Fast Model Predictive Control for Reactive Robotic Swimming

Yuval Tassa
Center for Neural Computation
Hebrew University
Jerusalem, Israel
tassa@alice.nc.huji.ac.il

Tom Erez
Dep. of Computer Science
Washington University
St. Louis, MO, USA
etom@cse.wustl.edu

Emo Todorov
Dep. of Computer Science and Engineering
Dep. of Applied Mathematics
University of Washington
Seattle, WA, USA
todorov@cs.washington.edu

***Abstract*—Model Predictive Control amounts to re-solving a trajectory optimization problem at every timestep. Using a fast multi-threaded simulator and an efficient trajectory optimizer, we generate complex, reactive and robust swimming behavior in simulated robots of 12-27 state dimensions, with time-steps in the range of 10-40ms.**

I. INTRODUCTION

Optimal Control promises a simple, principled approach to controller synthesis: quantify task performance, and a numerical algorithm will automatically generate the optimal global policy. In practice, the exponential scaling of computational complexity with the state dimension makes this impossible for all but the simplest systems.

Online trajectory optimization, also known as Model Predictive Control, avoids searching over the entire state-space, instead finding the optimal policy for only one particular state – the current one. The control is applied to the system, the clock advances, and the problem is re-solved, typically using the previous solution to warm-start the solver. Unlike global algorithms, MPC scales polynomially with the dimension of the state.

The challenge in applying MPC in robotics is that optimization must happen in real-time, matching the timescale of the robot. MPC was initially developed in the chemical process industry, where timescales of $10s$ or more are typical [2]. In robotics, a timescale of $10ms$ is not uncommon, demanding careful design of both algorithm and implementation. Fortunately, computational hardware is constantly becoming faster and cheaper, and has reached the point where efficient optimization is possible on these timescales.

As we demonstrate below, the bottleneck for online optimization is in computation of the dynamics (and its derivatives). To that end, we introduce a new general-purpose physics simulator, written expressly to take advantage of modern multi-core, multi-threaded environments. Running on standard hardware, we can perform $10^5 - 10^6$ evaluation the dynamics per second for multi-body systems with dozens of state-dimensions.

Due to the formidable computational challenge, MPC has been rarely used in robotics. The most impressive example we are aware of is aerobatic helicopter flight[1]. In that work, policies were updated every $50ms$, using a linearized 12-dimensional model, with the control objective of tracking a pre-specified trajectory.

Below, we demonstrate MPC on systems with $12-27$ state dimensions, using quadratized models, updating the policy every $10-30ms$. We generate complex and robust swimming behavior on the fly, letting the user interact with the system and modify various parameters in real-time. Importantly, the cost which encodes the control objective – to reach target while avoiding obstacles – is a function only of the 2 cartesian positions, effectively defining a target manifold (configurations where the nose is on the target), rather than a single point in state-space. The freedom to reach this manifold on different paths manifests as diverse swimming and twisting behaviour.

As described below, optimization can generate rich behaviors from very weak assumptions and simple cost functions. In some ways, this is similar to the classic work of Karl Sims [10] on optimization-based behavior, yet happens in real time rather than requiring hours of offline computation on a supercomputer. In this paper we used swimming as a test-bed, but the same efficient tools should allow us to tackle more complex tasks such as walking and hand manipulation.

This paper makes three contributions. First, we demonstrate for the first time that MPC can be applied to fast nonlinear dynamics ($dt \approx 10ms$) with such high dimensionality ($n \approx 20$). Since many interesting robots have comparable dimensionality, our methodology promises to deliver substantial improvements in a range of robotic control tasks. Second, on the algorithmic level, we provide a detailed description of our DDP-based trajectory optimization algorithm, including improvements to regularization and line-search, and show that a local quadratic model of the dynamics significantly outperforms a linear model. Third, we introduce a fast general-purpose multi-threaded physics engine that can provide a numerical foundation for future online Optimal Control algorithms.

II. MODEL PREDICTIVE CONTROL

Online Optimal Control or MPC involves repeatedly minimizing the cost of a finite-horizon trajectory emanating from the current state of the system. The state $\mathbf{x} \in \mathbb{R}^n$ evolves according to the discrete-time dynamics

$$\mathbf{x}(i+1) = \mathbf{f}(\mathbf{x}(i), \mathbf{u}(i)), \tag{1}$$

where $\mathbf{u} \in \mathbb{R}^m$ is the control. The *cost-to-go* at time i is the sum of running costs $\ell(\mathbf{x}, \mathbf{u}, k)$ and final cost $\ell_f(\mathbf{x})$, incurred when starting from state $\mathbf{x}(i)$ and applying the control sequence $\mathbf{u}_{i..N-1} \equiv \{\mathbf{u}(i), \mathbf{u}(i+1) \dots, \mathbf{u}(N-1)\}$ until the horizon is reached:

$$J(\mathbf{x}(i), \mathbf{u}_{i..N-1}, i) = \sum_{k=i}^{N-1} \ell(\mathbf{x}(k), \mathbf{u}(k), k) + \ell_f(\mathbf{x}(N)). \quad (2)$$

A *trajectory optimizer* is an algorithm which solves

$$\min_{\mathbf{u}_{1..N-1}} J(\mathbf{x}(1), \mathbf{u}_{1..N-1}, 1)$$
$$\text{s.t.} \quad \mathbf{x}(i+1) = \mathbf{f}(\mathbf{x}(i), \mathbf{u}(i)).$$

A useful distinction is made in [3] between *simultaneous* and *sequential* trajectory optimization algorithms. Simultaneous algorithms optimize over the entire sequence of states and controls (treating (1) as a constraint), while sequential algorithms optimize only over the controls, letting the dynamics propagate the states. Clearly simultaneous algorithms search over a much larger space, yet historically they have been preferred, perhaps due to the fact that they easily admit externally imposed state constraints, which are of prime importance in petrochemical process control. In the case of robotics, constraints such as joint limits or contacts are enforced by the plant rather than externally imposed, so sequential algorithms have an advantage.

Once the minimizing control sequence is found, $\mathbf{u}(1)$ is applied to the plant, the clock advances by one time-step, the current state is measured, and the trajectory optimizer is re-applied. If the optimization is iterative, it makes sense to warm-start the optimizer with the time-shifted previous solution, appended with the final control

$$\mathbf{u}_{1..N-1} \leftarrow \{\mathbf{u}(2), \mathbf{u}(3) \dots, \mathbf{u}(N-1), \mathbf{u}(N-1)\}.$$

The classic sequential trajectory optimizer is Pontryagin's Maximum Principle [9], which defines a two-point boundary-value problem whose solution gives the optimal sequence of states. The Maximum Principle is a first-order algorithm in the sense that it requires only first derivatives of the dynamics, but also in the sense that convergence can be quite slow, even close to the minimum. Differential Dynamic Programming (DDP) is a second-order trajectory optimizer introduced in [7] that requires second derivatives of the dynamics, but also features quadratic convergence (i.e. like Newton's Method) near the minimum. In the MPC context, where each initial state is close to the previous one, and the previous solution is used to warm-start the optimization, this feature is indispensable.

III. DIFFERENTIAL DYNAMIC PROGRAMMING

We now present the DDP algorithm using the notation first introduced in [8]. The total cost which we wish to minimize is (2). Defining the optimal Value function at time i as the the cost-to-go given the minimizing control sequence

$$V(\mathbf{x}, i) \equiv \min_{\mathbf{u}_{i..N-1}} J(\mathbf{x}(i), \mathbf{u}_{i..N-1}, i),$$

and setting $V_N(\mathbf{x}) \equiv \ell_f(\mathbf{x}_N)$, the Dynamic Programming Principle reduces the minimization over an entire sequence of controls to a sequence of minimizations over a single control, proceeding backwards in time:

$$V(\mathbf{x}, i) = \min_{\mathbf{u}} [\ell(\mathbf{x}, \mathbf{u}) + V(\mathbf{f}(\mathbf{x}, \mathbf{u}), i+1)] \quad (3)$$

DDP involves a backward pass of Eq. (3) along the current $(\mathbf{x}, \mathbf{u}, i)$ trajectory, recursively constructing a quadratic approximation to $V(\mathbf{x}, i)$ and a linear approximation to $\mathbf{u}(\mathbf{x}, i)$, followed by a forward pass which applies the new control sequence to form a new trajectory. Define the argument of the minimum in (3) as a function of perturbations around the i-th $(\mathbf{x}, \mathbf{u})$ pair:

$$Q(\delta\mathbf{x}, \delta\mathbf{u}) = \ell(\mathbf{x} + \delta\mathbf{x}, \mathbf{u} + \delta\mathbf{u}, i) - \ell(\mathbf{x}, \mathbf{u}, i) + V(\mathbf{f}(\mathbf{x} + \delta\mathbf{x}, \mathbf{u} + \delta\mathbf{u}), i+1) - V(\mathbf{f}(\mathbf{x}, \mathbf{u}), i+1) \quad (4)$$

and expand to second order

$$\approx \frac{1}{2} \begin{bmatrix} 1 \\ \delta\mathbf{x} \\ \delta\mathbf{u} \end{bmatrix}^\mathsf{T} \begin{bmatrix} 0 & Q_\mathbf{x}^\mathsf{T} & Q_\mathbf{u}^\mathsf{T} \\ Q_\mathbf{x} & Q_{\mathbf{xx}} & Q_{\mathbf{xu}} \\ Q_\mathbf{u} & Q_{\mathbf{ux}} & Q_{\mathbf{uu}} \end{bmatrix} \begin{bmatrix} 1 \\ \delta\mathbf{x} \\ \delta\mathbf{u} \end{bmatrix}. \quad (5)$$

The expansion coefficients are[1]

$$Q_\mathbf{x} = \ell_\mathbf{x} + \mathbf{f}_\mathbf{x}^\mathsf{T} V'_\mathbf{x} \quad (6a)$$
$$Q_\mathbf{u} = \ell_\mathbf{u} + \mathbf{f}_\mathbf{u}^\mathsf{T} V'_\mathbf{x} \quad (6b)$$
$$Q_{\mathbf{xx}} = \ell_{\mathbf{xx}} + \mathbf{f}_\mathbf{x}^\mathsf{T} V'_{\mathbf{xx}} \mathbf{f}_\mathbf{x} + V'_\mathbf{x} \cdot \mathbf{f}_{\mathbf{xx}} \quad (6c)$$
$$Q_{\mathbf{uu}} = \ell_{\mathbf{uu}} + \mathbf{f}_\mathbf{u}^\mathsf{T} V'_{\mathbf{xx}} \mathbf{f}_\mathbf{u} + V'_\mathbf{x} \cdot \mathbf{f}_{\mathbf{uu}} \quad (6d)$$
$$Q_{\mathbf{ux}} = \ell_{\mathbf{ux}} + \mathbf{f}_\mathbf{u}^\mathsf{T} V'_{\mathbf{xx}} \mathbf{f}_\mathbf{x} + V'_\mathbf{x} \cdot \mathbf{f}_{\mathbf{ux}}. \quad (6e)$$

Note that the last terms in (6c, 6d, 6e) denote contraction with a tensor. Minimizing (5) WRT $\delta\mathbf{u}$ we have

$$\delta\mathbf{u}^*(i) = \operatorname*{argmin}_{\delta\mathbf{u}} Q(\delta\mathbf{x}, \delta\mathbf{u}) = -Q_{\mathbf{uu}}^{-1}(Q_\mathbf{u} + Q_{\mathbf{ux}}\delta\mathbf{x}), \quad (7)$$

giving us an open-loop term $\mathbf{k} = -Q_{\mathbf{uu}}^{-1} Q_\mathbf{u}$ and a feedback gain term $\mathbf{K} = -Q_{\mathbf{uu}}^{-1} Q_{\mathbf{ux}}$. Plugging the result back in (5), we have a quadratic model of the Value at time i:

$$\Delta V(i) = -\tfrac{1}{2} Q_\mathbf{u} Q_{\mathbf{uu}}^{-1} Q_\mathbf{u} \quad (8a)$$
$$V_\mathbf{x}(i) = Q_\mathbf{x} - Q_\mathbf{u} Q_{\mathbf{uu}}^{-1} Q_{\mathbf{ux}} \quad (8b)$$
$$V_{\mathbf{xx}}(i) = Q_{\mathbf{xx}} - Q_{\mathbf{xu}} Q_{\mathbf{uu}}^{-1} Q_{\mathbf{ux}}. \quad (8c)$$

Recursively computing the local quadratic models of $V(i)$ and the control modifications $\{\mathbf{k}(i), \mathbf{K}(i)\}$, constitutes the backward pass. Once it is completed, a forward pass computes a new trajectory:

$$\hat{\mathbf{x}}(1) = \mathbf{x}(1) \quad (9a)$$
$$\hat{\mathbf{u}}(i) = \mathbf{u}(i) + \mathbf{k}(i) + \mathbf{K}(i)(\hat{\mathbf{x}}(i) - \mathbf{x}(i)) \quad (9b)$$
$$\hat{\mathbf{x}}(i+1) = \mathbf{f}(\hat{\mathbf{x}}(i), \hat{\mathbf{u}}(i)) \quad (9c)$$

[1] Dropping the index i, primes denoting the next time-step: $V' \equiv V(i+1)$.

IV. IMPROVEMENTS TO DDP

A. Regularization

It has been shown [6] that the steps taken by DDP are comparable to or better than a full Newton step for the entire control sequence. As in Newton's method, care must be taken when the Hessian is not positive-definite or when the minimum is far and the quadratic model inaccurate. The standard regularization, proposed in [4] and further explored in [5], is to add a diagonal term to the local control-cost Hessian

$$\widetilde{Q}_{\mathbf{uu}} = Q_{\mathbf{uu}} + \mu \mathbf{I}_m, \tag{10}$$

where μ plays the role of a Levenberg-Marquardt parameter. This modification amounts to adding a quadratic cost around the curren control sequence, making the steps more conservative. The drawback to this regularization scheme is that the same control perturbation can have different effects at different times, depending on the control-transition matrix $\mathbf{f_u}$. We therefore introduce a scheme that penalizes deviations from the states rather than controls:

$$\widetilde{Q}_{\mathbf{uu}} = \ell_{\mathbf{uu}} + \mathbf{f}_{\mathbf{u}}^{\mathsf{T}}(V'_{\mathbf{xx}} + \mu \mathbf{I}_n)\mathbf{f}_{\mathbf{u}} + V'_{\mathbf{x}} \cdot \mathbf{f}_{\mathbf{uu}} \tag{11a}$$

$$\widetilde{Q}_{\mathbf{ux}} = \ell_{\mathbf{ux}} + \mathbf{f}_{\mathbf{u}}^{\mathsf{T}}(V'_{\mathbf{xx}} + \mu \mathbf{I}_n)\mathbf{f}_{\mathbf{x}} + V'_{\mathbf{x}} \cdot \mathbf{f}_{\mathbf{ux}} \tag{11b}$$

$$\mathbf{k} = -\widetilde{Q}_{\mathbf{uu}}^{-1}\widetilde{Q}_{\mathbf{u}} \tag{11c}$$

$$\mathbf{K} = -\widetilde{Q}_{\mathbf{uu}}^{-1}\widetilde{Q}_{\mathbf{ux}} \tag{11d}$$

This regularization amounts to placing a quadratic state-cost around the previous state sequence. Unlike the standard control-based regularization, the feedback gains $\mathbf{K}$ do not vanish for large μ but rather force the new trajectory closer to the old one, significantly improving robustness.

B. Value Update

Examining (5, 7, 8), it is clear that the standard Value update equations (8) assume that several cancelations of $Q_{\mathbf{uu}}$ and its inverse have taken place. Regardless of whether we use the classic regularization (10) or the new one (11), $Q_{\mathbf{uu}}$ is modified and making those cancelations would induce an error in the approximation. The correct update is therefore

$$\Delta V(i) = \quad +\tfrac{1}{2}\mathbf{k}^{\mathsf{T}}Q_{\mathbf{uu}}\mathbf{k} + \mathbf{k}^{\mathsf{T}}Q_{\mathbf{u}} \tag{12a}$$

$$V_{\mathbf{x}}(i) = Q_{\mathbf{x}} + \mathbf{K}^{\mathsf{T}}Q_{\mathbf{uu}}\mathbf{k} + \mathbf{K}^{\mathsf{T}}Q_{\mathbf{u}} + Q_{\mathbf{ux}}^{\mathsf{T}}\mathbf{k} \tag{12b}$$

$$V_{\mathbf{xx}}(i) = Q_{\mathbf{xx}} + \mathbf{K}^{\mathsf{T}}Q_{\mathbf{uu}}\mathbf{K} + \mathbf{K}^{\mathsf{T}}Q_{\mathbf{ux}} + Q_{\mathbf{ux}}^{\mathsf{T}}\mathbf{K}. \tag{12c}$$

C. Line Search

The forward pass of DDP, given by Eqs. (9) is the key to the algorithm's fast convergence. This is because the feedback gains in (9b) generate a new control sequence that takes into account the new states as they are being integrated. For example when applying DDP to a linear-quadratic system, even a time-varying one, an exact solution is obtained after a single iteration. The caveat is that for a general non-linear system, when the new trajectory strays too far from the model's region of validity, the cost may increase and divergence can occur. The solution is to introduce a backtracking line-search parameter $0 < \alpha \leq 1$ and integrate using

$$\hat{\mathbf{u}}(i) = \mathbf{u}(i) + \alpha\mathbf{k}(i) + \mathbf{K}(i)(\hat{\mathbf{x}}(i) - \mathbf{x}(i)) \tag{13}$$

For $\alpha = 0$ the trajectory would be unchanged, but for intermediate values the resulting control step is not a simple scaled version of the full step, due to the presence of feedback. As advocated in [4], we use the expected total-cost reduction in the line-search procedure, with two differences. The first is that we use the improved formula (12a) rather than (8a) for the expected reduction:

$$\begin{aligned}\Delta J(\alpha) = \alpha \sum_{i=1}^{N-1} \mathbf{k}(i)^{\mathsf{T}} Q_{\mathbf{u}}(i) + \\ \frac{\alpha^2}{2} \sum_{i=1}^{N-1} \mathbf{k}(i)^{\mathsf{T}} Q_{\mathbf{uu}}(i)\mathbf{k}(i).\end{aligned} \tag{14}$$

By saving the linear and quadratic terms separately, we obtain a quadratic model of the expected reduction as a function of α. We then compare the actual and expected reductions

$$z = [J(\mathbf{u}_{1..N-1}) - J(\hat{\mathbf{u}}_{1..N-1})]/\Delta J(\alpha),$$

and accept the iteration only if the actual reduction is bigger than some minimal threshold.

$$0 < c_1 < z. \tag{15}$$

This is similar to the classic Armijo condition in optimization, but uses the more accurate quadratic reduction model (14).

D. Algorithm Summary

A single iteration of the optimizer described here is composed of 3 steps:

1. Derivatives: Given a nominal $(\mathbf{x}, \mathbf{u}, i)$ sequence, compute the derivatives of ℓ and $\mathbf{f}$ in the RHS of Eq. (6). This can be done in parallel for all i.

2. Backward pass: Iterate Eqs. (6, 11, 12) for decreasing $i = N-1, \ldots 1$. If a non-PD $\widetilde{Q}_{\mathbf{uu}}$ is encountered, increase μ and restart the backward pass. If successful and μ was not increased, decrease μ.

3. Forward pass: Set $\alpha = 1$. Iterate (13) and (9c) to compute a new nominal sequence. If the integration diverged or condition (15) was not met, decrease α and restart the forward pass.

E. Parameter Schedules

The fast and accurate modification of the regularization parameter μ in step **2** turns out to be quite important due to three conflicting requirements. If we are near the minimum we would like μ to quickly go to zero to enjoy the quadratic convergence. If the approximation diverges (a non-PD $\widetilde{Q}_{\mathbf{uu}}$), we would like it to increase very rapidly, since the minimum value of μ which prevents divergence is often very large ($\sim 10^6$). Finally, if we are in a regime where some $\mu > 0$ is required, we would like to accurately tweak it to be as close as possible to the minimum value (though not smaller).

Our solution is to use a log-quadratic modification schedule. Defining some minimal value $\mu_{\min}$ (we use $\mu_{\min} = 10^{-6}$) and a minimal modification factor Δ_0 (we use $\Delta_0 = 2$), we adjust μ as follows:

$$\begin{aligned}
&\text{increase } \mu\text{:}\\
&\qquad \Delta \leftarrow \max(\Delta_0, \Delta \cdot \Delta_0)\\
&\qquad \mu \leftarrow \max(\mu_{\min}, \mu \cdot \Delta)\\
&\text{decrease } \mu\text{:}\\
&\qquad \Delta \leftarrow \min(\tfrac{1}{\Delta_0}, \tfrac{\Delta}{\Delta_0})\\
&\qquad \mu \leftarrow \begin{cases} \mu \cdot \Delta & \text{if } \mu \cdot \Delta > \mu_{\min},\\ 0 & \text{if } \mu \cdot \Delta < \mu_{\min}. \end{cases}
\end{aligned}$$

Whenever μ increases or decreases consecutively across iterations, the size of the change grows geometrically. If increase and decrease alternate, the change remains small.

The decrease of the line search parameter α is not as sensitive, and a simple halving backtrack $\alpha \leftarrow \alpha/2$ was sufficient. For the minimal relative cost decrease of (15), we used $c_1 = 0.5$.

V. COMPUTATIONAL CONSIDERATIONS

In practice, the computational effort for step **1** is the largest, taking $\sim\%90$ of CPU time, never below $\%80$ in our experiments. Step **2** constituted $\sim\%15$ of the time. Step **3** was smallest, at an order of $\%1$. Specific numbers are given below in sec. VI.

To understand this distribution of computational load, consider the dynamics. For the mechanical systems considered here, the Euler-discretized equations of motion are

$$\begin{aligned}
\mathbf{q}(t+h) &= \mathbf{q}(t) + h\dot{\mathbf{q}}(t)\\
\dot{\mathbf{q}}(t+h) &= \dot{\mathbf{q}}(t) + hM^{-1}(\mathbf{r} + B\mathbf{u}),
\end{aligned}$$

where h is the timestep, $\mathbf{q}$ is the configuration, $M = M(\mathbf{q})$ is the mass-matrix, $\mathbf{r} = \mathbf{r}(\mathbf{q}, \dot{\mathbf{q}})$ is the vector of total Coriolis, centripetal and other intrinsic forces, $\mathbf{u}$ is the control, and B is the $n \times m$ matrix determining which degrees of freedom are actuated (with $m < n$ for underactuation). The computational effort for the dynamics lies mainly in the construction and factorization of $M(\mathbf{q})$, and the computation of $\mathbf{r}(\mathbf{q}, \dot{\mathbf{q}})$. In order to compute derivatives, we used simple finite-differences on the entire dynamics computation. This proved to be adequately accurate for a wide range of finite-difference parameters. Analytical computation of the derivatives, while more accurate, would not be cheaper than finite-differencing, and can be significantly more expensive if the multiplication-order of chain-rule factors is not chosen judiciously.

Let a single factorization and back-substitution of M be a rough proxy for the complexity of a single call to the dynamics engine, at $\sim O(n^3)$. First derivatives are therefore $O(Nn^4)$, and second derivatives are $O(Nn^5)$. In contrast steps **2** and **3** both have a complexity of $O(Nn^3)$, step **2** from the multiplications in (6) (with a high constant factor), step **3** from N calls to the dynamics.

To see if we can reduce the burden of computing second derivatives, consider the last terms in the RHS of (6c, 6d, 6e). Since mechanical systems are control-affine, $\mathbf{f_{uu}} = 0$ identically. Additionally, $\mathbf{f_u} = hM^{-1}B$ is essentially a by-product of the computation (having factorized M), so $\mathbf{f_{ux}}$ is a by-product of computing $\mathbf{f_x}$. This leaves us with $\mathbf{f_{xx}}$, which is the largest object (an n^3 tensor) and the most expensive to compute. The iLQG algorithm described in [11] is a DDP variant that assumed $\mathbf{f_{xx}} = 0$. Here we make the following observation: if first derivatives $\mathbf{f_x}$ are computed with central differences, then the computation of the diagonal of $\mathbf{f_{xx}}$ is free. This reduces the complexity of step **1** to $O(Nn^4)$, still the most expensive, but less so by a significant factor. In the experiments below we explore the tradeoff of using full versus diagonal $\mathbf{f_{xx}}$.

A significant speedup can be achieved by parallelizing step **1** across several cores on a modern multi-core processor. We used a new physics simulator created expressly for this purpose, as described in the Appendix.

VI. EXPERIMENTS

A. Swimmer Models

The models consisted of several elongated masses connected by joints, in a simulated fluid medium. We had both planar and 3-dimensional models. The interaction with the fluid was modeled by positing high linear drag κ_n in the direction normal to the long axis of the masses, and a smaller drag $\kappa_t < \kappa_n$ in the tangential direction.

A planar swimmer model with k links has 2 cartesian and k angular degrees of freedom for a state dimension $n = 4 + 2k$, of which $m = k - 1$ are actuated. This is true for any tree-like topology (we cannot represent topologies with loops).

A 3D k-link model has 3 cartesian and $3k$ rotational degrees of freedom. In principle, we could have ignored the rotational degrees of freedom of the joints in the axial direction, using two hinges or even one hinge at each joint. However, using single aligned hinges would essentially embed a planar swimmer in 3D space, and multiple unaligned hinges can reach singular points (gimbal lock). We therefore chose to represent all joints as normalized quaternions (ball joints), which have no singular points. This means that each joint is described by 7 numbers, 4 for the normalized quaternion, and 3 for the angular velocities, for a total state dimension of $n = 6 + 7k$ for a 3D k-link model. The actuated states in this case are of dimension $m = 3(k-1)$.

B. Cost Function

The cost we used was

$$\begin{aligned}
\ell(\mathbf{x}, \mathbf{u}, i) =& c_x \log(\cosh(\|\mathbf{x}_c - \mathbf{x}_t\|))\\
&+ c_o \sum_j \mathcal{N}(\mathbf{x}_c; \mathbf{x}_j(i), I)\\
&+ c_u \|\mathbf{u}\|^2
\end{aligned}$$

The first term penalizes the distance between the Cartesian components of the state $\mathbf{x}_c$ and the position of the target $\mathbf{x}_t$. The $\log(\cosh())$ function looks like a smoothed absolute-value

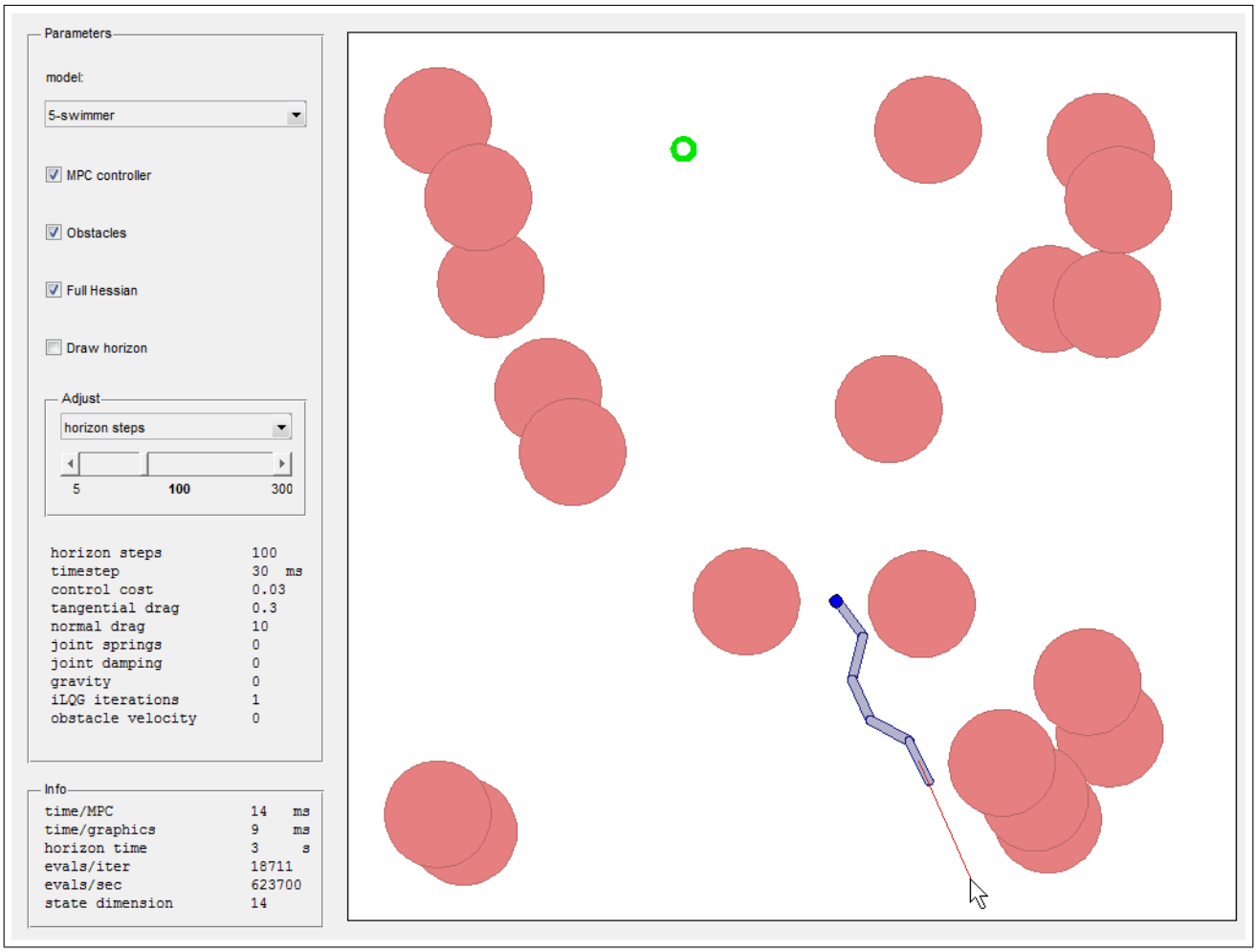

Fig. 1. Screen capture of the user interacting with a swimmer. The nose point $\mathbf{x}_c$ is in blue, the target $\mathbf{x}_t$ is the green circle, and the pink discs are the obstacles. The user is pulling on the tail link with a linear spring (red line). A movie of the interaction is available at http://goo.gl/ta3aL

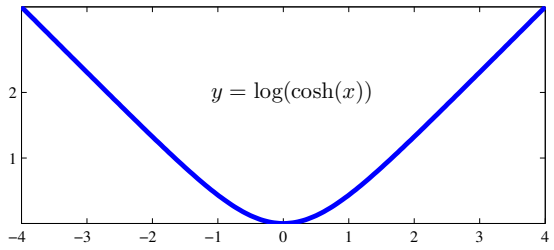

Fig. 2. The functional form of the state-cost component.

function, and was simply a convenient way of smoothing the cost at the singular point $\mathbf{x}_c = \mathbf{x}_t$. The second term describes a set of obstacles, represented by unit gaussians. Note that the means are time dependent, so the the gaussians can move along predefined trajectories, and the controller will be able to react predictively to their motion. Finally, the last term is the standard quadratic control-cost.

C. Graphical User Interface

One of benefits of MPC is that because there is no offline component, all the parameters can be changed in real time. To that end, we designed a rich graphical interface to allow a user to interact with the controller (Figure 1). The user can apply forces to the masses and move the target with the mouse (with the left and right buttons, respectively), and change a variety of parameters. These are:

- The number of timesteps N.
- The length of each timestep dt.
- The control cost c_u.
- The normal and tangential drag κ_n and κ_t.
- Springs and dampers at the joints.
- Gravity.
- The number of optimization iterations at each timestep.
- The velocity of the obstacles.

Additionally, the user can choose to use only the diagonal of the Hessian, as described in Sec. V. In the lower left part of the window, the simulation displays the computational time of every iteration, allowing the user to quantify the performance

TABLE I
SIMULATOR SPEED

n	Evals / Sec
10	2,580,000
14	1,650,000
16	1,470,000
18	1,140,000
27	530,000
34	380,000

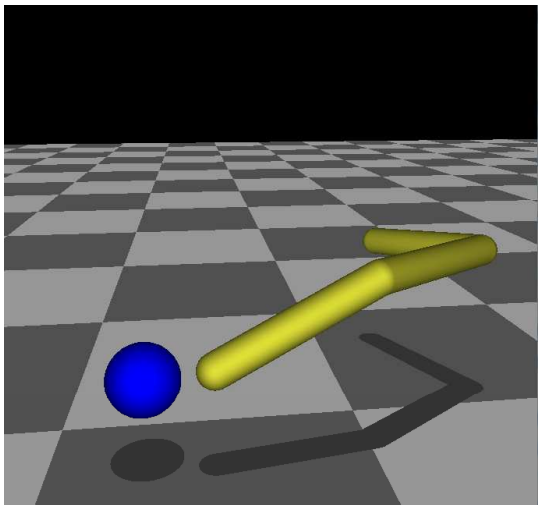

Fig. 3. A snapshot of a 3D swimmer in action while the user is moving the target in real-time.

implications of different parameter settings.

D. Results

As shown in the movie available in http://goo.gl/ta3aL, the MPC controller generates the complex behavior in real-time, including: steady-state swimming, coasting, braking, and various contortional maneuvers.

VII. ACKNOWLEDGMENTS

This work was supported by the US National Science Foundation.

REFERENCES

[1] P. Abbeel, A. Coates, M. Quigley, and A. Y Ng. An application of reinforcement learning to aerobatic helicopter flight. In *Advances in Neural Information Processing Systems 19: Proceedings of the 2006 Conference*, page 1, 2007.

[2] M. Diehl, H. G Bock, J. P Schloder, R. Findeisen, Z. Nagy, and F. Allgower. Real-time optimization and nonlinear model predictive control of processes governed by differential-algebraic equations. *Journal of Process Control*, 12(4):577, 2002.

[3] M. Diehl, H. Ferreau, and N. Haverbeke. Efficient numerical methods for nonlinear mpc and moving horizon estimation. *Nonlinear Model Predictive Control*, page 391, 2009.

[4] D. H. Jacobson and D. Q. Mayne. *Differential Dynamic Programming*. Elsevier, 1970.

[5] L. Z Liao and C. A Shoemaker. Convergence in unconstrained discrete-time differential dynamic programming. *IEEE Transactions on Automatic Control*, 36(6):692, 1991.

[6] L. Z. Liao and C. A Shoemaker. Advantages of differential dynamic programming over newton's method for discrete-time optimal control problems. *Cornell University, Ithaca, NY*, 1992.

[7] D. Q. Mayne. A second-order gradient method of optimizing non-linear discrete time systems. *Int J Control*, 3:85–95, 1966.

[8] J. Morimoto, G. Zeglin, and C. G Atkeson. Minimax differential dynamic programming: Application to a biped walking robot. In *Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, volume 2, page 1927–1932, 2003. ISBN 0780378601.

[9] L. S Pontryagin, V. G. Boltyanskii, R. V. Gamkrelidze, and E. F. Mishchenko. *The mathematical theory of optimal processes*. Interscience New York, 1962.

[10] K. Sims. Evolving virtual creatures. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, page 15–22, 1994.

[11] E. Todorov and Weiwei Li. A generalized iterative LQG method for locally-optimal feedback control of constrained nonlinear stochastic systems. In *Proceedings of the 2005, American Control Conference, 2005.*, pages 300–306, Portland, OR, USA, 2005. doi: 10.1109/ACC.2005.1469949.

APPENDIX: THE MuJoCo ENGINE

The simulations described in this paper were carried out using a new physics engine; the name stands for Multi Joint dynamics with Contact. This engine will soon be made publicly available and will be free for non-profit research. Table I shows computation speed for different swimmer topologies. A full description of this physics engine is given in a companion paper (submission ID: 50) Below is a brief description taken from the (current version of) the user's manual:

MuJoCo is a platform-independent physics simulator tailored to control applications. Multi-joint dynamics are represented in joint coordinates and computed via recursive algorithms. The computation is $O(n^3)$ because the inverse inertia matrix is needed (to compute contact responses), however due to tree-induced sparsity, performance is comparable to $O(n)$ algorithms in typical usage scenarios (e.g. simulating a humanoid). Geometry is modeled using a small library of smooth shapes allowing fast and accurate collision detection. Contact responses are computed by efficient new algorithms (which we have developed) that appear to be faster and more accurate than LCP-based methods, and are suitable for numerical optimization. Models are specified using either a high-level C++ API or an XML file. A built-in compiler

transforms the user model into an optimized data structure used for runtime computation. This data structure contains a scratchpad where all routines write their output. In this way all intermediate results are accessible to the user, making it easy to add functionality. The user can modify all real-valued model parameters in runtime without recompiling. To facilitate optimal control applications, MuJoCo provides routines for computing the cost of a given trajectory as well as the gradient and an approximate Hessian, and a built-in trajectory optimizer. The latter can exploit parallelism via multi-threading. MuJoCo can be used either as a library linked to a user program, or via a Matlab interface. A Windows-specific utility for interactive 3D rendering is also provided.