulation time.*

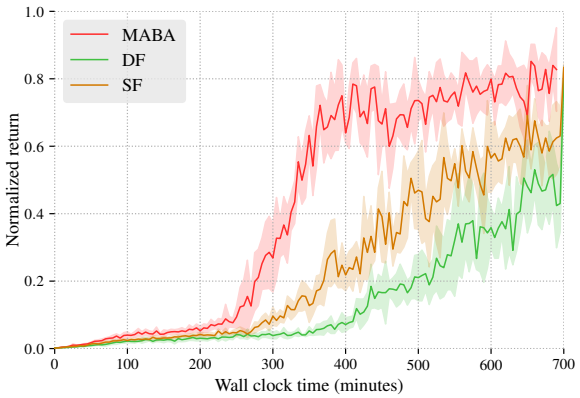
Table 3, reported performances are averaged values across all available motions for the humanoid and dog models.

4.3. DRL Training Performance

The motion tracking task that we have used for testing and comparison so far is only a quasi physics-based method, as it tracks the root joint with a hand-of-God type control. In order to see the effect of different solvers in true physics-based learning and control systems, where only internal joint actuation and ground reaction forces are used for control of the full body, we employ these SPD solvers within a state-of-the-art Deep Reinforcement Learning (DRL) framework named DeepMimic [PALvdP18]. DeepMimic employs both an imitation reward and a task reward to encourage physics-based characters to learn high-quality motor skills. We re-implement DeepMimic on NVIDIA PhysX and train several skills on our machine with 12 CPU threads in parallel and one GPU. We use exactly the same network architecture and training parameters as those given in the original implementation [PALvdP18]. Figure 7 reports the learning curves in wall-clock time. Our results show that for both the humanoid and the dog model, MABA not only improves the learning speed, but also helps DeepMimic converge to better solutions. This is because first MABA saves training time by directly reducing the SPD computational cost; and second DeepMimic requires less training samples when using MABA as its greater stability helps reduce the learning difficulty.



(a) *Humanoid run*



(b) *Dog canter*

Figure 7: *DRL learning curves measured in wall-clock time using different SPD implementations. As DeepMimic is non-deterministic, the curves we show are the average of five training runs. The shaded region indicates the standard deviation.*

5. Conclusion and Discussion

We have presented the Modified Articulated-Body Algorithm for SPD computation of articulated rigid body systems parameterized in generalized coordinates. We show that MABA runs in linear time, which is the theoretical minimum under the presented SPD formulation. We demonstrate the performance and stability advantages of MABA for physics-based character animation. Since SPD controllers are fundamental components in many time-critical or time-consuming systems, such as computer games and DRL-based algorithms, our proposed algorithm could potentially benefit a wide range of applications and research.

Our current MABA implementation is embedded into an ABA forward dynamics solver by directly modifying the ABA code in the PhysX simulation engine. A standalone implementation is necessary when the SPD control forces need to be computed explicitly. For example, when the SPD control forces need to be monitored and tailored before being sent to the forward dynamics simulation.

The standalone implementation is roughly equivalent to running a linear time forward dynamics solver at each simulation time step. So it still incurs $O(n)$ cost and is faster than a cubic time SPD implementation. The forward dynamics solver needs to be called separately after the standalone MABA, so the resultant speed and accuracy of the standalone MABA implementation will be inferior to the embedded MABA, due to the two passes of the ABA-type solver and error accumulations during this process.

In the future, we plan to investigate linear time SPD controllers for articulations parameterized in full coordinates. Such an algorithm should be developed on top of a linear forward dynamics algorithm in full coordinates, such as [Bar96], to maximize the potential performance gain. It is also interesting to explore parallel algorithms for solving SPD, so that GPU acceleration can be utilized.

Acknowledgements We would like to thank Jie Tan and Michiel van de Panne for their suggestions on an early draft of this paper. We also thank the anonymous reviewers for their constructive feedback. This project is partially supported by NSERC Discovery Grants Program RGPIN-06797 and RGPAS-522723.

References

[Bar96] BARAFF D.: Linear-time dynamics using lagrange multipliers. In *SIGGRAPH* (1996), pp. 137–146. 3, 9

[BCHF19] BERGAMIN K., CLAVET S., HOLDEN D., FORBES J. R.: DReCon: data-driven responsive control of physics-based characters. *ACM Transactions on Graphics (TOG) 38*, 6 (2019), 1–11. 1

[Cou15] COUMANS E.: Bullet physics simulation. p. 1. 1, 3, 5

[Fea10a] FEATHERSTONE R.: A beginner's guide to 6-D vectors (part 1). *IEEE Robotics Automation 17*, 3 (2010), 83–94. 3

[Fea10b] FEATHERSTONE R.: A beginner's guide to 6-D vectors (part 2). *IEEE Robotics Automation 17*, 4 (2010), 88–99. 3

[Fea14] FEATHERSTONE R.: *Rigid body dynamics algorithms*. Springer, 2014. 1, 2, 3, 4, 6, 9

[GJ*10] GUENNEBAUD G., JACOB B., ET AL.: Eigen v3, 2010. 6

[Kok04] KOKKEVIS E.: Practical physics for articulated characters. In *Game Developers Conference* (2004), vol. 2004. 3, 6

[LGH*18] LEE J., GREY M., HA S., KUNZ T., JAIN S., YE Y., SRINIVASA S., STILMAN M., LIU C.: Dart: Dynamic animation and robotics toolkit. *Journal of Open Source Software 3*, 22 (2018), 500. 1, 3, 5

[LH17] LIU L., HODGINS J.: Learning to schedule control fragments for physics-based characters using deep q-learning. *ACM Transactions on Graphics (TOG) 36*, 3 (2017), 1–14. 2

[LPLL19] LEE S., PARK M., LEE K., LEE J.: Scalable muscle-actuated human simulation and control. *ACM Transactions on Graphics (TOG) 38*, 4 (2019), 1–13. 3

[LPY16] LIU L., PANNE M. V. D., YIN K.: Guided learning of control graphs for physics-based characters. *ACM Transactions on Graphics (TOG) 35*, 3 (2016), 1–14. 2

[LYWG13] LIU L., YIN K., WANG B., GUO B.: Simulation and control of skeleton-driven soft body characters. *ACM Transactions on Graphics (TOG) 32*, 6 (2013), 1–8. 2

[Mir96] MIRTICH B. V.: *Impulse-based dynamic simulation of rigid body systems*. University of California, Berkeley, 1996. 3

[NVI19] NVIDIA: PhysX 4.1, 2019. URL: https://developer.nvidia.com/physx-sdk. 5, 6

[PALvdP18] PENG X. B., ABBEEL P., LEVINE S., VAN DE PANNE M.: DeepMimic: Example-guided deep reinforcement learning of physics-based character skills. *ACM Transactions on Graphics (TOG) 37*, 4 (2018), 1–14. 1, 3, 6, 8

[PBVdP16] PENG X. B., BERSETH G., VAN DE PANNE M.: Terrain-adaptive locomotion skills using deep reinforcement learning. *ACM Transactions on Graphics (TOG) 35*, 4 (2016), 1–12. 3

[PBYVDP17] PENG X. B., BERSETH G., YIN K., VAN DE PANNE M.: DeepLoco: Dynamic locomotion skills using hierarchical deep reinforcement learning. *ACM Transactions on Graphics (TOG) 36*, 4 (2017), 1–13. 3

[PRL*19] PARK S., RYU H., LEE S., LEE S., LEE J.: Learning predict-and-simulate policies from unorganized human motion data. *ACM Trans. Graph. 38*, 6 (2019). 1, 3

[TLT11] TAN J., LIU K., TURK G.: Stable proportional-derivative controllers. *IEEE Computer Graphics and Applications 31*, 4 (2011), 34–44. 1, 2

[WO82] WALKER M. W., ORIN D. E.: Efficient dynamic computer simulation of robotic mechanisms. 7

[YK19] YUAN Y., KITANI K.: Ego-pose estimation and forecasting as real-time PD control. In *Proceedings of the IEEE International Conference on Computer Vision* (2019), pp. 10082–10092. 3

[YLvdP07] YIN K., LOKEN K., VAN DE PANNE M.: SIMBICON: Simple biped locomotion control. *ACM Transctions on Graphics 26*, 3 (2007), Article 105. 1

[YTL18] YU W., TURK G., LIU C. K.: Learning symmetric and low-energy locomotion. *ACM Transactions on Graphics (TOG) 37*, 4 (2018), 144. 1

[ZH02] ZORDAN V. B., HODGINS J. K.: Motion capture driven simulations that hit and react. In *SCA* (2002), pp. 89–96. 1

[ZSKS18] ZHANG H., STARKE S., KOMURA T., SAITO J.: Mode-adaptive neural networks for quadruped motion control. *ACM Transactions on Graphics (TOG) 37*, 4 (2018), 1–11. 6

Appendix A: MABA Full Algorithm

Here we list the complete set of equations for re-implementing MABA. MABA can be implemented by a small set of simple and clean modifications to the original ABA. More specifically, if we set $\boldsymbol{K}_i = \boldsymbol{0}$ in Equation 38, and $\boldsymbol{Q}_i = \boldsymbol{\tau}_i$ the input joint actuation torques in Equation 40, we get the original ABA.

Extra Notations

Following Featherstone's notation system, we use $\times$ and $\times^*$ to denote spatial cross products, and ${}^i\boldsymbol{X}_j$ and ${}^i\boldsymbol{X}_j^*$ to denote coordinate transformation matrices from frame j to frame i. ${}^i\boldsymbol{X}_j$ applies to spatial motion vectors, while ${}^i\boldsymbol{X}_j^*$ applies to spatial force vectors. All other variables have been defined in Section 3, except for $\boldsymbol{H}_i$ and $\boldsymbol{D}_i$ which are solely used for removing repeated expressions. We refer interested readers to [Fea14] for more detailed explanations.

Pass 1

$$\boldsymbol{v}_0 = \boldsymbol{0} \tag{30}$$

$$\boldsymbol{v}_i = {}^i\boldsymbol{X}_{\lambda(i)}\boldsymbol{v}_{\lambda(i)} + \boldsymbol{S}_i\dot{\boldsymbol{q}}_i \tag{31}$$

$$\boldsymbol{c}_i = \boldsymbol{v}_i \times \boldsymbol{S}_i \dot{\boldsymbol{q}}_i \tag{32}$$

$$\boldsymbol{p}_i = \boldsymbol{v}_i \times^* \boldsymbol{I}_i \boldsymbol{v}_i \tag{33}$$

Pass 2

$$\boldsymbol{I}_i^A = \boldsymbol{I}_i + \sum_{j \in \mu(i)} {}^i\boldsymbol{X}_j^* \boldsymbol{I}_j^a {}^j\boldsymbol{X}_i \tag{34}$$

$$\boldsymbol{p}_i^A = \boldsymbol{p}_i + \sum_{j \in \mu(i)} {}^i\boldsymbol{X}_j^* \boldsymbol{p}_j^a \tag{35}$$

$$\boldsymbol{H}_i = \boldsymbol{I}_i^A \boldsymbol{S}_i \tag{36}$$

$$\boldsymbol{D}_i = \boldsymbol{S}_i^T \boldsymbol{H}_i + \boldsymbol{K}_i \tag{37}$$

$$\boldsymbol{K}_i = \boldsymbol{K}_{di} \Delta t \tag{38}$$

$$\boldsymbol{u}_i = \boldsymbol{Q}_i - \boldsymbol{S}_i^T \boldsymbol{p}_i^A \tag{39}$$

$$\boldsymbol{Q}_i = -\boldsymbol{K}_{pi}(\boldsymbol{q}_i + \Delta t \dot{\boldsymbol{q}}_i - \bar{\boldsymbol{q}}_i) - \boldsymbol{K}_{di} \dot{\boldsymbol{q}}_i \tag{40}$$

$$\boldsymbol{I}_i^a = \boldsymbol{I}_i^A - \boldsymbol{H}_i \boldsymbol{D}_i^{-1} \boldsymbol{H}_i^T \tag{41}$$

$$\boldsymbol{p}_i^a = \boldsymbol{p}_i^A + \boldsymbol{I}_i^a \boldsymbol{c}_i + \boldsymbol{H}_i \boldsymbol{D}_i^{-1} \boldsymbol{u}_i \tag{42}$$

Pass 3

$$\boldsymbol{a}_0 = \boldsymbol{0} \tag{43}$$

$$\boldsymbol{a}_i' = {}^i\boldsymbol{X}_{\lambda(i)} \boldsymbol{a}_{\lambda(i)} + \boldsymbol{c}_i \tag{44}$$

$$\ddot{\boldsymbol{q}}_i = \boldsymbol{D}_i^{-1}(\boldsymbol{u}_i - \boldsymbol{H}_i^T \boldsymbol{a}_i') \tag{45}$$

$$\boldsymbol{a}_i = \boldsymbol{a}_i' + \boldsymbol{S}_i \ddot{\boldsymbol{q}}_i \tag{46}$$