GPU-Based Contact-Aware Trajectory Optimization Using A Smooth Force Model

ZHERONG PAN*, University of North Carolina at Chapel Hill, USA
BO REN, Nankai University, China
DINESH MANOCHA, University of Maryland at College Park, USA

We present a new formulation of trajectory optimization for articulated bodies. Our approach uses a fully differentiable dynamic model of the articulated body, and a smooth force model that approximates all kinds of internal/external forces as a smooth function of the articulated body's kinematic state. Our formulation is contact-aware and its complexity is not dependent on the contact positions or the number of contacts. Furthermore, we exploit the block-tridiagonal structure of the Hessian matrix and present a highly parallel Newton-type trajectory optimizer that maps well to GPU architectures. Moreover, we use a Markovian regularization term to overcome the local minima problems in the optimization formulation. We highlight the performance of our approach using a set of locomotion tasks performed by characters with 15 − 35 DOFs. In practice, our GPU-based algorithm running on a NVIDIA TITAN-X GPU provides more than 30× speedup over a multi-core CPU-based implementation running on Intel Xeon E5-1620 CPU. In addition, we demonstrate applications of our method on various applications such as contact-rich motion planning, receding-horizon control, and motion graph construction.

CCS Concepts: • **Computing methodologies** → **Physical simulation**.

Additional Key Words and Phrases: trajectory optimization, articulated bodies, deformable bodies, position-based dynamics

ACM Reference Format:
Zherong Pan, Bo Ren, and Dinesh Manocha. 2019. GPU-Based Contact-Aware Trajectory Optimization Using A Smooth Force Model. 1, 1 (May 2019), 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Generating physics-based character locomotion is an important problem in computer animations. Examples of these movements include walking, jumping, climbing, kicking, etc. Over time, researchers have developed various techniques to generate the locomotive gaits and animations that satisfy physical constraints governed by articulated body dynamics. One challenge in these formulations is the handling of non-smooth changes in contact positions and number of contacts. Some of previous methods such as [Liu et al. 2005] require users to explicitly specify the contact points. More general methods [Posa et al. 2014] have also been proposed that search for contact positions automatically, but the number of contacts must be specified by users. All these methods involve solving a high-dimensional trajectory optimization problem, which is an expensive computation in high-dimensional spaces.

*corresponding author

Authors' addresses: Zherong Pan, University of North Carolina at Chapel Hill, Sitterson Hall, Chapel Hill, NC, 27514, USA, zherong@cs.unc.edu; Bo Ren, Nankai University, 94 Weijin Rd, Tianjin, 300071, China, rb@nankai.edu.cn; Dinesh Manocha, University of Maryland at College Park, A.V. Williams Building, 8223 Paint Branch Drive, College Park, MD, 20742, USA, dm@cs.unc.edu.

2019. XXXX-XXXX/2019/5-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

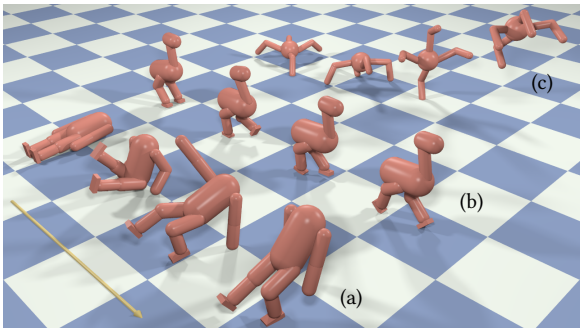


Fig. 1. Our method can generate trajectories with frequent changes in the contact points and the number of contacts such as a humanoid getting up (a), a 2-legged bird walking (b), and a 4-legged spider jumping (c). The yellow axis indicates the direction of time elapsing. The trajectories are found within a couple of minutes on NVIDIA TITAN-X GPU using trust region optimization and provide more than 30× speedup over a multi-core CPU-based implementation running on Intel Xeon E5-1620 CPU.

Previous formulations of trajectory optimization need to deal with two major issues. First, to handle the non-smooth changes in contact positions, auxiliary variables such as contact forces [Mordatch et al. 2013] or Lagrangian multipliers [Posa et al. 2014] are introduced, which increases the dimension of the search space. Second, previous methods induce a joint optimization problem with different sets of variables, such as character configurations, external forces, and control forces, making it difficult to design efficient numerical algorithms.

Main Results: We present a new formulation of trajectory optimization for articulated models that is invariant to both the contact positions and the number of contacts. Our key technique is to use a smooth force model to approximate all the internal/external forces as a smooth function of the articulated body's kinematic state. Our approach allows us to search for locally optimal contact positions and forces without introducing any auxiliary variables such as Lagrangian multipliers, leading to a smaller number of variables to be optimized. However, this smooth force model is a stiff force term leading to additional difficulties in time integration. To solve this problem, we extend position-based dynamics methods [Hahn et al. 2012; Müller et al. 2007; Pan and Manocha 2018] to re-derive the articulated body dynamics equation and solve it using a fully implicit time integration scheme. In addition, this formulation is fully differentiable whose derivatives can be computed analytically. As we use this formulation for trajectory optimization, the resulting Hessian has a simple, block-tridiagonal sparsity pattern and thereby enables parallel factorization. By putting all these components together, we

show that each and every step of a Newton-type trajectory optimizer can run in parallel on a GPU.

A well-known problem with smooth approximate internal/external forces is that it can introduce additional, sub-optimal local minima to the objective function. We use a Markovian regularization [Thodoroff et al. 2018] to avoid this problem, and show the resulting algorithm is parallel friendly and maps well to GPU architectures. Markovian regularization imposes the constraint that the next state of the articulated body should be determined from its last state via a recurrent neural-network (RNN) [Elman 1990]. We have evaluated the efficiency and generality of our formulation on several locomotion benchmarks as shown in Figure 1. In summary, our contributions include:

- A new formulation of trajectory optimization using a differentiable smooth force model, that is invariant to both the contact positions and the number of contacts.
- A GPU-parallel trajectory optimization using a Newton-type trust-region optimizer with analytic Hessian matrix computation and parallel factorization, that achieves more than 30× speedup over 4-core-CPU-based implementation on a desktop machine.
- A Markovian regularization based on recurrent neural networks that avoids sub-optimal local minima.

We highlight the performance of our algorithm on complex locomotion tasks as well as motion planning, receding-horizon control and motion graph construction algorithm. Our GPU-based algorithm takes a few minutes on $15 - 35$ DOF models on NVIDIA TITAN-X GPU. The rest of the paper is organized as follows. We briefly review prior work in character locomotion and trajectory optimization in Section 2. We give an overview of the problem of character locomotion and trajectory optimization in Section 3. Next, we formulate our dynamics model and smoothed force model in Section 4. The trajectory optimization problem is formulated in Section 5. Then, in Section 6 we present our parallel Newton-type trajectory optimization algorithm. Finally, we describe an effective regularization for optimization in Section 7. We highlight the performance on many complex benchmarks in Section 8.

2 RELATED WORK

We review previous work in character locomotion, trajectory optimization, position-based dynamics, and controller learning.

2.1 Character Locomotion

Character locomotion targets character animation generation with minimal user inputs. Methods for character locomotion include data-driven methods [Kang and Lee 2017; Kovar et al. 2002; Levine et al. 2012], which take motion capture data as inputs and output new animations by interpolating, generalizing, or re-organizing them. While data-driven methods are computationally efficient, they sacrifice physical correctness. For a specific task such as walking, animations can be generated very efficiently by designing special physics-based controllers [Hodgins et al. 1995; Wang et al. 2012; Yin et al. 2007]. Other methods [Liu et al. 2010] focus on generating physics-based animations for an arbitrary locomotion task. One such approach is trajectory optimization [Liu et al. 2005; Witkin and Kass 1988], which takes into account various constraints, such as physical correctness and energy efficiency. Our formulation is based on these trajectory optimization methods.

2.2 Trajectory Optimization

Trajectory optimization is an essential component in many character locomotion applications including motion control [Liu et al. 2005; Posa et al. 2014; Schulman et al. 2014; Tassa et al. 2012; Witkin and Kass 1988; Zucker et al. 2013] and reinforcement learning [Levine and Koltun 2013]. It has also been applied to other applications such as deformable body control [Barbič et al. 2009, 2012], fluid animation editing [Treuille et al. 2003], keyframe interpolation [Bai et al. 2016], and 3D reconstruction [Xu et al. 2014]. Trajectory optimization problem can be solved using derivative-free optimizers such as CMA-ES [Ha and Liu 2014; Lee et al. 2014] or particle belief propagation [Hämäläinen et al. 2015; Naderi et al. 2017]. Other methods are based on gradient-based optimization [Liu et al. 2005; Witkin and Kass 1988] and our approach follows these methods.

In applications with contact-rich locomotion, a challenging issue is to search for contact points and contact forces. To this end, early methods [Liu et al. 2005] rely on user inputs. The first automatic method [Wampler and Popović 2009] uses a two-stage stochastic method to search for contact phases and contact points. More recently, [Mordatch et al. 2013; Posa et al. 2014; Schulman et al. 2014] unify the search of contact points and contact phases by optimizing contact forces and positions with additional contact integrity cost functions or hard complementary constraints. Finally, [Tassa et al. 2012] uses an approximate force model which can be solved as a function of state and evaluated using finite differences.

Due to the parallel nature of trajectory optimization in the temporal domain, some previous techniques [Chretien et al. 2016; Heinrich et al. 2015; Park et al. 2013; Plancher and Kuindersma 2018; Wu et al. 2016] have used GPUs to accelerate the computation. Some methods [Chretien et al. 2016; Heinrich et al. 2015] use hybrid CPU-GPU schemes where GPU is used for derivative computation and CPU is responsible for trajectory optimization. Other methods [Wu et al. 2016] use sampling-based method and use GPUs to evaluate different samples in parallel.

2.3 Position-Based Dynamics

Position-based dynamics has recently become a prominent method for modeling various physical phenomena, including rigid bodies [Deul et al. 2014; Pan and Manocha 2018], elastic/plastic bodies [Bender et al. 2014], and fluid bodies [Macklin and Müller 2013]. Compared with previous velocity-based formulations, PBD can easily unify different physical models, and PBD exhibits superior stability, as compared with previous velocity-based methods such as [Stewart 2000]. A PBD simulator can take arbitrarily large timestep sizes by casting physical simulation as a numerical optimization problem [Bouaziz et al. 2014; Gast et al. 2015].

2.4 Controller Learning

It is common routine for animators to first optimize a controller and then use it to generate physically-based animations. A popular form of Learning-to-Control is model-free, sampling-based reinforcement

learning [Liu and Hodgins 2017; Peng et al. 2017; Won et al. 2017], which optimizes controllers to maximize a high-level reward function. After controller optimization, these controllers can be used to generate animations at real-time. Another form of Learning-to-Control is model-based reinforcement learning [Levine and Koltun 2013; Mordatch et al. 2015], which exploits trajectory optimization to accelerate the data-efficiency of Learning-to-Control. In these applications, thousands of trajectories are optimized to provide training data for controller learning, where trajectory optimization is the major computational bottleneck. Inspired by these works, we show that machine learning techniques can be used to avoid sub-optimal local minima in trajectory optimization by requiring the trajectory to be memoryless or Markovian.

3 PROBLEM STATEMENT

In this section, we formulate the problem of character animation. Next, we reduce the problem to trajectory optimization and discuss different approaches to represent a trajectory. Our formulation is also applicable to other dynamics models.

3.1 Character Animation

Given a character model, we denote its kinematic configuration as $\mathbf{x}$. For example, an articulated body consists of a set of rigid bodies and $\mathbf{x}$ consists of joint parameters and a global rigid transformation. In this paper, we consider articulated characters with $15-35$ DOFs. The goal of character animation is to generate a trajectory of N timesteps, $(\mathbf{x}_1, \cdots, \mathbf{x}_N)$, which represents an animation of the character performing a certain locomotion task. We assume the character is an articulated body that consists of a set of K rigid bodies $\{\mathcal{R}_1, \cdots, \mathcal{R}_K\}$, and $\mathcal{R} = \bigcup_{k=1}^{K} \mathcal{R}_k$ is the domain of entire articulated body. The kinematic configuration of $\mathcal{R}_k$ is described by a rotation $\mathbf{R}_k$ and a translation $\mathbf{t}_k$, which can be derived from $\mathbf{x}$ via forward kinematics [Murray et al. 1994, Chapter 3.2].

3.2 Trajectory Optimization

We compute a solution to the character animation problem using trajectory optimization. A key technical issue in trajectory optimization is the representation of a trajectory. Except for a series of kinematic configurations, the character can be in contact with the obstacles and under a set of external forces, $\mathbf{f}^E$. The character can also be controlled using a set of internal control forces, $\mathbf{u}$. As a result, a trajectory is typically represented using three sets of variables:

$$\mathcal{T}_\mathbf{x} = (\mathbf{x}_1, \cdots, \mathbf{x}_N) \quad \mathcal{T}_\mathbf{f} = (\mathbf{f}_1^E, \cdots, \mathbf{f}_N^E) \quad \mathcal{T}_\mathbf{u} = (\mathbf{u}_1, \cdots, \mathbf{u}_N),$$

where we use the subscript to denote a timestep index with the timestep size Δt. These three sets of variables can be optimized in different formulations. If we assume simplified controller such as linear feedback controller [Tassa et al. 2012] or PD controller [Yin et al. 2007], $\mathcal{T}_\mathbf{u}$ can be eliminated and expressed as a function of $\mathcal{T}_\mathbf{x}$, but handling $\mathcal{T}_\mathbf{f}$ is more challenging. A typical method [Mordatch et al. 2012] uses the following formulation:

$$\underset{\mathcal{T}_\mathbf{x}, \mathcal{T}_\mathbf{f}}{\mathbf{argmin}}\ \mathbf{E}_{\text{phys}}(\mathcal{T}_\mathbf{x}, \mathcal{T}_\mathbf{f}) + \mathbf{E}_{\text{int}}(\mathcal{T}_\mathbf{x}, \mathcal{T}_\mathbf{f}) + \mathbf{E}_{\text{obj}}(\mathcal{T}_\mathbf{x}) + \mathbf{E}_{\text{reg}}. \tag{1}$$

Among the objective function terms, $\mathbf{E}_{\text{phys}}$ measures how well $\mathcal{T}$ satisfies the laws of physics. $\mathbf{E}_{\text{obj}}$ is a high-level objective specified by users, such as moving to a specific point or exhibiting a specific pose. $\mathbf{E}_{\text{reg}}$ is a regularization function such as the magnitude of control forces. Finally, $\mathbf{E}_{\text{int}}$ is a soft constraint that ensures that $\mathbf{f}^E$ is valid. In other words, $\mathbf{E}_{\text{int}}$ ensures that $\mathbf{f}^E$ is non-zero only when a point is in contact. In another formulation [Posa et al. 2014], instead of using $\mathbf{E}_{\text{int}}$, $\mathbf{f}^E$ is ensured to be valid by using a set of hard constraints. The resulting optimization using these methods are joint optimizations over the space of $\mathcal{T}_\mathbf{x}, \mathcal{T}_\mathbf{f}$.

3.3 Other Dynamics Models

The formulation presented above for articulated models can also be extended to other dynamics models. For example, in Figure 2, we show our formulation working with two deformable characters, a beam and a ball. Our formulation is mainly used for low-DOF dynamics models, we use the reduced-order formulation [Hauser et al. 2003] to reduce the number of DOFs, $|\mathbf{x}|$, to 16 for both characters.

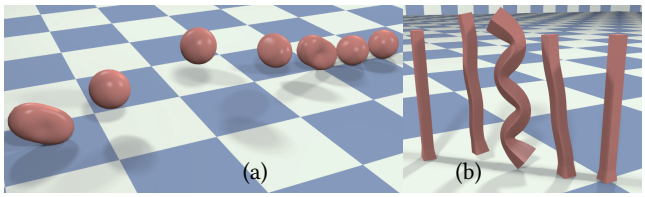


Fig. 2. A deformable ball (a) and a deformable beam (b) jumping to the left. In both cases, we use a reduced-order formulation, and the number of DOFs, $|\mathbf{x}|$, is 16 for both characters.

4 ARTICULATED BODY DYNAMICS

In this section, we present a new derivation of articulated body dynamics using smooth force models, which allows derivatives of our objective function to be analytically computed. We briefly introduce this formulation in Section 4.1. Next, in Section 4.2, we augment this formulation with our smooth internal/external force models to derive an invariant formulation in the number of contact points.

4.1 Forward Dynamics Function

In this section, we formulate our forward dynamics equation:

$$\mathbf{FD}_i \triangleq \mathbf{FD}(\mathbf{x}_{i+1}, \mathbf{x}_i, \mathbf{x}_{i-1}, \mathbf{u}_i) = 0, \tag{2}$$

where we represent the current dynamic state of the articulated body with $(\mathbf{x}_i, \mathbf{x}_{i-1})$. $\mathbf{FD}$ takes $(\mathbf{x}_i, \mathbf{x}_{i-1})$ and the control signal $\mathbf{u}_i$ as inputs and outputs the predicted state at the next timestep, $\mathbf{x}_{i+1}$. Note that we assume the use of an implicit time integrator in Equation 2. This is because our smooth force models are stiff and require an implicit time integrator for stability.

We first derive $\mathbf{FD}_i$ without considering the control signal $\mathbf{u}_i$, to do this we need to approximate velocities. Considering a continuous point $\mathbf{p} \in \mathcal{R}_k$ in the local frame of reference, its current position is given as $\mathbf{P}(\mathbf{x}_i) = \mathbf{R}_k(\mathbf{x}_i)\mathbf{p} + \mathbf{t}_k(\mathbf{x}_i)$ and its velocity $\dot{\mathbf{P}}$ at timestep i in the global frame of reference can be approximated using finite differences in the configuration space via the chain rule:

$$\dot{\mathbf{P}}_{conf}(\mathbf{x}_i, \mathbf{x}_{i-1}) = \frac{\partial \mathbf{P}(\mathbf{x}_i)}{\partial \mathbf{x}_i} \left[\mathbf{x}_i - \mathbf{x}_{i-1}\right] / \Delta t, \tag{3}$$

which is used in previous methods such as [Stewart 2000]. A key innovation in [Hahn et al. 2012; Müller et al. 2007; Pan and Manocha 2018] is that velocities are discretized in the Euclidean space instead

of the configuration space:

$$\dot{\mathbf{P}}(\mathbf{x}_i, \mathbf{x}_{i-1}) = [\mathbf{P}(\mathbf{x}_i) - \mathbf{P}(\mathbf{x}_{i-1})]/\Delta t, \quad (4)$$

which reduces the order of all derivatives by one and allows the derivatives to be computed analytically using an adjoint algorithm. By taking an integral of $\mathbf{p}$ over $\mathcal{R}$, we get the following forward dynamics function:

$$\mathbf{FD}_i \triangleq \int_{\mathcal{R}} \frac{\partial \mathbf{P}}{\partial \mathbf{x}_i}^T \left[\rho \frac{\dot{\mathbf{P}}(\mathbf{x}_i, \mathbf{x}_{i-1}) - \dot{\mathbf{P}}(\mathbf{x}_{i-1}, \mathbf{x}_{i-2})}{\Delta t}\right] d\mathbf{p} - \mathbf{f}^I - \mathbf{f}^E, \quad (5)$$

where ρ is the mass density of $\mathcal{R}$. $\mathbf{f}^I$ and $\mathbf{f}^E$ are the total internal and external forces, respectively, both defined in the configuration space. Finally, the multiplication by $\partial \mathbf{P}_i/\partial \mathbf{x}_i$ from the left transforms the equation back to the configuration space. The main advantage Equation 4 has over Equation 3 is that the order of derivatives is reduced by one, so that $\partial \mathbf{FD}/\partial \mathbf{x}_i$, $\partial \mathbf{FD}/\partial \mathbf{x}_{i-1}$ and $\partial \mathbf{FD}/\partial \mathbf{x}_{i-2}$ can be analytically computed, which is useful in our GPU-based trajectory optimizer (Section 6). Moreover, Equation 4 retains the ability to model the dynamics of articulated bodies under minimal coordinates, i.e., in terms of joint parameters.

4.2 Smoothed Internal/External Force Models

In this section, we represent $\mathbf{f}^E$ and $\mathbf{f}^I$ as functions of $\mathbf{x}$. In addition, we show that $\mathbf{f}^E$ and $\mathbf{f}^I$ are C^1-continuous, which is essential for a Newton-type optimizer to converge and for $\mathbf{FD}$ to be differentiable across the surface of obstacles.

Our characters are under two kinds of internal forces: joint limits and self-collisions. Assuming $x \in \mathbf{x}$ has joint limit $[l, u]$, we introduce the following potential energy:

$$\mathcal{P}_{\text{joint}}(\mathbf{x}) = w_{\text{joint}} \sum\nolimits_{x \in \mathbf{x}} \left[\mathbf{min}(x - l, 0)^3 + \mathbf{min}(u - x, 0)^3\right],$$

where w_{joint} is the penalty coefficient. We model self-collisions by approximating the character with spheres and penalizing the interpenetration between any spheres. The corresponding potential energy is:

$$\mathcal{P}_{\text{self}}(\mathbf{x}) = w_{\text{self}} \sum\nolimits_{\mathbf{S}_{a,b}} [\mathbf{min}(\|\mathbf{S}_a(\mathbf{x}) - \mathbf{S}_b(\mathbf{x})\| - \text{rad}_a - \text{rad}_b, 0)]^3,$$

where $\mathbf{S}_a$ is the center of a sphere with radius rad_a and w_{self} is the penalty coefficient. In summary, the internal force in Equation 5 takes the following form:

$$\mathbf{f}^I \triangleq -\partial\big(\mathcal{P}_{\text{joint}}(\mathbf{x}_i) + \mathcal{P}_{\text{self}}(\mathbf{x}_i)\big)/\partial \mathbf{x}_i.$$

Note that, although $\mathbf{min}$ is only C^0, we can use a cubic polynomial to ensure that $\mathcal{P}_{\text{joint}}, \mathcal{P}_{\text{self}}$ are C^2 and $\mathbf{f}^I(\mathbf{x}_i)$ is C^1.

We only approximate frictional contact forces because other external forces such as gravitational forces are already differentiable. As illustrated in Figure 3, we formulate $\mathbf{f}^E$ as a function of $\mathbf{x}_i$ and $\mathbf{x}_{i-1}$ by using the penetration depth $\mathbf{d}(\mathbf{P}(\mathbf{x}_i))$, or $\mathbf{d}$ for short. $\mathbf{d}$ is the distance from $\mathbf{x}_i$ to the closest point on the obstacle surface if $\mathbf{x}_i$ is inside the obstacle and zero otherwise. Contact forces consist of normal forces and tangential forces. Previous work [Gast et al. 2015] assumes that normal forces are proportional to $\mathbf{d}$. Instead, we assume that they are proportional to $\mathbf{d}^2$ so that $\mathbf{f}^E$ is C^1. For the frictional force, we notice that it takes effect by minimizing the tangential velocity and that it is only non-zero at points in collision. We model these two properties qualitatively by making frictional forces proportional to both $\mathbf{d}^2$ and $\dot{\mathbf{P}}$. Combining these two force terms, $\mathbf{f}^E$ takes the following form:

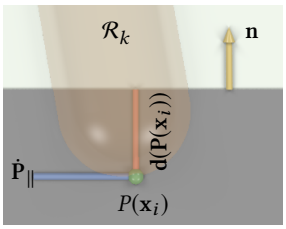


Fig. 3. We approximate the frictional forces on a point $\mathbf{p}$ (green) as a function of penetration depth $\mathbf{d}(\mathbf{P}(\mathbf{x}_i))$ (red) and the tangential component of $\dot{\mathbf{P}}$ (blue), where the gray area is the solid obstacle. The yellow axis is the outward normal.

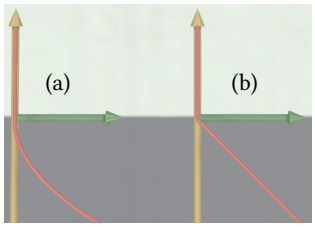


Fig. 4. We plot the change of $\mathbf{d}^2$ (a) and $\mathbf{d}$ (b) as the red curve. Obviously, $\mathbf{d}^2$ is C^1. The yellow axis indicates the signed distance to the obstacle (gray) and the green axis indicates the direction of force increase.

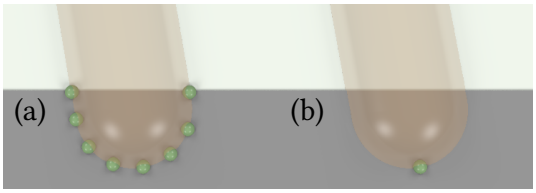


Fig. 5. If a character is represented as a mesh (a), we sum up Equation 6 for all the points in collision (green). If a character is represented using spheres/capsules (b), then each sphere or capsule contributes one force term on the point corresponding to the deepest penetration (green).

$$\mathbf{f}^E \triangleq \frac{\partial \mathbf{P}(\mathbf{x}_i)}{\partial \mathbf{x}_i}^T \left[w_\perp \mathbf{n} - w_{\parallel}(\mathbf{I} - \mathbf{n}\mathbf{n}^T)\dot{\mathbf{P}}(\mathbf{x}_i, \mathbf{x}_{i-1})\right] \mathbf{d}(\mathbf{P}(\mathbf{x}_i))^2, \quad (6)$$

where $\mathbf{n}$ is the outward normal direction and $w_\perp, w_{\parallel}$ are the magnitude coefficient for the normal and frictional forces, respectively. Finally, the multiplication by $\partial \mathbf{P}(\mathbf{x}_i)/\partial \mathbf{x}_i$ from the left transforms the equation back to the configuration space. Equation 6 is not as accurate as the conventional dry frictional model because it does not account for switching between static/sliding friction modes. Instead, by setting $w_{\parallel}$ to be a large constant, we can only approach static friction asymptotically. Note that, although $\mathbf{d}$ is a C^0 function of $\mathbf{P}(\mathbf{x}_i)$ as illustrated in Figure 4, $\mathbf{f}^E(\mathbf{x}_i, \mathbf{x}_{i-1})$ is C^1 because of the squared $\mathbf{d}$. Overall, we have the following lemma:

Lemma: *With the smooth force model Equation 6, the forward dynamics Equation 5 is a C^1-continuous function of $\mathbf{x}_i$, $\mathbf{x}_{i-1}$, and $\mathbf{x}_{i-2}$, whose derivatives can be computed analytically using a three-stage adjoint method described in Appendix A.*

This lemma ensures the global convergence of a Newton-type trust region optimizer. We refer readers to [Izmailov et al. 2011] for a proof.

As illustrated in Figure 5, for modeling characters whose limbs are represented as a general mesh, we define $\mathbf{f}^E$ by summing up Equation 6 from all the mesh vertices in collision. For limbs using spheres/capsules, each sphere/capsule contributes one force term on the point with the deepest penetration. The vertices or spheres/capsules in collision are detected using a bounding volume hierarchy [Lauterbach et al. 2009] constructed on a GPU.

5 TRAJECTORY OPTIMIZATION

In this section, we formulate our trajectory optimization problem. Our formulation has two important features that make it suitable for GPU parallelization. First, the formulation uses a small, fixed number of variables to represent a trajectory independent of the number of contact points. As a result, the Hessian matrices of both the objective function and the constraints have a fixed sparsity pattern. Second, all the derivatives required by the optimizer can be evaluated analytically. In addition, we ensure physics correctness via hard constraints.

In Section 4.1, we derived our uncontrolled forward dynamics function. Control forces can now be added to the right hand side, giving $\mathbf{FD}_i = \mathbf{U}^T\mathbf{u}_i$ where $\mathbf{U}$ is an $|\mathbf{u}| \times |\mathbf{x}|$-identity matrix:

$$\mathbf{U} = \left(\mathbf{I}^{|\mathbf{u}|}\ \mathbf{0}^{|\mathbf{u}|\times(|\mathbf{x}|-|\mathbf{u}|)}\right),$$

which projects $\mathbf{x}$ to the controllable degrees of freedom. If all the joints are controllable, $|\mathbf{u}| = (|\mathbf{x}| - 6)$, leaving the 6-dimensional global rigid transformation unactuated. Since $\mathbf{u}_i$ is linear in the forward dynamics equation, we can solve for $\mathbf{u}_i = \mathbf{U} \cdot \mathbf{FD}_i$ and express $\mathcal{T}_\mathbf{u}$ as a function of $\mathcal{T}_\mathbf{x}$. After eliminating $\mathcal{T}_\mathbf{u}$, we get the following reduced trajectory optimization problem:

$$\underset{\mathcal{T}_\mathbf{x}}{\textbf{argmin}}\ \mathbf{E}(\mathcal{T}_\mathbf{x}) \triangleq \mathbf{E}_{\text{obj}}(\mathcal{T}_\mathbf{x}) + \mathbf{E}_{\text{reg}}(\mathcal{T}_\mathbf{u}(\mathcal{T}_\mathbf{x})) \tag{7}$$

$$\textbf{s.t.}\quad \mathbf{C}(\mathcal{T}_\mathbf{x}) \triangleq (\mathbf{I}^N \otimes \mathbf{U}_/)\left(\mathbf{FD}_1 \cdots \mathbf{FD}_N\right)^T = 0,$$

where $\mathbf{U}_/$ is a $(|\mathbf{x}|-|\mathbf{u}|)\times|\mathbf{x}|$ matrix and is the orthogonal complement of $\mathbf{U}$ such that $\left(\mathbf{U}^T\ \mathbf{U}_/{}^T\right)^T = \mathbf{I}^{|\mathbf{x}|}$. $\mathbf{U}_/$ projects $\mathbf{FD}_i$ to the unactuated degrees of freedom, for which control forces are zero. Note that the number of hard constraints and objective function terms do not change with the number of contact points in Equation 7. Compared with Equation 1, our formulation does not require $\mathbf{E}_{\text{int}}$ because $\mathbf{f}^I, \mathbf{f}^E$ are functions of $\mathbf{x}$, and we ensure the governing equation $\mathbf{FD}_i = \mathbf{U}^T\mathbf{u}_i$ is satisfied using hard constraints. Our formulation solves for $N(|\mathbf{x}|)$ variables under $N(|\mathbf{x}| - |\mathbf{u}|)$ hard constraints.

5.1 Spline Interpolation

In this section, we present a spline interpolation scheme that is compatible with hard constraints. Spline interpolation is not a necessary component of trajectory optimization but it can ensure the smoothness of the animation, further reduce the decision variables to be optimized, and our scheme ensures the feasibility of the resulting optimization problem.

We use a Hermite spline interpolation scheme and denote the interpolated trajectory as:

$$\mathcal{T}_\mathbf{c} = (\mathbf{c}_1, \cdots, \mathbf{c}_Z),$$

where we have Z control points and each control point, $\mathbf{c}_i = \left(\mathbf{x}_i\ \dot{\mathbf{x}}_i\right)$, has $2|\mathbf{x}|$ parameters involving $\mathbf{x}$ and its tangent (time derivative). The interpolation scheme can be expressed as an interpolation stencil matrix $\mathbf{S}^{N\times 2Z}$ and the relationship $\mathcal{T}_\mathbf{x} = (\mathbf{S} \otimes \mathbf{I}^{|\mathbf{x}|})\mathcal{T}_\mathbf{c}$. Here we use the same stencil for all the DOFs of an articulated body. This relationship is sufficient for an unconstrained trajectory optimization such as [Byravan et al. 2014], by plugging the spline interpolated $\mathcal{T}_\mathbf{x}$ into $\mathbf{E}$. However, spline interpolation might result in incompatible hard constraints because the number of variables to be optimized decreases while the number of constraints do not change. This issue has been previously noticed by [Winkler et al. 2018], and we resolve it using a technique called constraint reduction [Wicke et al. 2009], which was previously used for reduced-order fluid modeling. The idea is that we interpolate the hard constraints along with the variables, using stencil matrix $\mathbf{S}$. As a result, the new hard constraints become:

$$\mathbf{C}(\mathcal{T}_\mathbf{c}) \triangleq (\mathbf{S}^T \otimes \mathbf{U}_/)\left(\mathbf{FD}_1 \cdots \mathbf{FD}_N\right)^T = 0.$$

This technique is a special form of the projection-based model reduction [Benner et al. 2015], where the spline interpolation stencil $\mathbf{S}$ is used as both the "test" basis and the "trial" basis. In summary, our spline-interpolated formulation solves for $2Z|\mathbf{x}|$ variables under $2Z(|\mathbf{x}| - |\mathbf{u}|)$ hard constraints. Our formulation after spline interpolation is:

$$\underset{\mathcal{T}_\mathbf{c}}{\textbf{argmin}}\ \mathbf{E}(\mathcal{T}_\mathbf{c}) \quad \textbf{s.t.} \quad \mathbf{C}(\mathcal{T}_\mathbf{c}) = 0. \tag{8}$$

Equation 8 is a more compact formulation of the trajectory optimization problem than previous method. The main computational benefit of this formulation is GPU-friendliness.

6 GPU-BASED TRAJECTORY OPTIMIZATION

In this section, we solve Equation 8 using a Newton-type trust-region algorithm [Nocedal and Wright 2006, Chapter 18.5] (Section 6.1). We also show that each and every step of this optimizer (Equation 8) can run in parallel on GPUs.

6.1 Newton-Type Trust-Region Algorithm

We handle non-linear hard constraints to ensure physics correctness and use an approximate Hessian to accelerate the convergence rate. The algorithm is outlined in Algorithm 1 with three main steps. First, we linearize the constrained system as follows (Line 3):

$$\bar{\mathbf{E}}(\Delta\mathcal{T}_\mathbf{c}) = \frac{1}{2}\Delta\mathcal{T}_\mathbf{c}{}^T\mathbf{H}\Delta\mathcal{T}_\mathbf{c} + \Delta\mathcal{T}_\mathbf{c}{}^T\mathbf{b} \tag{9}$$

$$\bar{\mathbf{C}}(\Delta\mathcal{T}_\mathbf{c}) = \mathbf{A}\Delta\mathcal{T}_\mathbf{c} + \mathbf{C}_0,$$

where $\mathbf{H}, \mathbf{b}$ are the Hessian and gradient of $\mathbf{E}$ and $\mathbf{A}, \mathbf{C}_0$ are the Jacobian and value of $\mathbf{C}$, respectively. Since all the terms in $\mathbf{E}$ are sums of squares, we use $\mathbf{J}^T\mathbf{J}$-approximation to compute $\mathbf{H}$. Note that, although $\mathbf{J}^T\mathbf{J}$-approximate Hessian only contains first order derivatives information, taking it into consideration leads to higher rate of convergence when the Hessian has a varying curvature [Ranganathan 2004]. Next, we solve the trust-region subproblem (Line 4):

$$\underset{\bar{\mathcal{T}}_\mathbf{c}}{\textbf{argmin}}\ \bar{\mathbf{E}}(\Delta\mathcal{T}_\mathbf{c}) \quad \textbf{s.t.}\ \bar{\mathbf{C}}(\Delta\mathcal{T}_\mathbf{c}) = 0, \|\Delta\mathcal{T}_\mathbf{c}\| < \delta r. \tag{10}$$

Equation 10 is a simplified version of the original subproblem in [Nocedal and Wright 2006, Chapter 18.5] because we use the Hessian of $\bar{\mathbf{E}}$ instead of the Hessian of the Lagrangian. In other words, we have discarded the Hessian of $\mathbf{C}$ because $\mathbf{C}$ is only C^1-continuous and its Hessian may not exist. We solve Equation 10 using the two-subspace minimization algorithm [Nocedal and Wright 2006, Chapter 4]. Then, if Equation 10 is infeasible, we relax the problem by adjusting $\mathbf{C}_0$ according to the method described in [Nocedal and Wright 2006, Chapter 18.5] (Line 5). Finally, we determine whether $\Delta\mathcal{T}_\mathbf{c}$ improves the solution according to the decrease in the value of

Algorithm 1 Newton-Type Trust-Region Algorithm

Input: Initial guess $\mathcal{T}_\mathbf{c}$ and parameters $\mathbf{C}_{\max}$, $\epsilon_\mathbf{C}$, and $\epsilon_\mathbf{E}$

1: Initialize $\delta r \leftarrow \infty$, converged $\leftarrow$ False
2: **while** converged = False **do**
3: Compute $\bar{\mathbf{E}}$ and $\bar{\mathbf{C}}$ using Equation 9
4: Solve subproblem Equation 10
5: Adjust $\mathbf{C}_0$ if Equation 10 is incompatible
6: Update μ according to Equation 11
7: ▷ Initialize trust-region radius
8: **if** $\delta r = \infty$ **then**
9: $\delta r \leftarrow \|\Delta\mathcal{T}_\mathbf{c}\|$
10: **end if**
11: ▷ Check whether $\Delta\mathcal{T}_\mathbf{c}$ can be accepted
12: **if** $\begin{cases}\Phi(\mathcal{T}_\mathbf{c}+\Delta\mathcal{T}_\mathbf{c},\mu) < \Phi(\mathcal{T}_\mathbf{c},\mu)\\ \|\mathbf{C}(\mathcal{T}_\mathbf{c}+\Delta\mathcal{T}_\mathbf{c})\|_\infty < \mathbf{C}_{\max}\end{cases}$ **then**
13: $\mathcal{T}_\mathbf{c} \leftarrow \mathcal{T}_\mathbf{c} + \Delta\mathcal{T}_\mathbf{c}$
14: $\delta r \leftarrow 1.2\delta r$
15: **else**
16: $\delta r \leftarrow 0.8\|\Delta\mathcal{T}_\mathbf{c}\|$
17: **end if**
18: ▷ Termination condition
19: **if** $\begin{cases}\|\Delta\mathcal{T}_\mathbf{c}\|_\infty < \epsilon_\mathbf{E}\\ \|\mathbf{C}(\mathcal{T}_\mathbf{c})\|_\infty < \epsilon_\mathbf{C}\end{cases}$ **then**
20: converged $\leftarrow$ True
21: **end if**
22: **end while**

the following merit function:

$$\Phi(\mathcal{T}_\mathbf{c}, \mu) = \frac{\mu}{2}\|\mathbf{C}(\mathcal{T}_\mathbf{c})\|^2 + \mathbf{E}(\mathcal{T}_\mathbf{c}),$$

where μ is monotonically increasing during each iteration, according to [Nocedal and Wright 2006, Chapter 18.5], to ensure that

$$\mu \geq \frac{\frac{1}{2}\Delta\mathcal{T}_\mathbf{c}^T\mathbf{H}\Delta\mathcal{T}_\mathbf{c} + \Delta\mathcal{T}_\mathbf{c}^T\mathbf{b}}{0.5\|\mathbf{C}_0\|^2}. \tag{11}$$

In addition, we introduce a new criterion to achieve faster convergence speed. We accept $\Delta\mathcal{T}_\mathbf{c}$ only when the maximal constraint violation is less than $\mathbf{C}_{\max}$. This new condition can effectively limit the value of μ and accelerate the reduction of $\mathbf{E}$ during the first few iterations. However, this new condition also requires that the constraint violation in the initial guess is less than $\mathbf{C}_{\max}$, which can be achieved by using an initial guess computed from quasistatic physical simulation. Specifically, we search for a static initial pose of the character and initialize every $\mathbf{x}_i$ to this pose. In practice, we observed that this strategy accelerates convergence. An optimization starting from a non-static pose will spend many early iterations on reducing constraint violations.

6.2 Parallel Algorithm

In this section, we show that Algorithm 1 can be parallelized entirely on a GPU. Three steps in our algorithm require special modifications to run in parallel. Other parts of Algorithm 1 only require basic vector math operations that can be performed in parallel using existing algorithms in [Bell and Hoberock 2011].

The first step is the computation of $\bar{\mathbf{E}}$ and $\bar{\mathbf{C}}$ (Line 3), for which our parallel algorithm is based on an analysis of the sparse matrices such as $\mathbf{H}$ and $\mathbf{A}$. These matrices have special sparsity patterns allowing us to design compact storage format and efficient GPU algorithms for matrix-vector/matrix-matrix multiplication and matrix factorization. These matrices are either block-low-triangular matrices where each row has at most K blocks and each block has $|\mathbf{x}|$ columns, denoted as $\mathbf{BLT}(K)$, or symmetric-block-tridiagonal matrices with block size B, denoted as $\mathbf{SBT}(B)$. Their compact storage formats are illustrated in Appendix B and matrix-matrix multiplications are performed blockwise using the high performance routines in [Nvidia 2008]. In Appendix C, we summarize the exact sparsity pattern of each sparse matrix and the steps to compute the matrices and vectors required by Algorithm 1 in parallel.

The second step that requires parallelization is the adjustment of $\mathbf{C}_0$ to resolve incompatible constraints (Line 4). When adjusting $\mathbf{C}_0$, we need to find $\Delta\mathcal{T}_\mathbf{c}$ that solves the normal equation:

$$\underset{\Delta\mathcal{T}_\mathbf{c}}{\mathbf{argmin}}\ \|\mathbf{A}\Delta\mathcal{T}_\mathbf{c} + \mathbf{C}_0\|^2,$$

which requires solving a sparse linear system whose left-hand-side is $\mathbf{A}^T\mathbf{A}$. This matrix does not have full-rank, so we solve it using the GPU-based conjugate gradient method [Bolz et al. 2003]. A GPU-based conjugate gradient is more costly than direct factorization, but this solution is only required in a few cases where there are incompatible constraints, which usually happens less then 10 times per optimization.

Finally, when solving subproblem Equation 10, we need to factorize and solve the following KKT-system:

$$\begin{pmatrix}\mathbf{H} & \mathbf{A}^T\\ \mathbf{A} & \end{pmatrix}\begin{pmatrix}\Delta\mathcal{T}_\mathbf{c}\\ \boldsymbol{\lambda}\end{pmatrix} = -\begin{pmatrix}\mathbf{b}\\ \mathbf{C}_0\end{pmatrix}. \tag{12}$$

The left-hand-side matrix of Equation 12 has a new sparsity pattern which is neither **BLT** nor **SBT**. However, after a permutation illustrated in Appendix B that interleaves blocks of $\mathbf{A}$ and blocks of $\mathbf{H}$, we can transform the matrix into one with a sparsity pattern $\mathbf{SBT}(4(2|\mathbf{x}| - |\mathbf{u}|))$. Fortunately, any matrix whose sparsity pattern matches $\mathbf{SBT}(B)$ can be solved in parallel using a block cyclic reduction (BCR) algorithm [Gander and Golub 1997]. As illustrated in Appendix B, the main idea of the BCR algorithm is to recursively permute the rows and columns of an $\mathbf{SBT}(B)$ matrix and divide the linear system into two smaller, independent subsystems. As a result, an $\mathbf{SBT}(B)$ matrix with D diagonal blocks can be factorized using $\mathbf{log}_2(D)$ passes of permutation and division. This leads to a serial computational overhead of $O(B^3 D\mathbf{log}_2(D))$, where B^3 is the cost of factorizing a $B \times B$ dense block. When we have BD processors, the BCR algorithm has a parallel computational overhead of $O(B^2\mathbf{log}_2(D))$, where B^2 is the cost of factorizing the $B \times B$ dense block in parallel. The BCR algorithm boils down to a list of independent $B \times B$ dense matrix factorization and multiplication operations, and we use the high-performance routines in [Nvidia 2008] to perform these operations.

7 MARKOVIAN REGULARIZATION

In this section, we propose a regularization term to make our formulation more robust in terms of avoiding sub-optimal local minima. Our formulation is inspired by methods that try to use learnable models to represent a locomotion trajectory. These models include Gaussian process [Levine et al. 2012], motion graphs [Kovar et al.

Algorithm 2 Trajectory Optimization+Markovian Regularization

1: **while** Not Converged **do**
2: ▷ Run GPU-based Algorithm 1
3: Fix $w_{\mathbf{RNN}}$, optimize $\mathcal{T}_\mathbf{c}$ to solve Equation 15
4: ▷ Run GPU-based LBFGS algorithm
5: Fix $\mathcal{T}_\mathbf{c}$, optimize $w_{\mathbf{RNN}}$ to minimize Equation 13 or Equation 16
6: **end while**

2002], or neural networks [Zhang et al. 2018]. All these methods assume that an animation trajectory should be encoded by a learning model. Similarly, we regularize our trajectory by requiring it to be encoded by a recurrent neural network (RNN):

$$\mathbf{c}_{i+1} = \mathbf{RNN}(\mathbf{c}_i, w_{\mathbf{RNN}}), \tag{13}$$

that brings the control point from its current timestep to the next timestep with $w_{\mathbf{RNN}}$ being its optimizable weights. However, we formulation this term as a regularization term in our optimization function. We use a fully-connected RNN with 3 hidden layers, each with 128 neurons and SmoothReLU activation functions [Dugas et al. 2000], as illustrated in Figure 6. Since the dependency of $\mathcal{T}_\mathbf{c}$ on $w_{\mathbf{RNN}}$ is differentiable, our trajectory optimization can be used to jointly optimize $w_{\mathbf{RNN}}$ and $\mathcal{T}_\mathbf{c}$. Using the following soft constraints:

$$\mathbf{E}_{\mathrm{RNN}} \triangleq \sum_{i=1}^{Z-1} \|\mathbf{c}_{i+1} - \mathbf{RNN}(\mathbf{c}_i, w_{\mathbf{RNN}})\|^2, \tag{14}$$

which is denoted as Markovian regularization because it requires the control points to be memoryless or Markovian. Intuitively, the **RNN** is trained to encode a locomotion trajectory. After training, the trajectory can be recovered from the **RNN** by unrolling it from the initial frame $\mathbf{c}_1$. Since our trajectory optimizer only requires trivial initialization from a static pose, we initialize the RNN to match this assumption. Specifically, we initialize all weights to a very small random value and then solve the joint optimization problem:

$$\underset{\mathcal{T}_\mathbf{c}, w_{\mathbf{RNN}}}{\mathbf{argmin}}\ \mathbf{E}(\mathcal{T}_\mathbf{c}) + \mathbf{E}_{\mathrm{RNN}}(\mathcal{T}_\mathbf{c}, w_{\mathbf{RNN}}) \quad \mathbf{s.t.} \quad \mathbf{C}(\mathcal{T}_\mathbf{c}) = 0. \tag{15}$$

However, if we solve Equation 15 jointly using TRSQP, the Hessian matrix is not in **SBT** so that we cannot factorize it on GPU. To exploit parallel computation, we solve the Equation 15 alternatives by first optimizing $\mathcal{T}_\mathbf{c}$ using Algorithm 1 for a fixed number of iterations and then optimizing $w_{\mathbf{RNN}}$ using the LBFGS algorithm. The alternative algorithm is guaranteed to converge because the two substeps reduce Equation 15 monotonically. Both of these two steps can run in parallel due to the following lemma:

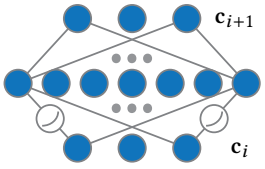


Fig. 6. A RNN encoding a spline-interpolated animation trajectory.

Lemma: *When solving Equation 15 using Equation 14, the sparsity pattern of the left-hand-side in Equation 12 does not change and is* $\mathbf{SBT}(4(2|\mathbf{x}| - |\mathbf{u}|))$.

In our experiments, Equation 15 performs reasonably well in avoiding sub-optimal local minima. However, the RNN optimized using this formulation is not stable. with Equation 14, the RNN can make a small error in every prediction denoted as $\|\mathbf{c}_{i+1} - \mathbf{RNN}(\mathbf{c}_i, w_{\mathbf{RNN}})\| = \epsilon$. The error, ϵ, can quickly accumulate in long trajectories. We can solve this problem using an asymmetric optimization scheme. When optimizing $\mathcal{T}_\mathbf{c}$, we use Equation 14, but when optimizing $w_{\mathbf{RNN}}$, we use a novel recurrent loss function:

$$\mathbf{E}^{\mathrm{recur}}_{\mathrm{RNN}} \triangleq \sum_{i=1}^{Z-1} \|\mathbf{c}_{i+1} - \mathbf{RNN}^i(\mathbf{c}_1, w_{\mathbf{RNN}})\|^2, \tag{16}$$

which differs from Equation 14 because it uses the recurrent relation to always infer $\mathbf{c}_{i+1}$ from the initial state $\mathbf{c}_1$. Here the superscript i means unrolling **RNN** i times from the first timestep. This approach is similar to the Bregman-ADMM method [Wang and Banerjee 2014], which uses an asymmetric loss function to accelerate convergence of conventional ADMM algorithms. In our experiments, using Equation 16 results in much smaller discrepancy between the trajectory and the RNN predictions and long-term stability of the RNN. However, a problem with Equation 16 is that it disallows fine-grained parallelism because computations for different timesteps, i, are not independent. Our final algorithm of trajectory optimization with Markovian regularization is outlined in Algorithm 2. When we require the trajectory to be representable using a RNN, we use Equation 16. Otherwise, we use Equation 14 for better parallelism and we always use Equation 14 in Section 8 unless otherwise noted.

8 EXPERIMENTS AND ANALYSIS

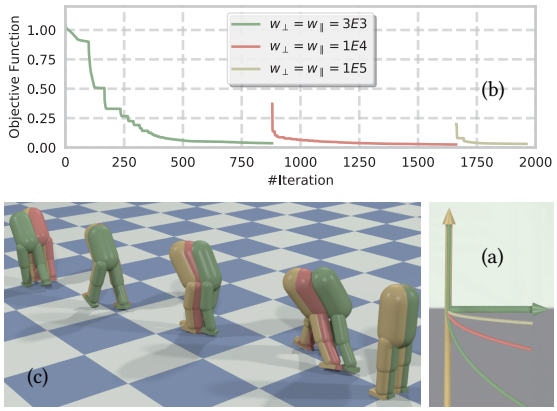


Fig. 7. We use three passes of optimization with $w_\perp = w_\parallel = 3E3, 1E4, 10E4$. The plot of the contact forces is given in (a). The convergence history of the optimizer is shown in (b), where the speed of convergence is much higher in the second and third passes. In (c), we show several frames of a bipedal walking animation after each pass. The difference between the second and third pass is very small, indicating converging behaviors.

In this section, we analyze the performance of our formulation in a set of benchmarks. We implement our algorithm using the Cuda programming interface [Nvidia 2008] and test our algorithm on a single desktop machine with one TITAN-X graphic card (3584 Cuda cores, 1.5GHz) and one Intel Xeon E5-1620 CPU (4 cores, 3.5GHz). When comparing the performance of CPU-based and GPU-based implementations, we use OpenMP to accelerate our CPU-based implementation. Due to the implicit time integration scheme, we allow the use of less timesteps via larger timestep sizes such as $\Delta t = 0.025 - 0.05$s in all experiments. When running the inner loop Algorithm 1, we set $\epsilon_\mathbf{E} = 10^{-3}$, $\epsilon_\mathbf{C} = 10^{-5}$, and $\mathbf{C}_{\max} = 0.1$. When running the outer loop, Algorithm 2, we first update $\mathcal{T}_\mathbf{c}$ using 200 iterations of Algorithm 1 and then update $w_{\mathbf{RNN}}$ using 1000

iterations of the LBFGS algorithm. In all our examples, we use only three objective function terms, $\mathbf{E}_{\text{obj}}, \mathbf{E}_{\text{reg}}, \mathbf{E}_{\text{RNN}}$. In addition, we use simple forms of objective functions and rely on the Markovian regularization to avoid sub-optimal local minima. For example, to generate a walking example, we set:

$$\mathbf{E}_{\text{obj}} = \sum_{i=2}^{N} \|\dot{\mathbf{COM}}(\mathbf{x}_i, \mathbf{x}_{i-1}) - \dot{\mathbf{COM}}^*\|^2, \tag{17}$$

where $\dot{\mathbf{COM}}$ is the speed of the center of mass and $\dot{\mathbf{COM}}^*$ is the target speed. To generate a jumping example, we set:

$$\mathbf{E}_{\text{obj}} = \|\mathbf{COM}(\mathbf{x}_i) - \mathbf{COM}^*\|^2. \tag{18}$$

Before optimization, we rescale $\mathbf{E}_{\text{obj}}$ so that its initial value is 1.0. Finally, we set $\mathbf{E}_{\text{reg}}$ to be a very small energy efficiency term $\mathbf{E}_{\text{reg}} = 10^{-4}\|\mathcal{T}_{\mathbf{u}}\|^2$. When using spline interpolation, we always set $Z = N/10$ or interpolate 10 timesteps using 2 control points.

8.1 Accuracy of Force Model

Any smooth force models suffer from model discrepancy. If we consider the Dry Friction model [Pennestrì et al. 2016] as the groundtruth, then our smooth model suffers from two kinds of model discrepancy: the lack of static/sliding friction mode switch and the penetration between a character and the obstacle. We choose to ignore the static/sliding friction mode switch because it has been shown in previous works [Boone and Hodgins 1997; Marvi et al. 2014; Yu et al. 2014] that, for various nature-inspired robots, the optimal locomotion gaits usually avoid sliding frictions.

For the second kind of model discrepancy, we can use a technique similar to that in [Mordatch et al. 2013] to reduce the penetrations between obstacles and also avoid ill-conditioned optimization problems. Specifically, we run multi-passes of trajectory optimizations using increasing $w_\perp, w_\parallel$, where earlier passes rapidly bring the trajectory close to its local minima and later passes refine the result to minimize penetrations. As shown in Figure 7, the convergence speed of the second and third passes is much higher than that of the first pass. As $w_\perp, w_\parallel \to \infty$, our model coincides with the dry frictional model with an infinite frictional coefficient.

8.2 Comparison with Other Force Models

A common limitation of [Mordatch et al. 2013] and [Posa et al. 2014] is that each contact point introduces a new term to the objective function ($\mathbf{E}_{\text{int}}$ in the case of [Mordatch et al. 2013]) or a new hard constraint (in the case of [Posa et al. 2014]). As a result, these formulations are not invariant to the number of contact points, which increases the dimension of the search space. In summary, most prior trajectory optimization formulations jointly optimize $\mathcal{T}_{\mathbf{x}}, \mathcal{T}_{\mathbf{f}}$.

Our smooth force model is similar to that in [Tassa et al. 2012], which also eliminates $\mathcal{T}_{\mathbf{f}}$ by solving it as a function of $\mathbf{x}$. However, our method differs from [Tassa et al. 2012] in three important ways. First, since we use an implicit time integrator, our dynamics model is stable under large timestep sizes. This property allows us to further reduces the problem size by using less timesteps and larger Δt. By comparison, [Tassa et al. 2012] reported stability when $\Delta t < 10$ms. Second, the dynamics model proposed by [Tassa et al. 2012] does not allow analytic derivative computation so that their formulation relies on finite-difference derivative evaluations. [Mordatch et al. 2012] and [Mordatch et al. 2013] suffers from the same problem.

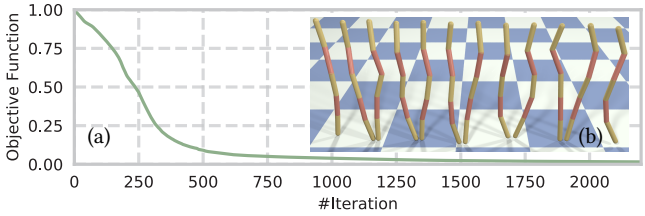


Fig. 8. Convergence history (a) of Algorithm 1 generating 40s of animation (b) for a 14-DOF chain, taking 110s of computation.

8.3 Performance of Optimization

Our first benchmark is illustrated in Figure 8. The goal is to have a 5-link, 14-DOF ($|\mathbf{x}| = 14$) chain move on the ground by jumping around. In this example, we do not use Markovian regularization and only use Algorithm 1. Our algorithm can work in two modes: full-horizon and receding-horizon. In the case of full-horizon, we generate a long animation of 40s using a timestep size of 0.05s ($N = 800, Z = 80$). Our optimizer takes 110s of computation and converges in 2200 iterations. The average time for performing each iteration of Algorithm 1 is 48ms and the cost of each substep is illustrated in Figure 9 (a) where the derivative computation and the matrix factorization are the most costly substeps.

In the case of receding-horizon, we always optimize a sub-trajectory of 1s starting from the current timestep. Unfortunately, although each optimization is faster, taking $3-5$s, the entire animation of 40s takes 1200s of computation to generate. This performance is slower than state-of-the-art [Tassa et al. 2012]. This is because our GPU-algorithm relies on long trajectories to exploit parallelism. As illustrated in Figure 9 (b), the total computational time increases slowly with the number of timesteps N or the number of spline control points Z. Therefore, our method is designed for efficiently generating long trajectories. In many applications, such as deep reinforcement learning [Peng et al. 2017] and motion graph construction [Wampler et al. 2013], we need to optimize many short trajectories. In these cases, we can optimize short trajectories in batches to better exploit GPU parallelism, as shown in Section 8.6.

In Figure 9 (c), we show the computational time to finish one iteration of Algorithm 1 under different number of chain links. The bottleneck of the computation is the matrix factorization which has parallel algorithmic complexity: $O(|\mathbf{x}|^2)$. This is consistent with our observation.

8.4 Comparison with Other Optimization Formulations

We compare the performances of different trajectory optimization formulations and different optimizers in our second benchmark (Figure 10 (a)), where we focus on computing long trajectories. The benchmark requires generating a 10s walking animation for a 4-legged, 18-DOF mammal. For this animation, we use a timestep size of 0.025s ($N = 400, Z = 40$). Again, we do not use Markovian regularization and only use Algorithm 1.

Soft Constrains: We first treat $\mathbf{C}$ as soft constraints so that we can use an unconstrained optimizer such as the LM algorithm [Nocedal and Wright 2006, Chapter 10.2] and we have also implemented a GPU-based LM algorithm. A GPU-based LM algorithm achieves

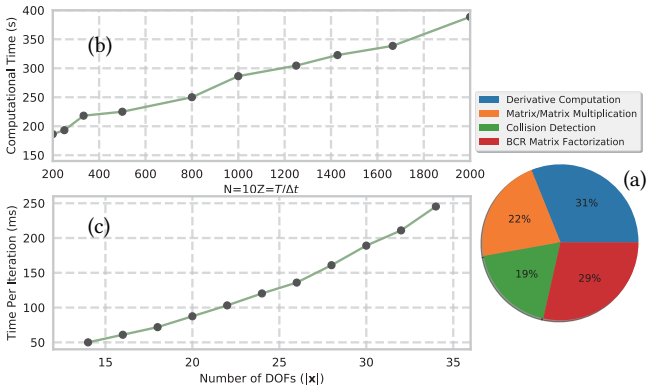
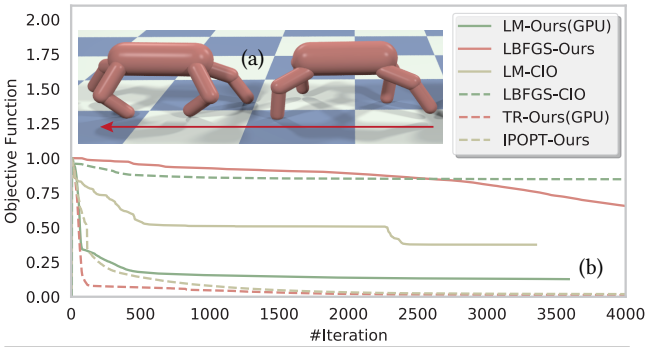


Fig. 9. (a): We plot the percentage of computation spent on the 4 kinds of main computations on GPU: computing the derivatives of **FD**, matrix/matrix multiplication, collision detection, and the BCR algorithm. (b): We plot the computational time of performing 5000 iterations of Algorithm 1 under different trajectory lengths. For optimizations that terminate early, we extrapolate the results. (c): We plot the computational time for performing one iteration of Algorithm 1, using different number of chain links.



Method	LM-Ours GPU	LM-Ours	LBFGS-Ours	LM-CIO	LBFGS-CIO	TR GPU	TR	IPOPT
Time(s)	**124**	10192	6497	12632	8145	**200**	8740	6116

Fig. 10. Our second benchmark (a) generates a 10s walking animation for an 18-DOF mammal. The convergence histories of different formulations are shown in (b). LM/LBFGS-Ours: our formulation with soft constraints optimized using the LM/LBFGS algorithm; LM/LBFGS-CIO: CIO formulation [Mordatch et al. 2013] optimized using the LM/LBFGS algorithm; TR/IPOPT-Ours: our formulation with hard constraints optimized using Algorithm 1/IPOPT. Note that GPU-based implementation is numerically identical to CPU-based implementation so that their convergence histories coincide. The computational overheads of different formulations are shown in the Table below. Our method is 30× faster than the CPU counterpart under the same formulation.

80× speedup over its CPU-based counterpart, taking 124s of computation. However, soft-constrained optimization (LM-Ours) leads to a physics-constraint violation of $\|\mathbf{C}\|_\infty = 0.012N$, while hard-constrained formulation (Algorithm 1) reduces this violation to less than $\|\mathbf{C}\|_\infty = 10^{-5}N$ (both **FD** and **C** have Newton(N) as their units). In practice, soft constraints require additional parameter tuning and can lead to erroneous behavior such as characters floating on the ground (see accompanying video).

Newton-Type Method: In addition, we show that a Newton-type method is necessary in our formulation. We compare the performances of the LM algorithm and the LBFGS algorithm, which

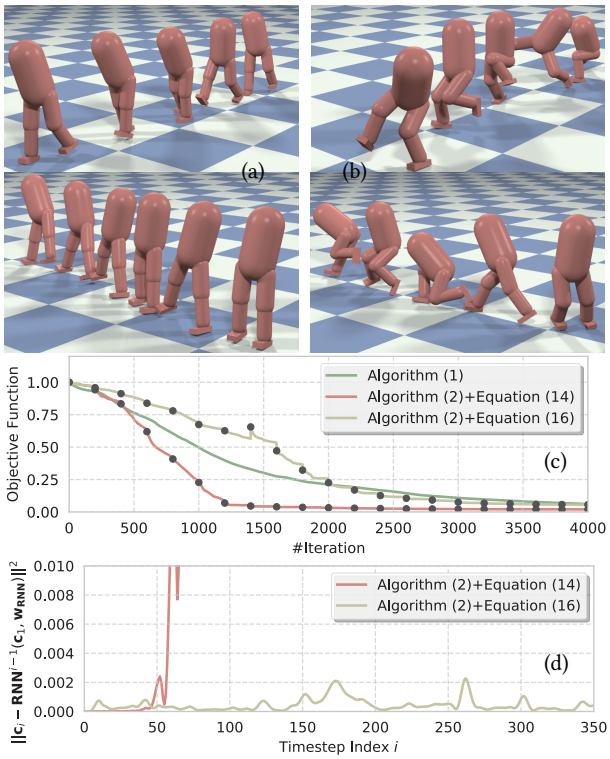


Fig. 11. A comparison of trajectory optimization with (a) and without (b) Markovian regularization. With Markovian regularization modeled using Equation 14, our method generates realistic walking gaits with simple objective functions. The convergence history is shown in (c), where every black dot indicates the RNN update, which happens every 200 iterations. Algorithm 2 outputs a trajectory and a RNN. We plot the stability of the RNN in (d). RNN trained with Equation 14 quickly becomes unstable, while that trained with Equation 16 has long-term stability.

only uses gradient information. Figure 10 (b) shows that the convergence speed of the LBFGS algorithm (LBFGS-Ours) is much slower than that of the LM algorithm (LM-Ours). As a result, although each iteration of the LBFGS algorithm is faster, Newton-type method such as Algorithm 1 always leads to lower overall computational cost.

Hard Constrains: GPU-based Algorithm 1 achieves 40× speedup over its CPU-based counterpart, taking 200s of computation which is slightly more costly than the soft-constrained formulation. In addition, we compare the performances of our method with an off-the-self optimizer, IPOPT [Wächter and Biegler 2006], which is a line-search-based Newton-type optimizer implemented on a CPU. The convergence speed of IPOPT is comparable to our method, but our GPU-based optimizer achieves 30× speedup over IPOPT.

Comparison with CIO: We have also compared our formulation with another well-known soft-constrained trajectory optimization formulation: the contact-invariant optimization (CIO) [Mordatch et al. 2013]. However, this formulation is not invariant to the number of contact points, so that we manually label the four legs of the mammal as its possible contact points. For each contact point **p**, CIO

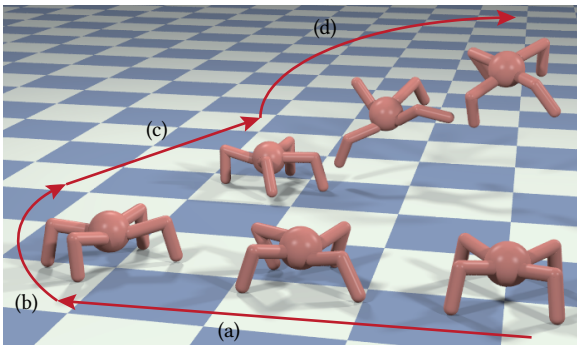


Fig. 12. An animation generated from our optimized motion graph, where the spider moves forward (a), turns right (b), moves forward (c), and finally jumps up (d).

introduces additional soft constraints:

$$\mathbf{E}_{\text{int}} = \|\mathbf{d}(\mathbf{P}(\mathbf{x}_i))\|^2 + \left[0.5\mathbf{tanh}(20\|\mathbf{f}_{\perp}^{E}\| - 2) + 0.5\right] \left[\|(\mathbf{I} - \mathbf{n}\mathbf{n}^T)\dot{\mathbf{P}}(\mathbf{x}_i, \mathbf{x}_{i-1})\|^2 + \|\mathbf{dist}(\mathbf{P}(\mathbf{x}_i))\|^2\right],$$

where the first term penalizes obstacle penetrations, the second term ensures that points under contact forces must be on the obstacle surfaces (**dist** is the signed distance to obstacle surfaces) with zero tangential velocities. Since CIO jointly optimizes $\mathbf{x}$ and $\mathbf{f}^E$, the structure of the Hessian matrix takes a more complex form, which cannot be factorized in parallel. As a result, we implement CIO on CPU using our position-based discretization scheme, so that we retain the ability to compute analytic derivatives. We compare the performance of our formulation and CIO using two optimizers, the LM algorithm and the LBFGS algorithm. The convergence speeds of LM-CIO and LM-Ours are comparable, taking around 3000 iterations. However, our GPU-based LM algorithm achieves 101 times speedup over our CPU-based CIO implementation due to repeated matrix factorizations without exploiting sparsity patterns. After optimization, CIO achieves a physics violation of $\|\mathbf{FD}\|_\infty = 0.008N$, which is better than our soft-constrained formulation but worse than our hard-constrained formulation. In our accompanying video, we compared the two formulations visually and found that they generate animations of similar qualities.

8.5 Markovian Regularization

In our third benchmark, we illustrate the effect of Markovian regularization. The benchmark generates two 10s animations for a 14-DOF bipedal walker to walk forward and stride sideways. We use a timestep size of 0.025s ($N = 400, Z = 40$). Since we do not use reward shaping, the simple objective function (Equation 17) does not generate realistic walking gaits, as shown in Figure 11 (b), where the bipedal takes uneven step sizes. With Markovian regularization, Algorithm 2+Equation 14 generates realistic walking gaits as shown in Figure 11 (a). However, the average computational time to generate these two animations is 510s, in which only 42% of the computational time (214s) is spent on updating the trajectory ($\mathcal{T}_\mathbf{c}$), and 58% of the computational time (295s) is spent on updating the RNN (w_{RNN}). In other words, Algorithm 2 is almost twice as costly as Algorithm 1. In Figure 11 (c), we plot the convergence history of Algorithm 2+Equation 14.

When using Algorithm 2+Equation 16, we achieve long-term stability of RNN as shown in Figure 11 (d). However, the total computational time further increases to 579s, where only 35% of the computational time (202s) is spent on updating the trajectory ($\mathcal{T}_\mathbf{c}$), and 65% of the computational time (376s) is spent on updating the RNN (w_{RNN}). This is because the loss function Equation 16 does not allow parallelism in the temporal domain. Note that, since we use an asymmetric loss function, Algorithm 2 does not monotonically decrease objective function. However, it always converges in our experiments.

8.6 Dense Motion Graph Construction

In our last benchmark, we highlight the robustness and efficiency of our method by constructing a dense motion graph [Kovar et al. 2002] for two character models: an 18-DOF 4-legged spider and an 18-DOF 4-legged mammal. We use a similar approach to that in [Wampler et al. 2013] to construct our motion graph. For each character, we first optimize four trajectories using Algorithm 2: walk forward, turn left, turn right, and jump, each lasting for 10s using a timestep size of 0.025s ($N = 400, Z = 40$). Next, we find a loop in each trajectory by connecting a pair of the most similar control points:

$$\underset{\mathbf{c}_1, \mathbf{c}_2 \in \mathcal{T}_\mathbf{c}}{\mathbf{argmin}}\ \mathbf{SimMetric}(\mathbf{c}_1, \mathbf{c}_2),$$

where $\mathbf{SimMetric}(\bullet, \bullet)$ is the similarity metric proposed in [Kovar et al. 2002]. Finally, for each pair of control points $\{\mathbf{c}_1, \mathbf{c}_2 | \mathbf{c}_1 \in \mathcal{T}_\mathbf{c}^1, \mathbf{c}_2 \in \mathcal{T}_\mathbf{c}^2, \mathcal{T}_\mathbf{c}^1 \neq \mathcal{T}_\mathbf{c}^2\}$ from two different trajectories, we optimize a short transitional trajectory lasting for 1s that starts from $\mathbf{c}_1$ and ends at $\mathbf{c}_2$. To perform this computation, we use Algorithm 1 with $\mathbf{E}_{\text{obj}} = 0$ and two end-point constraints. If Algorithm 1 can successfully find a trajectory that satisfies the hard constraints, we add a transitional path to the motion graph from $\mathbf{c}_1$ to $\mathbf{c}_2$. As a result, we construct a dense motion graph with as many transitional edges as possible, making it more responsive to user requests of task changes. Since a short trajectory lasting for 1s does not exploit GPU-parallelism, we optimize all the transitional trajectories in a batch manner. Our GPU-based algorithm can find the motion graph within 60min of computation for each characters. An animation generated from this motion graph is illustrated in Figure 12.

9 CONCLUSION, LIMITATIONS AND FUTURE WORK

We present a GPU-friendly formulation of trajectory optimization for character locomotions. By using a smooth force model, our approach treats the frictional contact forces and control forces as functions of the kinematic state. This formulation features a smaller problem size, contact invariance, fixed sparsity pattern of the Hessian matrix, and analytic derivative computations. Physical correctness is guaranteed by treating them as hard constraints, and we use constraint reduction to make hard constraints compatible under spline interpolation. Finally, we use a Markovian regularization term to avoid sub-optimal local minima. We show that our GPU-based algorithm can generate complex locomotions for various character models and achieve more than 30× speedup over a CPU-based counterpart.

Our approach has some limitations. First, our frictional contact force model is not as accurate as the conventional dry friction model.

Our model does not support the static/sliding friction mode switch. In addition, Equation 6 does not support frictional cone constraints and it assumes friction forces change continuously according to the tangent velocity. A second limitation is that, to derive the compact formulation, we assume the use of a simple controller. It has been shown in [Lee et al. 2018; Mordatch et al. 2013] that more natural locomotion gaits are generated when more general controllers are used, such as Muscle-Tendon units (MTUs). Finally, our entire dynamics model is a C^1-continuous function of the kinematic state. However, the Hessian of a C^1 function does not exist. As a result, we have to use a $\mathbf{J}^T\mathbf{J}$-approximate Hessian in Algorithm 1. This treatment can have a negative impact on the convergence rate of the TRSQP optimizer and further investigation is left as future work. Another avenue of future work is to consider graspable contacts [Mordatch et al. 2012] where a point can be under both pulling and pushing forces. In addition, the results on Markovian regularization open doors to future research on learning recurrent neural networks directly from physics rules, see Appendix D for more discussions. Finally, while there is a lot of work on trajectory optimization, there is very little work on GPU parallelization for dynamics applications. Therefore, it would be useful to develop GPU version of other trajectory optimization methods such as CIO and compare with our method.

ACKNOWLEDGMENTS

This research is supported in part by ARO grant W911NF-18-1-0313, and Intel.

REFERENCES

Yunfei Bai, Danny M. Kaufman, C. Karen Liu, and Jovan Popović. 2016. Artist-directed Dynamics for 2D Animation. *ACM Trans. Graph.* 35, 4, Article 145 (July 2016), 10 pages.

Jernej Barbič, Marco da Silva, and Jovan Popović. 2009. Deformable Object Animation Using Reduced Optimal Control. In *ACM SIGGRAPH 2009 Papers (SIGGRAPH '09)*. ACM, New York, NY, USA, Article 53, 9 pages.

Jernej Barbič, Funshing Sin, and Eitan Grinspun. 2012. Interactive Editing of Deformable Simulations. *ACM Trans. Graph.* 31, 4, Article 70 (July 2012), 8 pages.

Nathan Bell and Jared Hoberock. 2011. Thrust: A productivity-oriented library for CUDA. In *GPU computing gems Jade edition*. Elsevier, 359–371.

Jan Bender, Dan Koschier, Patrick Charrier, and Daniel Weber. 2014. Position-based simulation of continuous materials. *Computers & Graphics* 44 (2014), 1 – 10.

P. Benner, S. Gugercin, and K. Willcox. 2015. A Survey of Projection-Based Model Reduction Methods for Parametric Dynamical Systems. *SIAM Rev.* 57, 4 (2015), 483–531.

Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröoder. 2003. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. In *ACM transactions on graphics (TOG)*, Vol. 22. ACM, 917–924.

Gary N. Boone and Jessica K. Hodgins. 1997. Slipping and Tripping Reflexes for Bipedal Robots. *Autonomous Robots* 4, 3 (01 Sep 1997), 259–271.

Sofien Bouaziz, Sebastian Martin, Tiantian Liu, Ladislav Kavan, and Mark Pauly. 2014. Projective Dynamics: Fusing Constraint Projections for Fast Simulation. *ACM Trans. Graph.* 33, 4, Article 154 (July 2014), 11 pages.

A. Byravan, B. Boots, S. S. Srinivasa, and D. Fox. 2014. Space-time functional gradient optimization for motion planning. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*. 6499–6506.

B. Chretien, A. Escande, and A. Kheddar. 2016. GPU Robot Motion Planning Using Semi-Infinite Nonlinear Programming. *IEEE Transactions on Parallel and Distributed Systems* 27, 10 (Oct 2016), 2926–2939.

Crispin Deul, Patrick Charrier, and Jan Bender. 2014. Position-based rigid-body dynamics. *Computer Animation and Virtual Worlds* 27, 2 (2014), 103–112.

Charles Dugas, Yoshua Bengio, François Bélisle, Claude Nadeau, and René Garcia. 2000. Incorporating Second-order Functional Knowledge for Better Option Pricing. In *Proceedings of the 13th International Conference on Neural Information Processing Systems (NIPS'00)*. MIT Press, Cambridge, MA, USA, 451–457.

Jeffrey L. Elman. 1990. Finding structure in time. *Cognitive Science* 14, 2 (1990), 179 – 211.

Walter Gander and Gene H Golub. 1997. Cyclic reduction: history and applications. *Scientific computing (Hong Kong, 1997)* (1997), 73–85.

T. F. Gast, C. Schroeder, A. Stomakhin, C. Jiang, and J. M. Teran. 2015. Optimization Integrator for Large Time Steps. *IEEE Transactions on Visualization and Computer Graphics* 21, 10 (Oct 2015), 1103–1115.

Sehoon Ha and C. Karen Liu. 2014. Iterative Training of Dynamic Skills Inspired by Human Coaching Techniques. *ACM Trans. Graph.* 34, 1, Article 1 (Dec. 2014), 11 pages.

Fabian Hahn, Sebastian Martin, Bernhard Thomaszewski, Robert Sumner, Stelian Coros, and Markus Gross. 2012. Rig-space Physics. *ACM Trans. Graph.* 31, 4, Article 72 (July 2012), 8 pages.

Perttu Hämäläinen, Joose Rajamäki, and C. Karen Liu. 2015. Online Control of Simulated Humanoids Using Particle Belief Propagation. *ACM Trans. Graph.* 34, 4, Article 81 (July 2015), 13 pages.

Kris K. Hauser, Chen Shen, and James F. O'Brien. 2003. Interactive Deformation Using Modal Analysis with Constraints. In *Graphics Interface*. CIPS, Canadian Human-Computer Commnication Society, 247–256.

S. Heinrich, A. Zoufahl, and R. Rojas. 2015. Real-time trajectory optimization under motion uncertainty using a GPU. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 3572–3577.

Jessica K. Hodgins, Wayne L. Wooten, David C. Brogan, and James F. O'Brien. 1995. Animating Human Athletics. In *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '95)*. ACM, New York, NY, USA, 71–78.

A. F. Izmailov, A. L. Pogosyan, and M. V. Solodov. 2011. Semismooth SQP method for equality-constrained optimization problems with an application to the lifted reformulation of mathematical programs with complementarity constraints. *Optimization Methods and Software* 26, 4-5 (2011), 847–872.

Changgu Kang and Sung-Hee Lee. 2017. Multi-Contact Locomotion Using a Contact Graph with Feasibility Predictors. *ACM Trans. Graph.* 36, 4, Article 145b (April 2017).

Lucas Kovar, Michael Gleicher, and Frédéric Pighin. 2002. Motion Graphs. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '02)*. ACM, New York, NY, USA, 473–482.

Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. 2009. Fast BVH construction on GPUs. In *Computer Graphics Forum*, Vol. 28. Wiley Online Library, 375–384.

Seunghwan Lee, Ri Yu, Jungnam Park, Mridul Aanjaneya, Eftychios Sifakis, and Jehee Lee. 2018. Dexterous Manipulation and Control with Volumetric Muscles. *ACM Trans. Graph.* 37, 4, Article 57 (July 2018), 13 pages.

Yoonsang Lee, Moon Seok Park, Taesoo Kwon, and Jehee Lee. 2014. Locomotion Control for Many-muscle Humanoids. *ACM Trans. Graph.* 33, 6, Article 218 (Nov. 2014), 11 pages.

Sergey Levine and Vladlen Koltun. 2013. Guided Policy Search. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28 (ICML'13)*. JMLR.org, III–1–III–9.

Sergey Levine, Jack M. Wang, Alexis Haraux, Zoran Popović, and Vladlen Koltun. 2012. Continuous Character Control with Low-dimensional Embeddings. *ACM Trans. Graph.* 31, 4, Article 28 (July 2012), 10 pages.

C. Karen Liu, Aaron Hertzmann, and Zoran Popović. 2005. Learning Physics-based Motion Style with Nonlinear Inverse Optimization. *ACM Trans. Graph.* 24, 3 (July 2005), 1071–1081.

Libin Liu and Jessica Hodgins. 2017. Learning to Schedule Control Fragments for Physics-Based Characters Using Deep Q-Learning. *ACM Trans. Graph.* 36, 3, Article 42a (June 2017).

Libin Liu, KangKang Yin, Michiel van de Panne, Tianjia Shao, and Weiwei Xu. 2010. Sampling-based Contact-rich Motion Control. *ACM Trans. Graph.* 29, 4, Article 128 (July 2010), 10 pages.

Miles Macklin and Matthias Müller. 2013. Position Based Fluids. *ACM Trans. Graph.* 32, 4, Article 104 (July 2013), 12 pages.

Hamidreza Marvi, Chaohui Gong, Nick Gravish, Henry Astley, Matthew Travers, Ross L. Hatton, Joseph R. Mendelson, Howie Choset, David L. Hu, and Daniel I. Goldman. 2014. Sidewinding with minimal slip: Snake and robot ascent of sandy slopes. *Science* 346, 6206 (2014), 224–229.

Igor Mordatch, Kendall Lowrey, Galen Andrew, Zoran Popovic, and Emanuel V. Todorov. 2015. Interactive Control of Diverse Complex Characters with Neural Networks. In *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett (Eds.). Curran Associates, Inc., 3132–3140.

Igor Mordatch, Emanuel Todorov, and Zoran Popović. 2012. Discovery of Complex Behaviors Through Contact-invariant Optimization. *ACM Trans. Graph.* 31, 4, Article 43 (July 2012), 8 pages.

Igor Mordatch, Jack M. Wang, Emanuel Todorov, and Vladlen Koltun. 2013. Animating Human Lower Limbs Using Contact-invariant Optimization. *ACM Trans. Graph.* 32, 6, Article 203 (Nov. 2013), 8 pages.

Matthias Müller, Bruno Heidelberger, Marcus Hennix, and John Ratcliff. 2007. Position Based Dynamics. *J. Vis. Comun. Image Represent.* 18, 2 (April 2007), 109–118.

Richard M. Murray, S. Shankar Sastry, and Li Zexiang. 1994. *A Mathematical Introduction to Robotic Manipulation* (1st ed.). CRC Press, Inc., Boca Raton, FL, USA.

Kourosh Naderi, Joose Rajamäki, and Perttu Hämäläinen. 2017. Discovering and Synthesizing Humanoid Climbing Movements. *ACM Trans. Graph.* 36, 4, Article 43 (July 2017), 11 pages.

Jorge Nocedal and Stephen J. Wright. 2006. *Numerical Optimization, second edition.* World Scientific.

CUDA Nvidia. 2008. Cublas library. *NVIDIA Corporation, Santa Clara, California* 15, 27 (2008), 31.

Zherong Pan and Dinesh Manocha. 2018. Position-Based Time-Integrator for Frictional Articulated Body Dynamics. In *Algorithmic Foundations of Robotics XIV*. Springer.

C. Park, J. Pan, and D. Manocha. 2013. Real-time optimization-based planning in dynamic environments using GPUs. In *2013 IEEE International Conference on Robotics and Automation*. 4090–4097.

Xue Bin Peng, Glen Berseth, Kangkang Yin, and Michiel Van De Panne. 2017. DeepLoco: Dynamic Locomotion Skills Using Hierarchical Deep Reinforcement Learning. *ACM Trans. Graph.* 36, 4, Article 41 (July 2017), 13 pages.

Ettore Pennestrì, Valerio Rossi, Pietro Salvini, and Pier Paolo Valentini. 2016. Review and comparison of dry friction force models. *Nonlinear dynamics* 83, 4 (2016), 1785–1801.

Brian Plancher and Scott Kuindersma. 2018. A Performance Analysis of Parallel Differential Dynamic Programming on a GPU. In *Algorithmic Foundations of Robotics XIV*. Springer.

Michael Posa, Cecilia Cantu, and Russ Tedrake. 2014. A direct method for trajectory optimization of rigid bodies through contact. *The International Journal of Robotics Research* 33, 1 (2014), 69–81.

Ananth Ranganathan. 2004. The levenberg-marquardt algorithm. *Tutoral on LM algorithm* 11, 1 (2004), 101–110.

John Schulman, Yan Duan, Jonathan Ho, Alex Lee, Ibrahim Awwal, Henry Bradlow, Jia Pan, Sachin Patil, Ken Goldberg, and Pieter Abbeel. 2014. Motion planning with sequential convex optimization and convex collision checking. *The International Journal of Robotics Research* 33, 9 (2014), 1251–1270.

David E. Stewart. 2000. Rigid-Body Dynamics with Friction and Impact. *SIAM Rev.* 42, 1 (March 2000), 3–39.

Y. Tassa, T. Erez, and E. Todorov. 2012. Synthesis and stabilization of complex behaviors through online trajectory optimization. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 4906–4913.

Pierre Thodoroff, Audrey Durand, Joelle Pineau, and Doina Precup. 2018. Temporal Regularization for Markov Decision Process. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.). Curran Associates, Inc., 1784–1794.

Adrien Treuille, Antoine McNamara, Zoran Popović, and Jos Stam. 2003. Keyframe Control of Smoke Simulations. *ACM Trans. Graph.* 22, 3 (July 2003), 716–723.

Andreas Wächter and Lorenz T Biegler. 2006. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical programming* 106, 1 (2006), 25–57.

Kevin Wampler and Zoran Popović. 2009. Optimal Gait and Form for Animal Locomotion. *ACM Trans. Graph.* 28, 3, Article 60 (July 2009), 8 pages.

Kevin Wampler, Jovan PopoviÄĞ, and Zoran PopoviÄĞ. 2013. Animal Locomotion Controllers From Scratch. *Computer Graphics Forum* 32, 2pt2 (2013), 153–162.

Huahua Wang and Arindam Banerjee. 2014. Bregman Alternating Direction Method of Multipliers. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2 (NIPS'14)*. MIT Press, Cambridge, MA, USA, 2816–2824.

Jack M. Wang, Samuel R. Hamner, Scott L. Delp, and Vladlen Koltun. 2012. Optimizing Locomotion Controllers Using Biologically-based Actuators and Objectives. *ACM Trans. Graph.* 31, 4, Article 25 (July 2012), 11 pages.

Martin Wicke, Matt Stanton, and Adrien Treuille. 2009. Modular Bases for Fluid Dynamics. *ACM Trans. Graph.* 28, 3, Article 39 (July 2009), 8 pages.

A. W. Winkler, C. D. Bellicoso, M. Hutter, and J. Buchli. 2018. Gait and Trajectory Optimization for Legged Systems Through Phase-Based End-Effector Parameterization. *IEEE Robotics and Automation Letters* 3, 3 (July 2018), 1560–1567.

Andrew Witkin and Michael Kass. 1988. Spacetime Constraints. In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '88)*. ACM, New York, NY, USA, 159–168.

Jungdam Won, Jongho Park, Kwanyu Kim, and Jehee Lee. 2017. How to Train Your Dragon: Example-guided Control of Flapping Flight. *ACM Trans. Graph.* 36, 6, Article 198 (Nov. 2017), 13 pages.

Q. Wu, F. Xiong, F. Wang, and Y. Xiong. 2016. Parallel particle swarm optimization on a graphics processing unit with application to trajectory optimization. *Engineering Optimization* 48, 10 (2016), 1679–1692.

Zexiang Xu, Hsiang-Tao Wu, Lvdi Wang, Changxi Zheng, Xin Tong, and Yue Qi. 2014. Dynamic Hair Capture Using Spacetime Optimization. *ACM Trans. Graph.* 33, 6, Article 224 (Nov. 2014), 11 pages.

KangKang Yin, Kevin Loken, and Michiel van de Panne. 2007. SIMBICON: Simple Biped Locomotion Control. In *ACM SIGGRAPH 2007 Papers (SIGGRAPH '07)*. ACM, New York, NY, USA, Article 105.

Zhangguo Yu, Jing Li, Qiang Huang, Xuechao Chen, Gan Ma, Libo Meng, Si Zhang, Yan Liu, Wen Zhang, Weimin Zhang, et al. 2014. Slip prevention of a humanoid robot by coordinating acceleration vector. In *Information and Automation (ICIA), 2014 IEEE International Conference on*. IEEE, 683–688.

He Zhang, Sebastian Starke, Taku Komura, and Jun Saito. 2018. Mode-adaptive Neural Networks for Quadruped Motion Control. *ACM Trans. Graph.* 37, 4, Article 145 (July 2018), 11 pages.

Matt Zucker, Nathan Ratliff, Anca D. Dragan, Mihail Pivtoraiko, Matthew Klingensmith, Christopher M. Dellin, J. Andrew Bagnell, and Siddhartha S. Srinivasa. 2013. CHOMP: Covariant Hamiltonian optimization for motion planning. *The International Journal of Robotics Research* 32, 9-10 (2013), 1164–1193.