Efficient Robust Policy Optimization

Christopher G. Atkeson

Abstract— **We provide efficient algorithms to calculate first and second order gradients of the cost of a control law with respect to its parameters, to speed up policy optimization. We achieve robustness by simultaneously designing one control law for multiple models with potentially different model structures, which represent model uncertainty and unmodeled dynamics. Providing explicit examples of possible unmodeled dynamics during the control design process is easier for the designer and is more effective than providing simulated perturbations to increase robustness, as is currently done in machine learning. Our approach supports the design of deterministic nonlinear and time varying controllers for both deterministic and stochastic nonlinear and time varying systems, including policies with internal state such as observers or other state estimators. We highlight the benefit of control laws made up of collections of simple policies where only one component policy is active at a time. Controller optimization and learning is particularly fast and effective in this situation because derivatives are decoupled.**

I. INTRODUCTION

Parametric modeling error and unmodeled dynamics are a problem for model-based control law (policy) design and learning algorithms, such as dynamic programming and most forms of trajectory optimization. A heuristic approach to robust control law design popular in machine learning is to optimize a policy by evaluating its performance in simulation on a distribution of possible models [1]–[12]. This paper describes how to make this design approach more efficient by propagating analytic gradients backward along simulated trajectories.

Policy optimization is closely related to optimal output feedback design. Our contribution to the output feedback controller optimization community is efficient gradient methods to design time varying and nonlinear optimal output feedback as well as feedforward input for time varying and nonlinear plants. We present a robust control design approach that handles multiple models with different structures (for example, number of state variables). Our method also handles policies with internal state, allowing the simultaneous design of robust control laws and state estimators. A contribution to the machine learning community is to emphasize that uncorrelated additive or multiplicative noise is a poor proxy for unmodeled dynamics. The challenging aspect of unmodeled dynamics is that small errors are correlated across time, leading to large effects. We also emphasize the benefits of analytic first and second order gradients, and the benefits of Newton (second order) algorithms for model-based policy optimization. We highlight the benefit of control laws made up of collections of simple policies where only one simple policy is active at a time.

C. G. Atkeson is with the Robotics Institute, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA, USA cga@cmu.edu

This paper focuses on designing control laws for systems with discrete time dynamics, as the algorithms are similar for systems with continuous time dynamics, and our robots typically learn discrete time models. We have found many of the tasks we want to do are largely deterministic rather than stochastic, so we focus our discussion here on how to design deterministic nonlinear and potentially time varying discrete time control laws. For cases where the multiple models all have the same state vector, the common policy is $\mathbf{u} = \boldsymbol{\pi}(\mathbf{x}, \mathbf{p})$, where $\mathbf{u}$ is a vector of controls of dimensionality $N_{\mathbf{u}}$, $\mathbf{x}$ is the state vector of the controlled system (dimensionality $N_{\mathbf{x}}$), and $\mathbf{p}$ is a policy parameter vector of dimensionality $N_{\mathbf{p}}$ that describes the policy $\boldsymbol{\pi}()$. This approach attempts to handle unmodeled dynamics including time delays, bandwidth or power limits on actuation, unmodeled vibrational modes, and non-collocated sensing found in lightweight robot arms such as inflatable arms [13], robots with series elastic actuation, satellites with booms or large solar panels, and large space structures.

A Simple Example

We present our method applied to a simple double integrator example, with second order unmodeled dynamics or an unknown delay. We then compare our method to a perturbation-based robust control design approach. Consider a nominal linear plant which is a double integrator (mass = 1) sampled at 1kHz. The state vector $\mathbf{x}$ consists of the position p and velocity v. In this example the feedback control law has the structure $\mathbf{u} = \mathbf{K}\mathbf{x} = k_p p + k_v v$. An optimal Linear Quadratic Regulator (LQR) is designed for the nominal double integrator plant with a one step cost function of $L(\mathbf{x}, \mathbf{u}) = 0.5(\mathbf{x}^{\mathrm{T}}\mathbf{Q}\mathbf{x} + \mathbf{u}^{\mathrm{T}}\mathbf{R}\mathbf{u})$. In this example $\mathbf{Q} = [1000\ 0; 0\ 1]$ and $\mathbf{R} = [0.001]$ resulting in optimal feedback gains of $\mathbf{K} = [-973\ -54]$.

The true plant is the nominal plant with the following unmodeled dynamics: a second order low pass filter is added on the input with a cutoff of 10 Hz, which simulates actuator dynamics. The transfer function for the unmodeled dynamics is $\omega^2/(s^2 + 2\gamma\omega s + \omega^2)$, with a damping ratio $\gamma = 1$ and a natural frequency $\omega = 20\pi$. There is no resonant peak and the unmodeled dynamics acts as a well behaved low pass filter. However, the unmodeled dynamics drive the true plant unstable when the feedback gains designed using the nominal plant model are used. Figure 1 shows simulations of these conditions: the blue dot-dashed line is the nominal plant with the original gains $[-973\ -54]$, and the black dotted line shows the true plant with the original gains, which is unstable.

One way to design a robust control law is to optimize the parameters of a control law (in this case the position

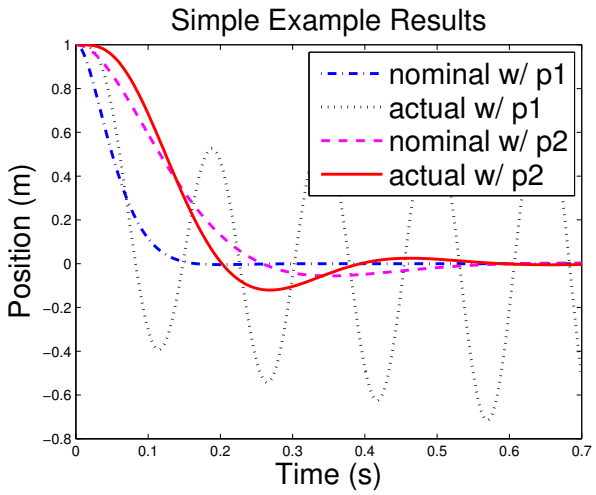

Fig. 1. Simulations of the simple example. Policy parameters are: $p1 = [-973\ -54]$, $p2 = [-148\ -16]$

and velocity feedback gains) by evaluating them on several different models. The control law is simulated for a fixed duration D on each of M models for S initial conditions, and the cost of each trajectory, $V^m(\mathbf{x}_s, \mathbf{p})$, is summed for the overall optimization criterion, using the above one step cost function $L(\mathbf{x}, \mathbf{u})$: $C = \sum_{m=1}^{M} \sum_{s=1}^{S} w(m,s) V^m(\mathbf{x}_s, \mathbf{p})$, where $w(m,s)$ is a weight on each trajectory. We will suppress the m superscript on V to simplify our results. We assume that each trajectory is created using the appropriate model and uses the appropriate model dynamics to calculate derivatives of V in what follows. First and second order gradients are summed using the same weights $w(m,s)$ as were used on the trajectories.

Optimizing $\mathbf{u} = \mathbf{K}\mathbf{x}$ for the nominal model and the nominal model with an added input filter with $\omega = 10\pi$ and $\gamma = 0.5$, with initial conditions $(1,0)$, results in feedback gains of $[-148\ -16]$. These gains are also stable for the true plant $(\omega = 20\pi, \gamma = 1)$. Figure 1 shows simulations of these conditions: The magenta dashed line shows the nominal plant with the gains optimized for multiple models, and the red solid line shows the true plant with the same gains. The multiple model gains are less aggressive than the original gains, and the true plant is stable and reasonably well damped.

A model with the same model structure as the true plant does not have to be included in the set of models used in policy optimization. Optimizing using the nominal double integrator model and the nominal model with an input delay of 50 milliseconds results in optimized gains of $[-141\ -18]$, which provide about the same performance on the true plant as the previous optimized gains. In addition, the new gains are stable for double integrator plants with delays up to 61 milliseconds, while the original gains of $[-973\ -54]$ are stable only for delays up to 22 milliseconds. We note that the nominal double integrator model, the nominal model with an input filter, and the nominal model with a delay all have different model structures (number of state variables for example), which a multiple model policy optimization approach should handle.

We compare our approach to the heuristic used in reinforcement learning of adding simulated perturbations to make the policy more robust. We use the method of common random numbers [14] (which has been reinvented many times and is also known as correlated sampling, matched pairs, matched sampling, and Pegasus [15]) to optimize the policy. An array of random numbers is created, and that same array is used to perturb each simulation of the nominal system, typically by adding noise to the plant input $\mathbf{u}$, while optimizing policy parameters. On the simple example, we found that the added noise needed to be quite large (uniformly distributed with limits $\pm$1000N on each time step, 100 times the weight of the mass assuming a mass of 1) for the generated controller to work reliably on the true plant with the input filter with a cutoff of 10 Hz. Additionally, there was only a narrow window of noise levels that worked reliably, and higher and lower levels of noise produced unstable controllers quite often. Table I shows how adding input noise to the input of the double integrator during optimization affects robustness to delays in the double integrator, as measured by the maximum delay the controller can stabilize. Again, $\pm$1000N uniformly distributed noise added to $\mathbf{u}$ on each time step provides the most robustness. However, this maximum robustness is less than that provided by optimizing with multiple models. We have found in general that added noise is not a reliable proxy for unmodeled dynamics. The challenging aspect of unmodeled dynamics is that small errors are correlated across time, leading to large effects.

Noise level (N)	Maximum stable delay (msec)
0	22
1	22
10	22
100	22
1000	30
10000	19
100000	16

TABLE I

INPUT NOISE LEVEL VS. ROBUSTNESS.

II. RELATED WORK

See [16] for a more extensive discussion of related work. The term multiple models means different things in different fields. We use it to mean alternative plants that could exist. In machine learning it often means multiple model structures that are selected or blended to fit data. In control theory it has been used both for alternative global models and local models that divide up the state space [17]. In Multiple Model Adaptive Control and Multiple Model Adaptive Estimation (MMAC and MMAE) instead of computing one policy based on multiple models as is done in this paper, a policy is computed for each possible model. An adaptive algorithm learns to select or combine the individual policies.

Our gradient methods are analogous to *reverse mode* and *reverse propagation of gradients* in automatic/algorithmic differentiation, and the *adjoint method* [18], [19]. There is a strong relationship between this work and Differential Dynamic Programming (DDP), which propagates value function information backward in time along a trajectory, and chooses

optimal actions and feedback gains at each time step [20], [21]. The optimization of global parameters and the general form of the value function update equations in [21] were an inspiration for this work. Our work suggests alternative forms of DDP, such as optimizing a trajectory-based policy discussed in Section V and a version which uses multiple models simultaneously.

Output feedback optimization computes the optimal control law for linear models when the structure of the control law is fixed, or when full state feedback is not available, and an observer or state estimator is not used [1]. Although linear matrix inequality (LMI) techniques are useful in addressing robust control design, we note that it is difficult to apply LMI or polytopic model-based optimal output feedback techniques to multiple models with different model structures since it is not clear how to interpolate between these models [1]. One can embed the multiple models in a much more complex single model so that structural differences become parametric differences, but that greatly complicates the design process. For linear systems one can interpolate models in the frequency domain, but it is not clear how to generalize frequency domain interpolation to nonlinear models with different structures. Varga showed how to apply multiple models to output feedback controller optimization where all models have the same state vector [1].

Policy optimization (also known as policy search/refinement/improvement/gradient) is of great interest in reinforcement learning (RL) [12]. Typically a stochastic policy is used to provide "exploration" or from our point of view perform numeric differentiation to find the dependence of the trajectory cost on the policy parameters. Gradient learning algorithms such as backpropagation applied to a lattice network model of the trajectory-based computations or backpropagation through time applied to a recurrent network model result in similar gradient equations to this work [22]–[25]. One area of reinforcement learning that is also closely related to this work is that of adaptive critics [26]–[31]. Function approximation is used to represent both a policy $\boldsymbol{\pi}(\mathbf{x},\mathbf{p})$ as we do and a parametrized global value function $\hat{V}(\mathbf{x},\boldsymbol{\omega})$. Gradient descent and other optimization techniques are used to learn $\mathbf{p}$ and $\boldsymbol{\omega}$. Our approach tries to avoid making a commitment to a global structure and parametrization for $V(\mathbf{x})$ or $V_{\mathbf{x}}(\mathbf{x})$ by using local quadratic models for $V(\mathbf{x})$ (or equivalently local linear models for $V_{\mathbf{x}}(\mathbf{x})$). Lewis and Vrabie developed first order analytic gradient equations for the special case when the policy is linearly parametrized [30]. Kolter developed a first order analytic gradient that propagates derivatives forward in time for deterministic policy optimization, which, because it does not take advantage of value functions, is in general less efficient than our approach [12], [16]. Kolter's approach is analogous to *forward mode* in automatic/algorithmic differentiation [18].

III. ANALYTIC GRADIENTS

A wide variety of optimization algorithms can be used to optimize the policy parameters $\mathbf{p}$. The goal of this paper is to provide efficient algorithms to calculate first and second order gradients of the total trajectory cost of a control law with respect to its parameters, to speed up policy optimization. We describe how to propagate analytic gradients backward along simulated trajectories. The gradient algorithms presented are intended to be used in a controller design process, so we assume one step cost function and the policy structure are known. Nominal models for the process are known or learned. In this section we will consider policy optimization problems using multiple discrete time models where there is no discounting, full state feedback is available, all the models use the same state vector, the policies are static, and there is no opponent. Later sections and [16] discuss extensions to the basic approach. We will show how to calculate the first and second order cost gradients for a single trajectory. The total derivatives for a set of models and trajectories are the sum of the derivatives for each trajectory.

A. First Order Gradient

A first order gradient descent algorithm updates the policy parameters in the following way: $\Delta\mathbf{p} = -\varepsilon\sum_{m=1}^{M}\sum_{s=1}^{S} w(m,s)V_{\mathbf{p}}^{\mathrm{T}}(\mathbf{x}_s,\mathbf{p})$ where $\Delta\mathbf{p}$ is the update, ε is a step size, and $V_{\mathbf{p}} = \partial V/\partial\mathbf{p}$. $V_{\mathbf{p}}$ and other derivatives of scalars are row vectors. We will use a finite horizon to a fixed point in time to evaluate the policy. In this case the Bellman Equation (principle of optimality [20]) becomes: $V^k(\mathbf{x},\mathbf{p}) = L(\mathbf{x},\boldsymbol{\pi}(\mathbf{x},\mathbf{p})) + V^{k+1}(\mathbf{F}(\mathbf{x},\boldsymbol{\pi}(\mathbf{x},\mathbf{p})),\mathbf{p})$ where $L(\mathbf{x}_i,\mathbf{u}_i)$ is the known one step cost function, $\mathbf{x}_{i+1} = \mathbf{F}(\mathbf{x}_i,\mathbf{u}_i)$ are the system dynamics equations appropriate for each model, and $V^k(\mathbf{x},\mathbf{p}) = \phi(\mathbf{x}_D) + \sum_{i=k}^{D-1} L(\mathbf{x}_i,\mathbf{u}_i)$ is the cost of the remaining trajectory generated by starting at $\mathbf{x}_k$ and using the policy $\mathbf{u}_i = \boldsymbol{\pi}(\mathbf{x}_i,\mathbf{p})$. $\phi(\mathbf{x})$ is a terminal cost function evaluated at the end of the trajectory. We note that the one step cost function $L()$ and terminal cost function $\phi()$ may depend on the model m, and also the initial state (s index). We will suppress this dependence in what follows, but it is straightforward to include it. The derivative $V_{\mathbf{p}}$ is $V_{\mathbf{p}}^0$, and we will use the notation $V_{\mathbf{p}}$ and $V_{\mathbf{p}}^0$ interchangeably. i and k are temporal indices and can appear as either subscripts and superscripts as needed for readability.

To calculate a first order gradient we will approximate the dynamics, one step cost, policy, and value function $V()$ with first order Taylor series approximations. For example, $\mathbf{F}(\mathbf{x},\mathbf{u}) = \bar{\mathbf{F}} + \mathbf{F}_{\mathbf{x}}\Delta\mathbf{x} + \mathbf{F}_{\mathbf{u}}\Delta\mathbf{u}$ where we follow the conventions of [21] in that $\mathbf{x}$, $\mathbf{u}$, and $\mathbf{p}$ subscripts indicate partial derivatives evaluated with the appropriate arguments at that time point along the trajectory. Derivatives of scalars ($L_{\mathbf{x}}$, $L_{\mathbf{u}}$, $V_{\mathbf{x}}$, and $V_{\mathbf{p}}$) are row vectors. Derivatives of vectors are matrices whose rows are the derivatives of the components of the original vector. $\mathbf{F}_{\mathbf{x}}$ is an $N_{\mathbf{x}} \times N_{\mathbf{x}}$ matrix, $\mathbf{F}_{\mathbf{u}}$ is $N_{\mathbf{x}} \times N_{\mathbf{u}}$, $\boldsymbol{\pi}_{\mathbf{x}}$ is $N_{\mathbf{u}} \times N_{\mathbf{x}}$, and $\boldsymbol{\pi}_{\mathbf{p}}$ is $N_{\mathbf{u}} \times N_{\mathbf{p}}$. In this case, the derivatives of the Bellman Equation are:

$$V_{\mathbf{x}}^k = L_{\mathbf{x}} + L_{\mathbf{u}}\boldsymbol{\pi}_{\mathbf{x}} + V_{\mathbf{x}}^{k+1}(\mathbf{F}_{\mathbf{x}} + \mathbf{F}_{\mathbf{u}}\boldsymbol{\pi}_{\mathbf{x}}) \tag{1}$$

and

$$V_{\mathbf{p}}^k = (L_{\mathbf{u}} + V_{\mathbf{x}}^{k+1}\mathbf{F}_{\mathbf{u}})\boldsymbol{\pi}_{\mathbf{p}} + V_{\mathbf{p}}^{k+1} \tag{2}$$

We are suppressing the k superscripts on the right hand sides of these equations since every symbol not indexed by $k+1$ is indexed by k. $V_{\mathbf{p}}^0$ is calculated by using these equations to propagate V and its derivatives backward in time along the trajectory. We are making extensive use of the chain rule. For a terminal cost function $\phi(\mathbf{x})$, $V_{\mathbf{x}}^D = \phi_{\mathbf{x}}$. Since $\phi()$ is independent of the policy parameters, $V_{\mathbf{p}}^D = 0$. If there is no terminal cost function, $V_{\mathbf{x}}^D = 0$.

Equations 1 and 2 can be used in many ways in optimization. Backward passes to calculate $\Delta\mathbf{p}$ can alternate with forward passes that generate new trajectories by using the new policy and integrating the appropriate dynamics forward in time for each model. Trajectory segments can be generated, as in multiple shooting. Trajectories can be represented parametrically and an optimization procedure can be used to make the trajectories consistent with the new policy and appropriate dynamics, as in collocation. A pure gradient method will potentially get stuck in bad local minima. [16] discusses ways to avoid such minima and also perform global optimization of a policy.

B. Second Order Gradient

Approximate second order gradients (Hessians) are useful for remedying the deficiencies of first order gradient descent. A second order gradient descent algorithm updates the policy parameters in the following way: $\Delta\mathbf{p} = -\left(\sum_{m=1}^{M}\sum_{s=1}^{S} w(m,s)V_{\mathbf{pp}}(\mathbf{x}_s,\mathbf{p})\right)^{-1}\sum_{m=1}^{M}\sum_{s=1}^{S} w(m,s)V_{\mathbf{p}}^{\mathrm{T}}(\mathbf{x}_s,\mathbf{p})$ where $V_{\mathbf{pp}} = \partial^2 V/\partial\mathbf{p}^2$. To calculate the second order gradient we will approximate the dynamics, one step cost, policy, and value function $V()$ with second order Taylor series approximations. For example: $\mathbf{F}(\mathbf{x},\mathbf{u}) = \bar{\mathbf{F}} + \mathbf{F}_{\mathbf{x}}\Delta\mathbf{x} + \mathbf{F}_{\mathbf{u}}\Delta\mathbf{u} + 0.5\Delta\mathbf{x}^{\mathrm{T}}\mathbf{F}_{\mathbf{xx}}\Delta\mathbf{x} + \Delta\mathbf{x}^{\mathrm{T}}\mathbf{F}_{\mathbf{xu}}\Delta\mathbf{u} + 0.5\Delta\mathbf{u}^{\mathrm{T}}\mathbf{F}_{\mathbf{uu}}\Delta\mathbf{u}$. We follow the conventions of [21] in that the second derivatives of vectors ($\mathbf{F}_{\mathbf{xx}}$, $\mathbf{F}_{\mathbf{xu}}$, $\mathbf{F}_{\mathbf{uu}}$, $\boldsymbol{\pi}_{\mathbf{xx}}$, ...) are third-order tensors. A quadratic form including the second derivative of a vector such as $\Delta\mathbf{x}^{\mathrm{T}}\mathbf{F}_{\mathbf{xu}}\Delta\mathbf{u}$ is a vector whose jth component is the quadratic form using the second derivative of the jth component of the original vector: $\Delta\mathbf{x}^{\mathrm{T}}\mathbf{F}_{\mathbf{xu}}^j\Delta\mathbf{u}$. Another useful formula is the product of a row vector $\mathbf{v}$, a matrix $\mathbf{A}$, the third order tensor ($\boldsymbol{\pi}_{\mathbf{xp}}$ for example), and another matrix $\mathbf{B}$ which is: $\mathbf{v}(\mathbf{A}\boldsymbol{\pi}_{\mathbf{xp}}\mathbf{B}) = \sum_j \mathbf{v}^j(\mathbf{A}\boldsymbol{\pi}_{\mathbf{xp}}^j\mathbf{B})$ We note that cross derivatives are independent of the order in which the derivatives are taken, so $L_{\mathbf{ux}} = L_{\mathbf{xu}}^{\mathrm{T}}$, $V_{\mathbf{px}} = V_{\mathbf{xp}}^{\mathrm{T}}$, $\mathbf{F}_{\mathbf{ux}}^j = (\mathbf{F}_{\mathbf{xu}}^j)^{\mathrm{T}}$, and $\boldsymbol{\pi}_{\mathbf{px}}^j = (\boldsymbol{\pi}_{\mathbf{xp}}^j)^{\mathrm{T}}$.

This results in the following recursion in time for the second order derivatives of V:

$$\begin{aligned} V_{\mathbf{xx}}^k &= L_{\mathbf{xx}} + L_{\mathbf{xu}}\boldsymbol{\pi}_{\mathbf{x}} + (L_{\mathbf{xu}}\boldsymbol{\pi}_{\mathbf{x}})^{\mathrm{T}} + \boldsymbol{\pi}_{\mathbf{x}}^{\mathrm{T}}L_{\mathbf{uu}}\boldsymbol{\pi}_{\mathbf{x}} + L_{\mathbf{u}}\boldsymbol{\pi}_{\mathbf{xx}} \\ &\quad + (\mathbf{F}_{\mathbf{x}} + \mathbf{F}_{\mathbf{u}}\boldsymbol{\pi}_{\mathbf{x}})^{\mathrm{T}}V_{\mathbf{xx}}^{k+1}(\mathbf{F}_{\mathbf{x}} + \mathbf{F}_{\mathbf{u}}\boldsymbol{\pi}_{\mathbf{x}}) \\ &\quad + V_{\mathbf{x}}^{k+1}(\mathbf{F}_{\mathbf{xx}} + \mathbf{F}_{\mathbf{xu}}\boldsymbol{\pi}_{\mathbf{x}} + (\mathbf{F}_{\mathbf{xu}}\boldsymbol{\pi}_{\mathbf{x}})^{\mathrm{T}} + \boldsymbol{\pi}_{\mathbf{x}}^{\mathrm{T}}\mathbf{F}_{\mathbf{uu}}\boldsymbol{\pi}_{\mathbf{x}} + \mathbf{F}_{\mathbf{u}}\boldsymbol{\pi}_{\mathbf{xx}}) \end{aligned} \quad (3)$$

$$\begin{aligned} V_{\mathbf{xp}}^k &= L_{\mathbf{xu}}\boldsymbol{\pi}_{\mathbf{p}} + \boldsymbol{\pi}_{\mathbf{x}}^{\mathrm{T}}L_{\mathbf{uu}}\boldsymbol{\pi}_{\mathbf{p}} + L_{\mathbf{u}}\boldsymbol{\pi}_{\mathbf{xp}} \\ &\quad + (\mathbf{F}_{\mathbf{x}} + \mathbf{F}_{\mathbf{u}}\boldsymbol{\pi}_{\mathbf{x}})^{\mathrm{T}}V_{\mathbf{xx}}^{k+1}\mathbf{F}_{\mathbf{u}}\boldsymbol{\pi}_{\mathbf{p}} + (\mathbf{F}_{\mathbf{x}} + \mathbf{F}_{\mathbf{u}}\boldsymbol{\pi}_{\mathbf{x}})^{\mathrm{T}}V_{\mathbf{xp}}^{k+1} \\ &\quad + V_{\mathbf{x}}^{k+1}(\boldsymbol{\pi}_{\mathbf{x}}^{\mathrm{T}}\mathbf{F}_{\mathbf{uu}}\boldsymbol{\pi}_{\mathbf{p}} + \mathbf{F}_{\mathbf{xu}}\boldsymbol{\pi}_{\mathbf{p}} + \mathbf{F}_{\mathbf{u}}\boldsymbol{\pi}_{\mathbf{xp}}) \end{aligned} \quad (4)$$

$$\begin{aligned} V_{\mathbf{pp}}^k &= \boldsymbol{\pi}_{\mathbf{p}}^{\mathrm{T}}L_{\mathbf{uu}}\boldsymbol{\pi}_{\mathbf{p}} + L_{\mathbf{u}}\boldsymbol{\pi}_{\mathbf{pp}} + (\mathbf{F}_{\mathbf{u}}\boldsymbol{\pi}_{\mathbf{p}})^{\mathrm{T}}V_{\mathbf{xx}}^{k+1}\mathbf{F}_{\mathbf{u}}\boldsymbol{\pi}_{\mathbf{p}} \\ &\quad + (\mathbf{F}_{\mathbf{u}}\boldsymbol{\pi}_{\mathbf{p}})^{\mathrm{T}}V_{\mathbf{xp}}^{k+1} + ((\mathbf{F}_{\mathbf{u}}\boldsymbol{\pi}_{\mathbf{p}})^{\mathrm{T}}V_{\mathbf{xp}}^{k+1})^{\mathrm{T}} + V_{\mathbf{pp}}^{k+1} \\ &\quad + V_{\mathbf{x}}^{k+1}(\boldsymbol{\pi}_{\mathbf{p}}^{\mathrm{T}}\mathbf{F}_{\mathbf{uu}}\boldsymbol{\pi}_{\mathbf{p}} + \mathbf{F}_{\mathbf{u}}\boldsymbol{\pi}_{\mathbf{pp}}) \end{aligned} \quad (5)$$

$V_{\mathbf{pp}}^0$ is calculated by using these equations to propagate V and its derivatives backward in time, again making extensive use of the chain rule. For a terminal cost function $\phi(\mathbf{x})$, $V_{\mathbf{x}}^D = \phi_{\mathbf{x}}$ and $V_{\mathbf{xx}}^D = \phi_{\mathbf{xx}}$. Since $\phi()$ is independent of the policy parameters, $V_{\mathbf{p}}^D$, $V_{\mathbf{xp}}^D$, and $V_{\mathbf{pp}}^D$ are zero. Often the second derivative matrix is regularized (made positive definite) by adding a diagonal matrix $\lambda\mathbf{I}$, with λ chosen by a Levenberg Marquardt or Trust Region algorithm:

$$\begin{aligned} \Delta\mathbf{p} &= -\left(\sum_{m=1}^{M}\sum_{s=1}^{S} w(m,s)V_{\mathbf{pp}}^0 + \lambda\mathbf{I}\right)^{-1} \\ &\quad \sum_{m=1}^{M}\sum_{s=1}^{S} w(m,s)(V_{\mathbf{p}}^0)^{\mathrm{T}} \end{aligned} \quad (6)$$

There are a wide variety of ways to use first and second order gradients in optimization, and our methods to calculate gradients can be used in many of them.

C. Discounting

It is often useful to apply a discount factor γ to the Bellman Equation: $V^k() = L() + \gamma V^{k+1}()$. This is easily handled by modifying the above algorithms, either by multiplying each occurrence of V^{k+1} and its derivatives in the above derivative propagation equations by γ, or equivalently, including the discounting as a separate step interleaved with the above derivative propagation equations: $V^k = \gamma V^k$, $V_{\mathbf{x}}^k = \gamma V_{\mathbf{x}}^k$, $V_{\mathbf{p}}^k = \gamma V_{\mathbf{p}}^k$, $V_{\mathbf{xx}}^k = \gamma V_{\mathbf{xx}}^k$, $V_{\mathbf{xp}}^k = \gamma V_{\mathbf{xp}}^k$, and $V_{\mathbf{pp}}^k = \gamma V_{\mathbf{pp}}^k$.

IV. LINEAR QUADRATIC BILINEAR REGULATOR

This section describes how to handle problems where full state feedback is not available, and discusses a special case that is useful to compare to Optimal Output Feedback (OOF) and also Linear Quadratic Regulator control design. The plant is linear (second derivatives of $\mathbf{F}$ are zero), the one step cost function is a pure quadratic, the policy is bilinear in $\mathbf{x}$ and $\mathbf{p}$: $\mathbf{u} = \mathbf{KCx}$, and the state $\mathbf{x} = 0$ is an equilibrium point. The gain matrix $\mathbf{K}$ contains the adjustable policy parameters and acts on a measurement vector (output) $\mathbf{y} = \mathbf{Cx}$ of dimensionality $N_{\mathbf{y}}$. To better relate to the existing literature on OOF and LQR design, we will use the notation $\mathbf{A} = \mathbf{F}_{\mathbf{x}}$, $\mathbf{B} = \mathbf{F}_{\mathbf{u}}$, $\mathbf{Q} = L_{\mathbf{xx}}$, $\mathbf{S} = L_{\mathbf{xu}}$, and $\mathbf{R} = L_{\mathbf{uu}}$. These quantities and $\mathbf{C}$ are time invariant and independent of $\mathbf{x}$. The value function is quadratic in $\mathbf{x}$ and $\mathbf{p}$. This defines the Linear Quadratic Bilinear Regulator (LQBR). Although linear policy design methods from the output feedback controller optimization community are in general more efficient (this depends on problem parameters such as M, D, S, $N_{\mathbf{p}}$, $N_{\mathbf{x}}$, and $N_{\mathbf{u}}$) [1], [16] we present this case to help the reader understand our approach and to prepare the reader for the nonlinear and time varying example we present (Section V). Unlike the LQR case, $L_{\mathbf{x}}$, $L_{\mathbf{u}}$, and $V_{\mathbf{x}}$ are non-zero since the trajectories along which these quantities are evaluated start at non-zero states. The policy parameter vector $\mathbf{p}$ is the rows of $\mathbf{K}$ concatenated

into a vector. $\boldsymbol{\pi}_{\mathbf{x}} = \mathbf{KC}$, $\boldsymbol{\pi}_{\mathbf{xx}} = 0$, and $\boldsymbol{\pi}_{\mathbf{pp}} = 0$. The first order derivative propagation equations are:

$$V_{\mathbf{x}}^k = L_{\mathbf{x}} + L_{\mathbf{u}}\mathbf{KC} + V_{\mathbf{x}}^{k+1}(\mathbf{A} + \mathbf{BKC}) \quad (7)$$

$$V_{\mathbf{p}}^k = (L_{\mathbf{u}} + V_{\mathbf{x}}^{k+1}\mathbf{B})\boldsymbol{\pi}_{\mathbf{p}} + V_{\mathbf{p}}^{k+1} \quad (8)$$

and the second order derivative propagation equations are:

$$V_{\mathbf{xx}}^k = \mathbf{Q} + \mathbf{SKC} + (\mathbf{SKC})^{\mathrm{T}} + (\mathbf{KC})^{\mathrm{T}}\mathbf{RKC} + (\mathbf{A} + \mathbf{BKC})^{\mathrm{T}} V_{\mathbf{xx}}^{k+1}(\mathbf{A} + \mathbf{BKC}) \quad (9)$$

$$V_{\mathbf{xp}}^k = \mathbf{S}\boldsymbol{\pi}_{\mathbf{p}} + (\mathbf{KC})^{\mathrm{T}}\mathbf{R}\boldsymbol{\pi}_{\mathbf{p}} + L_{\mathbf{u}}\boldsymbol{\pi}_{\mathbf{xp}} + (\mathbf{A} + \mathbf{BKC})^{\mathrm{T}} V_{\mathbf{xx}}^{k+1}\mathbf{B}\boldsymbol{\pi}_{\mathbf{p}} + (\mathbf{A} + \mathbf{BKC})^{\mathrm{T}} V_{\mathbf{xp}}^{k+1} + V_{\mathbf{x}}^{k+1}\mathbf{B}\boldsymbol{\pi}_{\mathbf{xp}} \quad (10)$$

$$V_{\mathbf{pp}}^k = \boldsymbol{\pi}_{\mathbf{p}}^{\mathrm{T}}\mathbf{R}\boldsymbol{\pi}_{\mathbf{p}} + (\mathbf{B}\boldsymbol{\pi}_{\mathbf{p}})^{\mathrm{T}} V_{\mathbf{xx}}^{k+1}\mathbf{B}\boldsymbol{\pi}_{\mathbf{p}} + (\mathbf{B}\boldsymbol{\pi}_{\mathbf{p}})^{\mathrm{T}} V_{\mathbf{xp}}^{k+1} + ((\mathbf{B}\boldsymbol{\pi}_{\mathbf{p}})^{\mathrm{T}} V_{\mathbf{xp}}^{k+1})^{\mathrm{T}} + V_{\mathbf{pp}}^{k+1} \quad (11)$$

where

$$\boldsymbol{\pi}_{\mathbf{p}}^k = \begin{pmatrix} \mathbf{y}_k^{\mathrm{T}} & \mathbf{0}^{\mathrm{T}} & \dots & \mathbf{0}^{\mathrm{T}} \\ \mathbf{0}^{\mathrm{T}} & \mathbf{y}_k^{\mathrm{T}} & \dots & \mathbf{0}^{\mathrm{T}} \\ \cdot & \cdot & \dots & \cdot \\ \mathbf{0}^{\mathrm{T}} & \mathbf{0}^{\mathrm{T}} & \dots & \mathbf{y}_k^{\mathrm{T}} \end{pmatrix} \quad (12)$$

and $\mathbf{0}^{\mathrm{T}}$ is a row vector of $N_{\mathbf{y}}$ zeros. The $\boldsymbol{\pi}_{\mathbf{p}}$ matrix has $N_{\mathbf{u}}$ rows and $N_{\mathbf{p}} = N_{\mathbf{u}}N_{\mathbf{y}}$ columns. The product of a vector $\mathbf{v}$ of length $N_{\mathbf{u}}$ and $\boldsymbol{\pi}_{\mathbf{xp}}$ is given by the $N_{\mathbf{x}}$ by $N_{\mathbf{p}}$ matrix: $\mathbf{v}\boldsymbol{\pi}_{\mathbf{xp}} = (\mathbf{v}_1\mathbf{C}^{\mathrm{T}}\ \mathbf{v}_2\mathbf{C}^{\mathrm{T}}\ \dots\ \mathbf{v}_{N_{\mathbf{u}}}\mathbf{C}^{\mathrm{T}})$. (1) and (2) or (6) can be used to update $\mathbf{K}$. (9) is the standard LQR discrete time algebraic Riccati equation when $\mathbf{S} = 0$ (no cross terms in the one step cost function) and $\mathbf{C}$ is an identity matrix (full state feedback).

Implementation Results

To verify the LQBR policy optimization algorithm and explore timing, we implemented first and second order policy optimization using both numeric (using finite differences of trajectory costs) and our analytic derivatives on the following system:

$$\mathbf{A} = \begin{pmatrix} 1 & T & 0 & 0 \\ 0 & 1 & T & 0 \\ 0 & 0 & 1 & T \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} 0 & 0 & 0 \\ T & 0 & 0 \\ 0 & T & 0 \\ 0 & 0 & T \end{pmatrix}$$

where $T = 0.001$, $\mathbf{C}$ is an identity matrix, and $L(\mathbf{x},\mathbf{u}) = 0.5(\mathbf{x}^{\mathrm{T}}\mathbf{x} + \mathbf{u}^{\mathrm{T}}\mathbf{u})$, Since we have full state feedback, the policy (gain matrix $\mathbf{K}$ in $\mathbf{u} = \mathbf{Kx}$) has 12 free parameters. We can use LQR design to identify the optimal gain matrix:

$$\mathbf{K} = \begin{pmatrix} -0.9612 & -1.5849 & -0.6312 & -0.1138 \\ -0.2717 & -0.6299 & -1.2674 & -0.5064 \\ -0.0187 & -0.1132 & -0.5052 & -1.3196 \end{pmatrix} \quad (13)$$

We tested optimization of $\mathbf{K}$ starting with the elements of $\mathbf{K}$ all equal to -1 (a stabilizing controller) along a trajectory starting at $\mathbf{x} = (1,0,0,0)^{\mathrm{T}}$. The starting cost of this trajectory is 1187 and the optimal cost is 850. Table II reports the computational cost and timing of each approach to reach a cost of 852. This cost threshold was chosen because all approaches were able to attain this cost. In Table II, Time reports wall clock time, Derivatives reports the number of gradient or Hessian calculations, and Trajectories reports the number of trajectories integrated forward. All approaches varied the length of the trajectory during optimization with a maximum trajectory length of 15000 steps. Integration of a trajectory forwards in time is cut off early if the cost is larger than the current best cost, or if a good estimate of the total cost has already been attained (future costs will be small). ε and λ are adapted during the optimization to find steps that improve the cost. We see that for both first and second order approaches using analytic derivatives is faster than using numeric derivatives. An order analysis suggests that analytic approaches will scale better in the LQBR case than numeric approaches. The cost of computing the numeric first order gradient is proportional to $N_{\mathbf{x}}^2 N_{\mathbf{p}}$. The cost of analytically computing $V_{\mathbf{x}}$ is proportional to $N_{\mathbf{x}}^2 N_{\mathbf{u}}$ and the cost to compute $V_{\mathbf{p}}$ is proportional to $N_{\mathbf{x}}N_{\mathbf{u}}N_{\mathbf{p}}$. For the LQBR case $N_{\mathbf{p}} = N_{\mathbf{x}}N_{\mathbf{u}}$, so the first order numeric gradient approach scales as $N_{\mathbf{x}}^3 N_{\mathbf{u}}$ and the corresponding analytic approach scales as $N_{\mathbf{x}}^2 N_{\mathbf{u}}^2$. For small $N_{\mathbf{u}}$ the analytic approach has a factor of $N_{\mathbf{x}}$ advantage, but for large $N_{\mathbf{u}} \approx N_{\mathbf{x}}$ the orders are the same. The cost of computing the numeric second order gradient is proportional to $N_{\mathbf{x}}^2 N_{\mathbf{p}}^2$. The most expensive matrix multiply in the analytic second order gradient computation comes in computing $V_{\mathbf{pp}}$ and is proportional to $N_{\mathbf{u}}N_{\mathbf{p}}^2$. For small $N_{\mathbf{u}}$ the analytic approach seems to have much better scaling, $N_{\mathbf{u}} < N_{\mathbf{x}}^2$, and since $N_{\mathbf{u}}$ will typically be smaller than $N_{\mathbf{x}}$ we expect the analytic approach to typically scale better than the numeric approach. The second order analytic approach can be further improved by taking advantage of sparsity in $\boldsymbol{\pi}_{\mathbf{p}}$. We see from Table II that for this policy optimization problem second order approaches are faster than the corresponding first order approaches because the number of derivative calculations is much less.

Method	Time	Derivatives	Trajectories
First order numeric	1.18s	101	1433
First order analytic	0.35s	100	207
Second order numeric	0.65s	4	529
Second order analytic	0.28s	4	14

TABLE II

LQBR IMPLEMENTATION TIMING COMPARISON.

V. LOCALLY LINEAR POLICIES

A case of particular interest to machine learning are control laws made up of collections of simple policies. Such control laws where only one simple policy is active at any one time lead to especially efficient policy update rules, relative to those of more complex global policies. In this paper we consider collections of policies that are affine: $\mathbf{u}(\mathbf{x},\mathbf{p}) = \bar{\mathbf{u}} + \bar{\mathbf{K}}\mathbf{C}(\mathbf{x} - \bar{\mathbf{x}})$. There are several ways to generate such collections. We can divide the state space up into a grid or some other tessellation and place an affine policy in each cell. We can also place affine policies along a trajectory or at random locations in state space [32] and use nearest neighbor operations to find the closest affine policy based on an appropriate distance metric. At time k a single policy is used (the jth affine policy), and its adjustable parameters are $(\bar{\mathbf{u}}^j, \bar{\mathbf{K}}^j)$. We refer to this case

as Locally Linear Policy Optimization (LLPO). The time varying version of this approach where on each time step k the kth affine policy is used is the policy optimization analog of Differential Dynamic Programming (DDP) [20], [21], which can be referred to as DDP-PO.

The key reason why using simple policies one at a time leads to efficient derivative computations is that updating the active policy j is decoupled from updating other policies. The second order gradient descent update typically has a very large reduction in computational cost. For simple policy j first order gradient descent updates its parameters in the following way:

$$\Delta \mathbf{p}^j = -\varepsilon^j \sum_{m=1}^{M} \sum_{s=1}^{S} w(m,s) V_{\mathbf{p}^j}^{\mathrm{T}} \tag{14}$$

Note that the step size ε can now depend on the simple policy being updated (this is especially useful if adaptive step size algorithms are used). Since only simple policies that are actually used are updated this leads to a reduction in computational cost. The second derivative with respect to policy parameters $V_{\mathbf{pp}}$ or Hessian matrix is block diagonal. Policy parameters of different simple policies do not interact, since only one policy operates on each time step and $V_{\mathbf{x}}$ and $V_{\mathbf{xx}}$ are used to decouple the current policy optimization from optimization of simple policies used in the future. Second order policy updates can be handled independently for each simple policy. Second order gradient descent updates the jth simple policy in the following way:

$$\begin{aligned} \Delta \mathbf{p}^j &= -\left(\sum_{m=1}^{M} \sum_{s=1}^{S} w(m,s) V_{\mathbf{p}^j \mathbf{p}^j} + \lambda^j \mathbf{I} \right)^{-1} \\ &\quad \sum_{m=1}^{M} \sum_{s=1}^{S} w(m,s) V_{\mathbf{p}^j}^{\mathrm{T}} \end{aligned} \tag{15}$$

Inverting several small Hessian matrices is typically much less expensive that inverting a single large Hessian matrix. Note that the regularization parameter λ can now depend on the simple policy being updated, which is useful if the Hessians have negative eigenvalues of various magnitudes.

The parameter vector for the jth affine policy $\mathbf{p}_j$ concatenates $\bar{\mathbf{u}}^j$ and the rows of $\bar{\mathbf{K}}^j$. If the jth affine policy is used on time step k, $\boldsymbol{\pi}_{\mathbf{x}}^k = \bar{\mathbf{K}}^j \mathbf{C}$ as in Section IV. The $\boldsymbol{\pi}_{\mathbf{p}^j}^k$ matrix has N_u rows and $N_u(1+N_y)$ columns.

$$\boldsymbol{\pi}_{\mathbf{p}^j}^k = \begin{pmatrix} 1 & 0 & \dots & 0 & \mathbf{y}_k^{\mathrm{T}} & \mathbf{0}^{\mathrm{T}} & \dots & \mathbf{0}^{\mathrm{T}} \\ 0 & 1 & \dots & 0 & \mathbf{0}^{\mathrm{T}} & \mathbf{y}_k^{\mathrm{T}} & \dots & \mathbf{0}^{\mathrm{T}} \\ \cdot & \cdot & \dots & \cdot & \cdot & \cdot & \dots & \cdot \\ 0 & 0 & \dots & 1 & \mathbf{0}^{\mathrm{T}} & \mathbf{0}^{\mathrm{T}} & \dots & \mathbf{y}_k^{\mathrm{T}} \end{pmatrix} \tag{16}$$

$\boldsymbol{\pi}_{\mathbf{xx}}^k = 0$ and $\boldsymbol{\pi}_{\mathbf{p}^j \mathbf{p}^j}^k = 0$. The product of a vector $\mathbf{v}$ of length N_u and $\boldsymbol{\pi}_{\mathbf{xp}^j}^k$ is given by the N_x by $N_u(1+N_y)$ matrix: $\mathbf{v}\boldsymbol{\pi}_{\mathbf{xp}} = \left(\mathbf{0}_{N_u \times N_u}\ \mathbf{v}_1 \mathbf{C}^{\mathrm{T}}\ \mathbf{v}_2 \mathbf{C}^{\mathrm{T}}\ \dots\ \mathbf{v}_{N_u} \mathbf{C}^{\mathrm{T}}\right)$.

The derivative propagation equations along a trajectory are:

$$V_{\mathbf{x}}^k = L_{\mathbf{x}} + L_{\mathbf{u}} \bar{\mathbf{K}}^j \mathbf{C} + V_{\mathbf{x}}^{k+1} (\mathbf{F}_{\mathbf{x}} + \mathbf{F}_{\mathbf{u}} \bar{\mathbf{K}}^j \mathbf{C}) \tag{17}$$

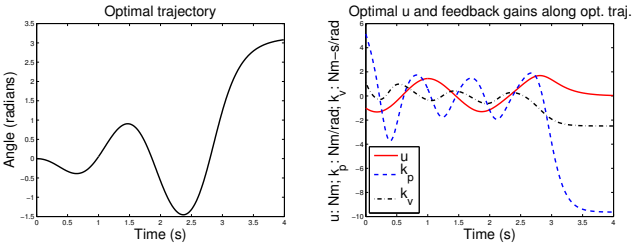

Fig. 2. Left: The optimal trajectory for the pendulum swing up (θ vs. time). Right: Optimal parameters of locally linear policies along the optimal trajectory: ($\bar{\mathbf{u}}$, position gain k_p, and velocity gain k_v).

$$\begin{aligned} V_{\mathbf{xx}}^k &= L_{\mathbf{xx}} + L_{\mathbf{xu}} \bar{\mathbf{K}}^j \mathbf{C} + (L_{\mathbf{xu}} \bar{\mathbf{K}}^j \mathbf{C})^{\mathrm{T}} + (\bar{\mathbf{K}}^j \mathbf{C})^{\mathrm{T}} L_{\mathbf{uu}} \bar{\mathbf{K}}^j \mathbf{C} \\ &\quad + (\mathbf{F}_{\mathbf{x}} + \mathbf{F}_{\mathbf{u}} \bar{\mathbf{K}}^j \mathbf{C})^{\mathrm{T}} V_{\mathbf{xx}}^{k+1} (\mathbf{F}_{\mathbf{x}} + \mathbf{F}_{\mathbf{u}} \bar{\mathbf{K}}^j \mathbf{C}) \\ &\quad + V_{\mathbf{x}}^{k+1} (\mathbf{F}_{\mathbf{xx}} + \mathbf{F}_{\mathbf{xu}} \bar{\mathbf{K}}^j \mathbf{C} + (\mathbf{F}_{\mathbf{xu}} \bar{\mathbf{K}}^j \mathbf{C})^{\mathrm{T}} \\ &\quad + (\bar{\mathbf{K}}^j \mathbf{C})^{\mathrm{T}} \mathbf{F}_{\mathbf{uu}} \bar{\mathbf{K}}^j \mathbf{C}) \end{aligned} \tag{18}$$

For the affine policy currently in use (simple policy j):

$$V_{\mathbf{p}^j}^k = (L_{\mathbf{u}} + V_{\mathbf{x}}^{k+1} \mathbf{F}_{\mathbf{u}}) \boldsymbol{\pi}_{\mathbf{p}^j} + V_{\mathbf{p}^j}^{k+1} \tag{19}$$

$$\begin{aligned} V_{\mathbf{xp}^j}^k &= L_{\mathbf{xu}} \boldsymbol{\pi}_{\mathbf{p}^j} + (\bar{\mathbf{K}}^j \mathbf{C})^{\mathrm{T}} L_{\mathbf{uu}} \boldsymbol{\pi}_{\mathbf{p}^j} + L_{\mathbf{u}} \boldsymbol{\pi}_{\mathbf{xp}^j} \\ &\quad + (\mathbf{F}_{\mathbf{x}} + \mathbf{F}_{\mathbf{u}} \bar{\mathbf{K}}^j \mathbf{C})^{\mathrm{T}} V_{\mathbf{xx}}^{k+1} \mathbf{F}_{\mathbf{u}} \boldsymbol{\pi}_{\mathbf{p}^j} \\ &\quad + V_{\mathbf{x}}^{k+1} (\mathbf{F}_{\mathbf{xu}} \boldsymbol{\pi}_{\mathbf{p}^j} + (\bar{\mathbf{K}}^j \mathbf{C})^{\mathrm{T}} \mathbf{F}_{\mathbf{uu}} \boldsymbol{\pi}_{\mathbf{p}^j} + \mathbf{F}_{\mathbf{u}} \boldsymbol{\pi}_{\mathbf{xp}^j}) \\ &\quad + (\mathbf{F}_{\mathbf{x}} + \mathbf{F}_{\mathbf{u}} \bar{\mathbf{K}}^j \mathbf{C})^{\mathrm{T}} V_{\mathbf{xp}^j}^{k+1} \end{aligned} \tag{20}$$

$$\begin{aligned} V_{\mathbf{p}^j \mathbf{p}^j}^k &= \boldsymbol{\pi}_{\mathbf{p}^j}^{\mathrm{T}} L_{\mathbf{uu}} \boldsymbol{\pi}_{\mathbf{p}^j} + (\mathbf{F}_{\mathbf{u}} \boldsymbol{\pi}_{\mathbf{p}^j})^{\mathrm{T}} V_{\mathbf{xx}}^{k+1} \mathbf{F}_{\mathbf{u}} \boldsymbol{\pi}_{\mathbf{p}^j} \\ &\quad + (\mathbf{F}_{\mathbf{u}} \boldsymbol{\pi}_{\mathbf{p}^j})^{\mathrm{T}} V_{\mathbf{xp}^j}^{k+1} + ((\mathbf{F}_{\mathbf{u}} \boldsymbol{\pi}_{\mathbf{p}^j})^{\mathrm{T}} V_{\mathbf{xp}^j}^{k+1})^{\mathrm{T}} \\ &\quad + \boldsymbol{\pi}_{\mathbf{p}^j}^{\mathrm{T}} V_{\mathbf{x}}^{k+1} \mathbf{F}_{\mathbf{uu}} \boldsymbol{\pi}_{\mathbf{p}^j} + V_{\mathbf{p}^j \mathbf{p}^j}^{k+1} \end{aligned} \tag{21}$$

For affine policies not currently in use but that have been used (simple policy l):

$$V_{\mathbf{p}^l}^k = V_{\mathbf{p}^l}^{k+1} \tag{22}$$

$$V_{\mathbf{xp}^l}^k = (\mathbf{F}_{\mathbf{x}} + \mathbf{F}_{\mathbf{u}} \bar{\mathbf{K}}^l \mathbf{C})^{\mathrm{T}} V_{\mathbf{xp}^l}^{k+1} \tag{23}$$

$$V_{\mathbf{p}^l \mathbf{p}^l}^k = V_{\mathbf{p}^l \mathbf{p}^l}^{k+1} \tag{24}$$

The update equations (14) or (15) are used.

Implementation Results

To verify the LLPO algorithm and explore timing, we implemented both numeric and analytic first and second order policy optimization on a pendulum swing up problem with the following dynamics: $\ddot{\theta} = (\tau - mgl\sin(\theta))/\mathbf{I}$ and one step cost function: $L(\mathbf{x},\mathbf{u}) = 0.5T(0.1\theta^2 + \tau^2)$, where $\mathbf{x} = (\theta, \dot{\theta})^{\mathrm{T}}$, θ is the pendulum angle with straight down being 0, τ is a torque at the base, $T = 0.01$, the moment of inertia about the joint is $\mathbf{I} = 0.3342$, the product of mass, gravity, and the pendulum length is $mgl = 4.905$, and $\mathbf{C}$ is an identity matrix. We have found the optimal trajectory (cost = 3.5385) using dynamic programming (DP) and differential dynamic programming (DDP) (Figure 2 Left). We can use these solutions to see how the optimal parameters of locally linear policies ($\bar{\mathbf{u}}$, position gain k_p, and velocity gain k_v) vary along the optimal trajectory (Figure 2 Right).

Method	10 policies	100 policies	500 policies
First order numeric	0.108	11.3	53
First order analytic	0.098	0.104	0.124
Second order numeric	450	45000	1061000
Second order analytic	0.77	0.89	1.20

TABLE III

LLPO IMPLEMENTATION TIMING COMPARISON. ENTRIES ARE MILLISECONDS PER GRADIENT CALCULATION.

We will use this problem to test algorithm timing in the context of numeric and analytical first and second order gradients on a 500 step trajectory. In the numeric approach we used finite differencing of total trajectory costs to numerically estimate $V_{\mathbf{p}}$ and $V_{\mathbf{pp}}$. We can vary the number of affine policies and see how the cost of computing $V_{\mathbf{p}}$ and $V_{\mathbf{pp}}$ increases (Table III). Table entries report time in milliseconds for one calculation of $V_{\mathbf{p}}$ or $V_{\mathbf{p}}$ and $V_{\mathbf{pp}}$ for 10, 100, and 500 local policies. We see that analytic derivatives become relatively much cheaper to compute as the number of affine policies increases, since the numeric approaches have to vary all the parameters of all the simple policies to estimate derivatives, while the analytic approaches only require a number of updates related to the length of the trajectory and largely independent of the number of simple policies. The cost of the numeric first order gradient computation is proportional to the number of simple policies, while the numeric second order gradient computation grows with the square of the number of simple policies. In the analytic approaches the computational cost of finding the nearest neighbor simple policy and initializing all simple policies for each new trajectory depend on the number of simple policies. The cost for updating the policy gradients for simple policies not used on the current time step, inverting the Hessian matrices, and updating simple policy parameters depend only on the number of simple policies used, so in the worst case this cost is proportional to the length of each trajectory. Simple policies that have not been used on the current trajectory do not need to be updated until they are used. In practice the total cost of the analytic approaches is almost independent of the number of simple policies available or used.

Using a single simple policy at a time and using analytic derivatives makes optimization and learning possible for large complex policy optimization problems. In our experience so far using collections of simple policies, sometimes the second order analytic approach is faster than the first order analytic approach because it takes many fewer iterations to converge, and sometimes the first order approach has a slight edge. Either the solutions found are equivalent, or the second order approach finds a better solution.

Weighted Locally Linear Policy Optimization

The consequence of using multiple simple policies on each time step by forming a weighted average of the outputs is that the Hessian matrix may no longer be block diagonal or have a form that reduces the computational cost of inverting it. If on time step k the jth policy has the weight w_j, then the derivatives of the policy also now have that weighting factor. With the lth policy also active, cross terms between $\boldsymbol{\pi}_{\mathbf{p}^j}$, $\boldsymbol{\pi}_{\mathbf{p}^l}$, $V_{\mathbf{xp}^j}$, and $V_{\mathbf{xp}^l}$ arise which destroy the block diagonal nature of the Hessian ($V_{\mathbf{p}^j\mathbf{p}^l} \neq 0$). In the general case the Hessian is sufficiently full that (6) must be used to compute the second order update. We refer to this algorithm as Weighted Locally Linear Policy Optimization (WLLPO). The weights provide an interesting perspective on eligibility traces used in reinforcement learning. w_j appears quadratically in some terms in the second order $V_{\mathbf{p}^j\mathbf{p}^j}$ update.

VI. HANDLING MULTIPLE MODELS WITH DIFFERENT STATES

So far we have assumed all of the multiple models have the same state vector in terms of which variables are in each element and the size of the state vector. We will outline what happens to the first order update formulas (Equations 1 and 2) when we are optimizing over multiple models with different state vectors $\mathbf{z}^m$. Since we are planning, we assume we know the dynamics of each model: $\mathbf{z}^m_{i+1} = \mathbf{F}^m(\mathbf{z}^m_i, \mathbf{u}_i)$. In order to use the same policy with all models, each model must provide a vector of observations: $\mathbf{y} = \mathbf{g}^m(\mathbf{z}^m)$ and the common policy is a function of those measurements: $\boldsymbol{\pi}(\mathbf{y}, \mathbf{p})$. In order to use the same one step cost function $L(\mathbf{x}, \mathbf{u})$, each model must provide a way to generate a "nominal" state: $\mathbf{x} = \mathbf{h}^m(\mathbf{z}^m)$. This function is unnecessary if the one step cost function is a function of the observation vector $L(\mathbf{y}, \mathbf{u})$. Finally, there must be a way to start each model's trajectory from an equivalent state $\mathbf{z}^m_0$, given a nominal starting state $\mathbf{x}_0$. The Bellman Equation for each model is: $V^{m,k}(\mathbf{z}^m, \mathbf{p}) = L(\mathbf{h}^m(\mathbf{z}^m), \boldsymbol{\pi}(\mathbf{g}^m(\mathbf{z}^m), \mathbf{p})) + V^{m,k+1}(\mathbf{F}^m(\mathbf{z}^m, \boldsymbol{\pi}(\mathbf{g}^m(\mathbf{z}^m), \mathbf{p})), \mathbf{p})$. The first order derivative propagation equations are (suppressing the m superscripts):

$$V^k_{\mathbf{z}} = L_{\mathbf{x}}\mathbf{h}_{\mathbf{z}} + L_{\mathbf{u}}\boldsymbol{\pi}_{\mathbf{y}}\mathbf{g}_{\mathbf{z}} + V^{k+1}_{\mathbf{z}}(\mathbf{F}_{\mathbf{z}} + \mathbf{F}_{\mathbf{u}}\boldsymbol{\pi}_{\mathbf{y}}\mathbf{g}_{\mathbf{z}}) \tag{25}$$

$$V^k_{\mathbf{p}} = (L_{\mathbf{u}} + V^{k+1}_{\mathbf{z}}\mathbf{F}_{\mathbf{u}})\boldsymbol{\pi}_{\mathbf{p}} + V^{k+1}_{\mathbf{p}} \tag{26}$$

The second order derivative propagation equations for $V_{\mathbf{zz}}$, $V_{\mathbf{zp}}$, and $V_{\mathbf{pp}}$ are extended in a similar way (extensive use of the chain rule). During policy optimization the appropriate model specific dynamics, observation equation, $\mathbf{h}()$, value function, and derivative propagation equations are used on each application of a model m to a starting point.

VII. DISCUSSION

See [16] for more discussion of how to handle constraints, policies with internal state, global optimization of policies, policy regularization, handling stochastic systems and dynamic policies, simplifying policy structures, adaptive grids and parametrizations, receding horizon control (RHC)/model predictive control (MPC), model following, continuous time policy optimization, minimax approaches to policy optimization, inverse optimal control, convergence, and how to best choose the distribution of models used in the design process based on experience with the system to be controlled, how to pick initial states, and how to weight the models and initial states during training.

When considering computer architectures that offer many cores or computational pipelines, it is important to note that

both numeric and analytic gradient-based policy optimization algorithms are embarrassingly parallel. Each trajectory can be processed on a different processing element. Numeric derivative approaches are embarrassingly parallel on a finer scale, in that each forward integration of a trajectory with a particular set of policy parameters can be processed in parallel. Analytic approaches need to compute $V_{\mathbf{x}}$ and $V_{\mathbf{xx}}$ for a trajectory first, then $V_{\mathbf{xp}}$, and then $V_{\mathbf{p}}$ and $V_{\mathbf{pp}}$, so a little more care is needed to implement parallelism at the single trajectory level, but it is still relatively easy to do with some additional memory cost.

VIII. CONCLUSION

We presented a policy optimization approach that achieves robustness by simultaneously designing for multiple models. We developed analytic first and second order derivatives for efficient control law design using gradient-based algorithms. The gradient-based approach enables handling of nonlinear and time-varying models and policies, and multiple models with different model structures. We described a generalization of LQR design: the Linear Quadratic Bilinear Regulator (LQBR) and presented some implementation and timing results, with a comparison to a perturbation-based policy optimization approach. We emphasized that uncorrelated additive or multiplicative noise is a poor proxy for unmodeled dynamics in control law design. We highlighted the benefit of control laws made up of collections of simple policies where only one simple policy is active at a time. Controller optimization and learning is particularly fast and effective in this situation because second derivatives are decoupled.

IX. ACKNOWLEDGMENTS

This material is based upon work supported in part by the US National Science Foundation (EEC-0540865, ECCS-0824077, and IIS-0964581) and the DARPA M3 program.

REFERENCES

[1] A. Varga, "Optimal output feedback control: a multi-model approach," in *IEEE International Symposium on Computer-Aided Control System Design*, 1996, pp. 327–332.

[2] L. A. Feldkamp and G. V. Puskorius, "Fixed-weight controller for multiple systems," in *International Conference on Neural Networks*, 1997, pp. 773–778.

[3] H. T. Su and T. Samad, "Neuro-control design: Optimization aspects," in *Neural Systems For Control*, O. Omidvar and D. L. Elliott, Eds. Academic Press, 1997, pp. 259–288.

[4] Y. Piguet, U. Holmberg, and R. Longchamp, "A minimax approach for multi-objective controller design using multiple models," *International Journal of Control*, vol. 72, no. 7-8, pp. 716–726, 1999.

[5] D. Bagnell and J. Schneider, "Autonomous helicopter control using reinforcement learning policy search methods," in *IEEE International Conference on Robotics and Automation*, 2001.

[6] R. H. Nyström, J. M. Böling, J. M. Ramstedt, H. T. Toivonen, and K. E. Häggblom, "Application of robust and multimodel control methods to an ill-conditioned distillation column," *Journal of Process Control*, vol. 12, pp. 39–53, 2002.

[7] D. V. Prokhorov, "Training recurrent neurocontrollers for robustness with derivative-free Kalman filter," *IEEE Transactions on Neural Networks*, vol. 17, no. 6, pp. 1606–1616, 2006.

[8] F. J. Bejarano, A. Poznyak, and L. Fridman, "Min-max output integral sliding mode control for multiplant linear uncertain systems," in *American Control Conference*, 2007, pp. 5875–5880.

[9] M. McNaughton, "CASTRO: robust nonlinear trajectory optimization using multiple models," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2007, pp. 177 – 182.

[10] J. Shinar, V. Glizer, and V. Turetsky, "Solution of a linear pursuit-evasion game with variable structure and uncertain dynamics," in *Advances in Dynamic Game Theory - Numerical Methods, Algorithms, and Applications to Ecology and Economics*, S. Jorgensen, M. Quincampoix, and T. Vincent, Eds. Birkhauser, 2007, vol. 9, pp. 195–222.

[11] V. Azhmyakov, V. Boltyanski, and A. Poznyak, "The dynamic programming approach to multi-model robust optimization," *Nonlinear Analysis, Theory, Methods & Applications*, vol. 72, no. 2, pp. 1110–1119, 2010.

[12] J. Z. Kolter, "Learning and control with inaccurate models," Ph.D. dissertation, Stanford, August 2010.

[13] S. Sanan, M. H. Ornstein, and C. G. Atkeson, "Physical human interaction for an inflatable manipulator," in *IEEE Engineering in Medicine and Biology Society (EMBC)*, 2011.

[14] R. G. Heikes, D. C. Montgomery, and R. L. Rardin, "Using common random numbers in simulation experiments - an approach to statistical analysis," *Simulation*, vol. 27, no. 3, pp. 81–85, 1976.

[15] A. Y. Ng and M. Jordan, "Pegasus: A policy search method for large MDPs and POMDPs," in *Uncertainty in Artificial Intelligence, Proceedings of the Sixteenth Conference*, 2000.

[16] C. G. Atkeson, "Efficient robust policy optimization (long version)," 2012, www.cs.cmu.edu/∼cga/papers/acc2012-tr.pdf.

[17] P. J. Werbos, "Multiple models for approximate dynamic programming and true intelligent control: Why and how," in *Proc. 10th Yale Conference on Learning and Adaptive Systems*, K. Narendra, Ed. EE Dept. Yale University, 1998.

[18] A. Griewank and A. Walther, *Evaluating derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, 2008.

[19] P. J. Werbos, "Backwards differentiation in AD and neural nets: Past links and new opportunities," in *Automatic Differentiation: Applications, Theory and Implementations*, M. Bücker, G. Gorliss, P. Hovland, U. Naumann, and B. Norris, Eds., 2006, pp. 15–34.

[20] D. H. Jacobson and D. Q. Mayne, *Differential Dynamic Programming*. Elsevier, New York, NY, 1970.

[21] P. Dyer and S. R. McReynolds, *The Computation and Theory of Optimal Control*. Academic Press, New York, NY, 1970.

[22] R. J. Williams, "Adaptive state representation and estimation using recurrent connectionist networks," in *Neural Networks for Control*. MIT Press, 1990, pp. 97–114.

[23] P. J. Werbos, "Neurocontrol and supervised learning: An overview and evaluation," in *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches*. Van Nostrand Reinhold, 1992, pp. 65–89.

[24] ——, *The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting*. Wiley, 1994.

[25] D. V. Prokhorov, "Backpropagation through time and derivative adaptive critics — a common framework for comparison," in *Handbook of Learning and Approximate Dynamic Programming*. Wiley-IEEE Press, 2004, pp. 381–404.

[26] W. T. Miller, R. S. Sutton, and P. J. Werbos, *Neural Networks for Control*. MIT Press, 1990.

[27] D. A. White and D. A. Sofge, *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches*. Van Nostrand Reinhold, 1992.

[28] J. Si, A. G. Barto, W. B. Powell, and D. W. II, *Handbook of Learning and Approximate Dynamic Programming*. Wiley-IEEE Press, 2004.

[29] F. L. Lewis, G. Lendaris, and D. Liu, "Special issue on approximate dynamic programming and reinforcement learning for feedback control," *IEEE Trans. Syst. Man, Cybern – B: Cybernetics*, vol. 41, no. 1, pp. 896–897, 2008.

[30] F. L. Lewis and D. Vrabie, "Reinforcement learning and adaptive dynamic programming for feedback control," *IEEE Circuits and Systems Magazine*, vol. 9, no. 3, pp. 32–50, 2009.

[31] F.-Y. Wang, H. Zhang, and D. Liu, "Adaptive dynamic programming: An introduction," *IEEE Computational Intelligence Magazine*, vol. 4, no. 2, pp. 39–47, 2009.

[32] C. G. Atkeson and B. Stephens, "Random sampling of states in dynamic programming," *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, vol. 38, no. 4, pp. 924–929, 2008.