Real World Control as Online Learning:
Fixing the Dynamics Model in Real Time

Nathan Ratliff

Tech Report: Max Planck Institute for Intelligent Systems

Written Sep 10, 2015; Updated Nov 18, 2015; Posted Jan 30, 2016
Patent pending technology

Abstract

This document develops a new control methodology based on running online learning to fix the (inverse) dynamics model in real time using the data stream from the robot (usually at a rate of 1000 point per second). We motivate this methodology by analyzing the problem of tracking integral curves of an acceleration policy, emphasizing the practical deficiency of using vanilla inverse dynamics (despite the acceleration policy representing a full feedback policy). We then generalize this methodology to demonstrate that many of the most popular control tools that have had real world success may actually be viewed as an online gradient descent in this same way. Real world success comes from fixing the dynamics model in real time. More we show that this new view shows that existing controllers typically use very simplistic models for representing the inverse dynamics function, usually just as a constant offset from the the rigid body calculation. This observation opens up a wide range of more sophisticated control methodologies based on more informative local hypothesis classes that we can explore further in future work.

1 The difficulty of following acceleration policies

Acceleration policies are natural representations of motion: how should the robot accelerate if it finds itself in a given configuration moving with a particular velocity? Writing down these policies is easy, especially for manipulation platforms with invertible dynamics. And, importantly, it's very easy to shape these policies as desired. Simulation is as easy as simple kinematic forward integration, which means that motion optimizers, especially those generating second-order approximations to the problems (LQRs), can analyze in advance what following these policies means. Additionally, LQRs can effectively shape a whole (infinite) bundle of integral curve trajectories simultaneously by adjusting (maybe through a training process) the strengths and shape of various cost terms. DMPs are another form of acceleration policy shaped through training. And importantly, acceleration policies are naturally reactive. They know what to do from any position and velocity the robot might find itself in.

Unfortunately, in practice following acceleration policies is hard. The fidelity of the dynamics model defines how accurately we can simulate the behavior of the robot, and it's widely understood that we can never actually simulate the robot well enough to sufficiently predict it's behavior on the real system.

So we need control. If we know exactly the kinematic motion trajectory we want the robot to take, we can meticulously measure the actual system configurations (positions and velocities) at a very fast rate (maybe a thousand times a second (1kHz)) and correct in real time for errors induced by model inaccuracies. We can even throw away the model entirely as long as we're aggressive enough in correcting the robot's behavior.

But in order to leverage traditional control strategies, we need trajectories. Trajectories tell the robot where it should be and how fast it should be moving at any moment in time. They're expressed in units of kinematic states (positions and velocities) enabling the straightforward calculation of state-errors. But that means we need to reduce the robot's view of the expressive, sometimes even global, acceleration policy down to a single myopic thread of kinematic motion. Using these tools at face value makes it difficult to adjust naturally to perturbations which makes applications like collaborative robotics, where robots must work in close proximity to (or even collaboratively with) human co-workers, complicated. The norm should be an expectation of reactive behavior and spurious perturbation, but that's hard with fixed trajectories.

This document proposes an alternative form of control rooted in a combination of techniques from online learning and adaptive control, that enables the direct execution of full acceleration policies significantly simplifying the interface to behavior generation. The aim of this methodology is to separate acceleration policies from the robot's dynamics through a layer of adaptive control adjusting in real-time to errors in the dynamics model so that actual observed accelerations correctly reflect the desired commanded accelerations. The communication channel between acceleration commands and the torques that would generate those accelerations is usually very noisy. Our aim is to leverage the real-time stream of data produced by the robot (often at a rate of 1000 data points per second) to fix that communication channel in real time, enabling a true interface to executing acceleration policies.

2 Motivation by analogy

There are two analogies that motivate the techniques derived in this document. Both are based around the simple observation that if we want to accelerate at a certain rate and we aren't, we should add more torque.

The first analogy is driving a car, especially when starting from a standstill. Getting a car to move from a standstill can be tricky, particularly if we aren't familiar with the stickiness of the accelerator. If we want to accelerate, we push on the accelerator. But if you're me, pushing on the accelerator does nothing. So, frustrated, you push harder and harder until suddenly you lurch forward. Both before, when you weren't accelerating at all, and now when you're accelerating too much, you can measure the magnitude of the acceleration because you feel it as a pressure against your back. Now that you're accelerating too much, you let off on the accelerator to adjust. Generally, we don't imagine a hallucinated car ("the car that might have been") getting away from us to decide how much to press down on the accelerator. We might think of a desired velocity set point and decide on a desired acceleration to get there with ease, but that's just used to calculate accelerations (changing velocities requires acceleration). The feedback we're using to adjust the pressure on the petal is based on the discrepancy between what we wanted the acceleration to be and what we actually measure. If we aren't accelerating as much as desired, then just press harder.

The second analogy is a simple thought experiment on how you'd control around stictions if you were a little torque gnome observing a real-time 1kHz stream of data coming from the robot. You want to accelerate at a desired rate, but you estimate some torques, send them down for execution, and nothing happens. You see from your stream of data that the system is correctly applying those torques, but it must be that there are stictions (or something) preventing the robot from moving. So you add some more torque. You don't add too much, but you add a little, generally proportional to the magnitude of the difference between the acceleration we wanted and the acceleration we're getting. For a while you're data stream is still showing zero acceleration, but eventually you break free and push past the stiction. But now the torque's too high! The friction coefficient has dropped to normal moving levels and the robot's accelerating too fast. So now you again do the same thing: you remove torque until the acceleration is right where you want it to be. All of this happens within just tenths of a second, but to you, you've seen 200-300 data points stream by, so you've had time to adjust accordingly. More, even though it's actually difficult to get a handle on the true acceleration because the numbers in the data stream are quite noisy, you have time to take averages of those values to get a better signal to adjust the torque. The world around you runs very very slowly, as does the desired acceleration signal you're trying to match, so you have time to do a lot of measuring and adjusting to make sure the slow pokes in the real world don't realize just how hard your job is.

3 Basic equations

This section presents the basic equations behind the algorithm.

3.1 General form of the equations of motion

Lagrangian mechanics states that the equations of motion can be summarized in the general form

$$\boldsymbol{\tau} = \frac{d}{dt}\frac{\partial \mathcal{L}}{\partial \dot{\boldsymbol{q}}} - \frac{\partial \mathcal{L}}{\partial \boldsymbol{q}}, \tag{1}$$

where $\mathcal{L}$ is the Lagrangian

$$\mathcal{L}(\boldsymbol{q}, \dot{\boldsymbol{q}}) = \mathcal{K}(\boldsymbol{q}, \dot{\boldsymbol{q}}) - \mathcal{V}(\boldsymbol{q}), \tag{2}$$

with $\mathcal{K}(\boldsymbol{q}, \dot{\boldsymbol{q}})$ the system's kinetic energy and $\mathcal{V}(\boldsymbol{q})$ the system's potential energy. When the kinetic energy takes on a quadratic form $\mathcal{K}(\boldsymbol{q}, \dot{\boldsymbol{q}}) = \frac{1}{2}\dot{\boldsymbol{q}}^T \boldsymbol{M}(\boldsymbol{q})\dot{\boldsymbol{q}}$ (as it does for all classical mechanical systems), we can write

$$\boldsymbol{\tau} = \boldsymbol{M}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \boldsymbol{h}(\boldsymbol{q}, \dot{\boldsymbol{q}}). \tag{3}$$

In general, we don't know what $\boldsymbol{M}$ and $\boldsymbol{h}$ are, but we can estimate them using rigid body assumptions as $\widehat{\boldsymbol{M}}$ and $\widehat{\boldsymbol{h}}$. These assumptions often severely restrict the class of representable models, so even using parameter estimation techniques with friction and stiction models, the fitted models can't accurately describe the true behavior of the system. This disparity constitutes the observed discrepancy between simulation and real-world performance.

Equation 3 is actually written in *inverse dynamics* form. Given an acceleration $\ddot{\boldsymbol{q}}$, it tells us what torque $\boldsymbol{\tau}$ would produce that acceleration. Inverting the expression gives the equation for forward dynamics:

$$\ddot{\boldsymbol{q}} = \boldsymbol{M}^{-1}\big(\boldsymbol{\tau} - \boldsymbol{h}\big). \tag{4}$$

This equation expresses the kinematic effect of applied torques $\boldsymbol{\tau}$ in terms of the accelerations $\ddot{\boldsymbol{q}}$ they generate.

3.2 The problem with inverse dynamics methods

Our model is only approximate. So given any desired acceleration $\ddot{\boldsymbol{q}}_d$, we can calculate torques $\boldsymbol{\tau} = \widehat{\boldsymbol{M}}\ddot{\boldsymbol{q}}_d + \widehat{\boldsymbol{h}}$ using our approximate inverse dynamics model, but when we apply those torques we have to push them through the true system which may differ substantially from the model:

$$\ddot{\boldsymbol{q}}_a = \boldsymbol{M}^{-1}\Big(\big(\widehat{\boldsymbol{M}}\ddot{\boldsymbol{q}}_d + \widehat{\boldsymbol{h}}\big) - \boldsymbol{h}\Big), \tag{5}$$

where here $\ddot{\boldsymbol{q}}_a$ denotes the actual observed accelerations of the true system.

This problem is particularly acute in the presence of nonlinear frictions and strong stictions. Often the calculated torque doesn't correctly compensate for those forces, which means in many cases the robot doesn't even move, or it moves much differently than expected, especially when commanded to move slowly for meticulous manipulation motions.

The typical solution is to instead reduce the problem to our hammer: PD control with feed-forward inverse dynamics. Why? Because it works really well. Any acceleration policy $\ddot{\boldsymbol{q}}_d = f(\boldsymbol{q}, \dot{\boldsymbol{q}})$ represents an infinite collection of trajectories. For every initialization $(\boldsymbol{q}_0, \dot{\boldsymbol{q}}_0)$, forward integration produces a desired integral curve that we'd like the system to follow. If all we want to do is follow that integral curve well (which holds even when feedback comes in the form of modifications to future portions of our time varying differential equation), this technique works. Independent of the fidelity of the inverse dynamics model, we can tune the PD gains to reject disturbances resulting from modeling errors. DMPs often follow this pattern: they're used as *trajectory generators* and we can follow the resulting trajectories robustly using traditional methods.

However, these system's can't be robust to physical perturbations because they aren't actually following the policy mapped out by the differential equation. You push the robot, and it can't distinguish between that perturbation and one resulting from a simple model inaccuracy. In the best case, you have a pretty good dynamics model enabling low PD gains, and the system will perturb away from the trajectory, but since it's following a trajectory it'll always then move back to continue along its path. It's a very mechanical motion stereotypical of a robot that you might see good comedians mimic on stage for laughs. Strange and somewhat puppeted behavior sitting squarely within the uncanny valley.

The only way to circumvent these issues when reducing the problem to trajectory following is to somehow detect when physical perturbations occur, acknowledge new states as seeds for new integral curves, and devise a policy for blending from one trajectory to the next. This system is complicated and requires a lot of tuning since unpredictable perturbations have unpredictable profiles. Often we want to choose an epsilon trough around the trajectory and predict physical perturbations as deviations beyond that epsilon trough. In practice, that epsilon is both too big and too small. It's too big because perturbations ultimately lock onto a new trajectory and we see typical trajectory following artifacts within that epsilon such as low-frequency oscillations around the trajectory or slight overshoots given the unpredictability of the perturbation profile. But it's also too small, and when the model inaccuracies are most severe, we see spurious blending triggered simply from attempts to follow the integral curves.

Our solution: fix the model locally in real time using the 1000 data points per second we get back from the robot. If we can fix the model in real time, we don't have to worry about any of the complexity introduced by reducing the problem to trajectory following. We can simply execute the acceleration policy using the patched inverse dynamics model. The key will be a new gradient descent online learning method that attempts to directly minimize the error between desired and observed accelerations. Section 5 below details the differences between this technique, which we might characterize as a *direct* adaptive control method (as it directly minimizes the tracking error), and traditional online dynamics learning techniques which actually learn slightly off policy and may be characterized *indirect* adaptive control methods.

3.3 Taking the gradient of an error on accelerations

Returning to the equation expressing the how desired accelerations translate to applied torques through inverse dynamics and then push through the true system to create observed accelerations, we now propose adding a torque offset $\boldsymbol{w}$ to the applied torques that we'll attempt to track over time as the difference between the torques our inaccurate model predicts and the torque we should have predicted to generate the desired accelerations:

$$\ddot{\boldsymbol{q}}_a(\boldsymbol{w}) = \boldsymbol{M}^{-1}\left[\left(\widehat{\boldsymbol{M}}(\boldsymbol{q})\ddot{\boldsymbol{q}}_d + \widehat{\boldsymbol{h}}(\boldsymbol{q},\dot{\boldsymbol{q}}) + \boldsymbol{w}\right) - \boldsymbol{h}\right]. \tag{6}$$

The observed accelerations $\ddot{\boldsymbol{q}}_a$ are then a function of $\boldsymbol{w}$. Increasing the added applied torque increases the accelerations, etc. We can measure the error between desired accelerations and observed accelerations as

$$f(\boldsymbol{w}) = \frac{1}{2}\|\ddot{\boldsymbol{q}}_d - \ddot{\boldsymbol{q}}_a(\boldsymbol{w})\|^2_{\boldsymbol{M}}. \tag{7}$$

Here we're expressing the error as a squared norm scaled by $\boldsymbol{M}$, the *true* mass matrix. This error is a common metric most famously used in Gauss's Principle of Least Constraint. In this case, we don't actually know $\boldsymbol{M}$ (or $\boldsymbol{h}$, which appears in $\ddot{\boldsymbol{q}}_a(\boldsymbol{w})$), so we can't evaluate the objective directly.

However, if we evaluate the gradient of the expression, we see that

$$\nabla_{\boldsymbol{w}} f(\boldsymbol{w}) = \nabla_{\boldsymbol{w}} \frac{1}{2}\|\ddot{\boldsymbol{q}}_d - \ddot{\boldsymbol{q}}_a(\boldsymbol{w})\|_{\boldsymbol{M}}^2 \tag{8}$$

$$= -\left[\frac{\partial \ddot{\boldsymbol{q}}_a(\boldsymbol{w})}{\partial \boldsymbol{w}}\right] \boldsymbol{M}\big(\ddot{\boldsymbol{q}}_d - \ddot{\boldsymbol{q}}_a(\boldsymbol{w})\big) \tag{9}$$

$$= -\boldsymbol{M}^{-1}\boldsymbol{M}\big(\ddot{\boldsymbol{q}}_d - \ddot{\boldsymbol{q}}_a(\boldsymbol{w})\big) \tag{10}$$

$$= -\big(\ddot{\boldsymbol{q}}_d - \ddot{\boldsymbol{q}}_a(\boldsymbol{w})\big), \tag{11}$$

which is just a difference between the acceleration we wanted and the acceleration we actually measured. So even though we can't evaluate either objective function or it's gradient directly, we can measure it's gradient at every control cycle by estimating the true accelerations. Note that throughout this derivation, we can replace the offset $\boldsymbol{w}$ with a more sophisticated function approximator $\phi(\boldsymbol{q}, \dot{\boldsymbol{q}}, \ddot{\boldsymbol{q}}_d, \boldsymbol{w})$, and the drivation is similar. The only difference in the final expression is that the gradient is pushed through the transpose of the function approximator's Jacobian $\boldsymbol{J}_\phi = \frac{\partial \phi}{\partial \boldsymbol{w}}$. For simplicity throughout this paper, we derive everything with just $\phi(\boldsymbol{w}) = \boldsymbol{w}$, which reduces the Jacobian to $\boldsymbol{J}_\phi = \boldsymbol{I}$.

Adding regularization to the objective gives

$$f(\boldsymbol{w}) = \frac{1}{2}\|\ddot{\boldsymbol{q}}_d - \ddot{\boldsymbol{q}}_a(\boldsymbol{w})\|_{\boldsymbol{M}}^2 + \frac{\lambda}{2}\|\boldsymbol{w}\|^2. \tag{12}$$

so the resulting online gradient descent update rule becomes

$$\boldsymbol{w}_{t+1} = (1-\lambda)\boldsymbol{w}_t + \eta_t\big(\ddot{\boldsymbol{q}}_d^{(t)} - \ddot{\boldsymbol{q}}_a^{(t)}\big). \tag{13}$$

This expression is essentially an integral term on the acceleration error with a forgetting factor. The advantage of writing it as online learning is that now we have a box of tricks from the learning community we can leverage to improve performance.

4 Tricks from online learning and adaptive control

This section reviews two tricks pulled from the combined online learning literature (or more general stochastic gradient descent machine learning literature) and the adaptive control literature to remove the potential for parameter oscillations and track changes in modeling errors while simultaneously enabling high accuracy for precise meticulous movements.

4.1 Parameter oscillations in online learning and their physical manifestation

Parameter oscillations in neural networks are a problem resulting from ill-conditioning of the objective. The objective, as seen from the the parameter space (under Euclidean geometry, which is a common easy choice), is quite elongated, meaning that it's extremely stretched with a highly diverse Hessian Eigenspectrum. Gradient descent alone in those settings undergoes severe oscillations making it's progress slow. More importantly, in our case, oscillations resulting from the ill-conditioning of the problem manifest physically as oscillations in the controller when the step size is too large. Fortunately, the learning community has a number of tricks to prevent these oscillations and promote fast convergence.

The most commonly used method for preventing oscillations is the use of momentum. Denoting $\boldsymbol{g}_t$ as the gradient at time t, the momentum update is

$$\boldsymbol{u}_{t+1} = \gamma \boldsymbol{u}_t + (1-\gamma)\big(-\boldsymbol{g}_t\big) \tag{14}$$

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \eta \boldsymbol{u}_{t+1}. \tag{15}$$

effectively, we treat the parameter $\boldsymbol{w}$ as the location of a particle with mass and treat the objective as a potential field generating forces on the particle. The amount of mass affects its perturbation response to the force field, so larger mass results in smoother motion.

Section A below shows that this update is equivalent to an update that can be viewed as an exponential smoother:

$$\boldsymbol{u}_{t+1} = \boldsymbol{u}_t - \eta \boldsymbol{g}_t, \tag{16}$$

$$\boldsymbol{w}_{t+1} = \gamma \boldsymbol{w}_t + (1-\gamma)\boldsymbol{u}_{t+1}. \tag{17}$$

Note that $\boldsymbol{g}_t$ is still being evaluated at $\boldsymbol{w}_t$, so it's not exactly equivalent to running simply a smoother on gradient descent (gradients in our case come from evaluations at smoothed points), but it's similar.

This latter interpretation is nice because it shows that we're taking the gradients and 1. literally smoothing them over time, and 2. effectively operating on a slower time scale. That second point is important: this technique works because the time scale of the changing system across motions generated by the acceleration policies is fundamentally slower than that of the controller. This enables the controller (online learning) to use hundreds or even thousands of examples to adjust to new changes as it moves between different areas of the configuration space.

Another, common trick found in the machine learning literature (especially recently due to it's utility in deep learning training), is to scale the space by the observed variance of the error signal. When the error signal has high variance in a given dimension, the length scale of variation is smaller (small perturbations result in large changes). In that case, the step size should decrease. Similarly, when the observed variance is small, we can increase the step size to some maximal value. In our case, we care primarily about variance in the actual accelerations $\ddot{\boldsymbol{q}}_a$ (which measures the baseline noise, too, in the estimates) since we can assume the desired acceleration $\ddot{\boldsymbol{q}}_d$ signal is changing only slowly relative to the 1ms control loop. Denoting this $\ddot{\boldsymbol{q}}_a$ variance estimate as $\boldsymbol{v}_t$ we scale the update as

$$\boldsymbol{u}_{t+1} = \boldsymbol{u}_t - \big(\boldsymbol{I} + \alpha\ \mathbf{diag}(\boldsymbol{v}_t)\big)^{-1}\boldsymbol{g}_t. \tag{18}$$

Note that this is equivalent to using an estimated metric or Hessian approximation of the form $\boldsymbol{A} = \boldsymbol{I} + \alpha\ \mathbf{diag}(\boldsymbol{v}_t)$.

This combination of exponential smoothing (momentum) and a space metric built on an estimate of variance results in a smoothly changing $\boldsymbol{w}$ that's still able to track changes in the dynamics model errors.

4.2 Adaptive tuning of the forgetting factor

Indirect approaches to adaptive control (essentially online regressions of linear dynamics models) often tune their forgetting factor based on the magnitude of the error they're seeing. Larger errors mean that the previous model is bad and we should forget it fast to best leverage the latest data. Smaller errors mean that we're doing pretty well, and we should use as much data as possible to fully converge to zero tracking error. Adaptively tuning the forgetting factor, which manifests as adaptive tuning of the regularizer in our case, enables fast response to new modeling errors while simultaneously promoting accurate convergence to meticulous manipulation targets. For controlling the end-effector to a fixed Cartesian point the forgetting factor converges to 1 (no forgetting; zero regularization) and within a couple second and we quickly achieve accuracies of around 10^{-5}.

4.3 Handling noisy acceleration measurements

Section 4.1 described the tools we use from the machine learning (especially stochastic gradient descent) literature to reduce oscillations. But additionally, since handling noisy data is a fundamental problem to machine learning in general, these same tools enable us to handle noisy acceleration measurements. Momentum acts as a damper to the forces generated by the objective. The interpretation as an exponential smoother shows that white noise in the estimates cancels over time averaging to a more clear acceleration signal.

Additionally, we get 1000 training points per second. Physically the robot doesn't move very far in a second, and we can assume that the errors in the dynamics model will be changing at a time scale of tenths of a second, .1, .5, or even 1 second. That's anywhere between 100 and 1000 training examples available to track how errors in the dynamics model change as the robot executes its policy. That's plenty of data to average out noise and run a sufficient number of gradient descent iterations.

5 Direct vs indirect adaptive control: How does this differ from traditional online dynamics learning?

Traditional methods of online dynamics learning are often categorized as *indirect* adaptive control in the literature. The dynamics parameters affect the performance of the controller, but rather than adjusting the performance of the controller directly, these methods take streams of dynamics data and adjust the accuracy of the dynamics model under the understanding that accurate dynamics models should lead to accurate tracking. However, this section argues that the inverse dynamics problem is actually fairly difficult when frictions and especially stictions are involved since even invertible rigid body models end up being many to one forward models under stictions, *and* the data we get back from the system is actually slightly off policy. And we argue that the above outlined *direct* adaptive control approach, which directly attempts to minimize the error between desired and actual accelerations, is a more straightforward signal to train on.

Consider the situation of starting the online learning process from zero motion. With zero motion, stictions are problematic. That means there are may torques that we might apply that are too small. All of these torques produce zero acceleration, and therefore the forward dynamics mapping becomes many to one and, therefore, non-invertible. Intuitively, this problem manifests as a learning process that needs to train the inverse model to adjust it's prediction of the torque needed to produce zero acceleration over a period of a number of trials that all produce zero accelerations (hopefully with increasing torque commands) until the predicted torque finally breaks through the stiction barrier to produce a nonzero acceleration target to train on.

More concretely, consider an objective of the form:

$$l_{\boldsymbol{\tau}}(\boldsymbol{w}) = \left\|\boldsymbol{\tau}_t - f(\boldsymbol{q}_t, \dot{\boldsymbol{q}}_t, \ddot{\boldsymbol{q}}_t, \boldsymbol{w})\right\|^2, \tag{19}$$

where f is any class of function approximators parameterized by $\boldsymbol{w}$ predicting the inverse dynamics. In this case, $\boldsymbol{\tau}_t$ is the actual applied torques, and $\ddot{\boldsymbol{q}}_t$ is the corresponding actual accelerations that were observed. For instance, we might use just a simple torque offset $\boldsymbol{w}$ from the rigid body inverse dynamics model, paralleling what we outlined above:

$$\begin{align} f(\boldsymbol{q}_t, \dot{\boldsymbol{q}}_t, \ddot{\boldsymbol{q}}_t, \boldsymbol{w}) &= \left(\widehat{\boldsymbol{M}}\ddot{\boldsymbol{q}}_d^{(t)} + \widehat{\boldsymbol{h}}(\boldsymbol{q}_t, \dot{\boldsymbol{q}}_t)\right) + \boldsymbol{w} \tag{20}\\ &= \widehat{\boldsymbol{\tau}}_t + \boldsymbol{w}. \tag{21}\end{align}$$

Here we denote $\widehat{\boldsymbol{\tau}}_t = \widehat{\boldsymbol{M}}\ddot{\boldsymbol{q}}_d^{(t)} + \widehat{\boldsymbol{h}}(\boldsymbol{q}_t, \dot{\boldsymbol{q}}_t)$ as the underlying rigid body inverse dynamics prediction. The gradient of this expression is just the torque error:

$$\nabla l_{\boldsymbol{\tau}}(\boldsymbol{w}) = -\left(\boldsymbol{\tau}_t - (\widehat{\boldsymbol{\tau}}_t + \boldsymbol{w})\right). \tag{22}$$

That seems natural enough. But this expression is actually operating on a different set of data. This algorithm attempts to make the model's predicted torque on the *actual* observed acceleration more accurate. But those observed accelerations are the wrong accelerations. We didn't want them. We wanted the desired accelerations. We want the model to be accurate on the desired accelerations, but we're fixing it on the observed accelerations.

This discrepancy is particularly problematic when stictions are involved: all actual observed accelerations are zero when we aren't applying enough torque. The breakdown of these terms is as follows: $\boldsymbol{\tau}_t$ is the torque we actually applied. The acceleration we actually achieved is $\ddot{\boldsymbol{q}}_t$. If we push that back through the inverse dynamics, we get $f(\boldsymbol{q}_t, \dot{\boldsymbol{q}}_t, \ddot{\boldsymbol{q}}_t, \boldsymbol{w}) = \widehat{\boldsymbol{\tau}}_t + \boldsymbol{w}$. It should have predicted $\boldsymbol{\tau}_t$ at that acceleration point, so a gradient descent algorithm updates the model accordingly.

While the direct approach tries to explicitly find the torque that will produce a non-zero desired acceleration (if we don't see the desired acceleration, then just adjust the torque until we do), this indirect approach (for this stiction example) requires a feedback sequence that keeps updating the torque predicted for zero accelerations to (hopefully) higher and higher values until we successfully start seeing data with non-zero acceleration. The hope is that we're actually increasing those torques over time. The basic assumption is that non-zero desired accelerations under our rigid body dynamics model will always predict higher torques (assuming positive desired accelerations) than zero accelerations would. So those non-zero desired accelerations predict a torque that in turn creates and actual acceleration (which would be zero at first before breaking through the stiction barrier) that in turn predicts a smaller torque under our erroneous model. The algorithm then updates the predictions for zero accelerations to be higher and that in turn increases the prediction for the desired acceleration so that the next cycle actually applies higher torque.

The direct approach, on the other hand, tries to directly minimize the error in accelerations, whereas the indirect approach updates the model to predict that a larger torque is actually the thing that generates the observed accelerations. The hope is that iterating that process reduces tracking error (the error between desired and actual accelerations), but the analysis of this process is difficult.[1]

6 A note on step size gains and an analogy to PD gains

The larger the step size, the quicker the adaptive control strategy adjusts to errors between desired and actual accelerations. That means it will fight physical perturbations of the system stronger with larger step size. To accurately track desired accelerations, we either need large step sizes for fast adaptation, or a good underlying dynamics model. If we have a really good dynamics model, we can get away with smaller That means the better the dynamics model, the easier physical interaction with the robot becomes. Bad models require larger step sizes which manifests as a feeling of "tightness" in the robot's joints. We can always push the robot around, and it'll always follow its underlying acceleration policy from wherever it ends up, but the more we rely on the adaptive control techniques, the tighter the robot becomes and the more force we need to apply to physically push it around.

This behavior parallels the trade-offs we see in the choice of PD gains for trajectory following. Bad (or no) dynamics models require hefty PD gains, which means it can be near impossible to perturb the robot off it's path. But the better the dynamics model, the better the feedforward term is, and the looser we can make the PD gains while still achieving good tracking. We're able to push the robot around more easily (and it's safer). The difference is whether or not we need that trajectory. The proposed direct adaptive control method attempts to follow the desired acceleration policy well, which means when we perturb it, it always continues from where it finds itself rather than attempting in any sense to return to a predefined trajectory or integral curve.

7 A simple simulated experiment

This experiment shows a simple 2D example of the behavior of the adaptive control system for a scenario where the dynamics model used by the robot differs drastically from the true dynamics and where unmodeled nonlinear frictions are significant.

The true mass matrix of the system is $\boldsymbol{M}(\boldsymbol{q}) = 5\big(\boldsymbol{v}(\boldsymbol{q})\boldsymbol{v}(\boldsymbol{q})' + .05\boldsymbol{I}\big)$, where $\boldsymbol{v}(\boldsymbol{q}) = (\sin(5q_1); \cos(2q_2))$. The robot uses a diagonal constant approximation of the form $\widehat{\boldsymbol{M}} = .5\boldsymbol{I}$, which is assumes masses that are an order of magnitude smaller. Additionally, the true

[1]I believe the adaptive control literature has shown convergence in some cases for the indirect approach, especially for simple models. But the error metric under the direct algorithm is much more straightforward to interpret; my belief is that the convergence rate of the indirect method (if it converges) will be slower, but that's an experimental question. From a systems perspective, the combined simplicity and real-world efficacy of the direct method make it a strong tool for addressing the issues outlined in this document.

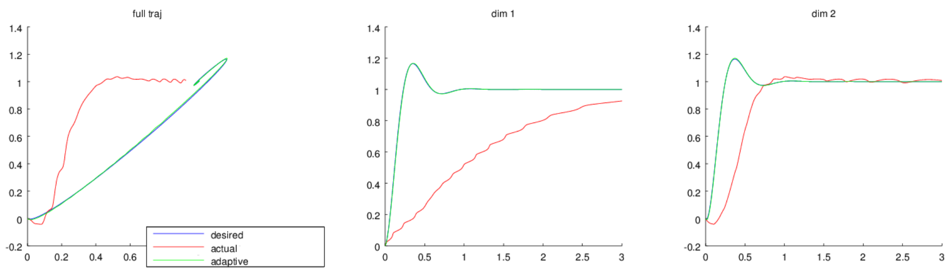

Figure 1: 2D simulation of online adaptation to severe model inaccuracies and underlying friction (which varies sinusoidally with each dimension in this case). The time axes (lower axis of the second and third plots) are in units of seconds, and all spacial axes are in units of meters. The controller updated at a rate of 1kHz.

system experiences sinusoidal frictional forces of the form

$$\mu(\boldsymbol{q}) = \begin{bmatrix} 100\sin(50\ q_1) \\ 5\sin(50\ q_2) \end{bmatrix}, \tag{23}$$

which the robot has no knowledge of.

The system is using a simple PD controller (outputting accelerations) to move to a desired fixed point to a target velocity of zero.

$$\ddot{\boldsymbol{q}} = K(\boldsymbol{q}_d - \boldsymbol{q}) - D\dot{\boldsymbol{q}}, \tag{24}$$

where $K = 100.$ and $D = 10.$. The desired target is $\boldsymbol{q}_d = (1,1)$, and the robot starts from $\boldsymbol{q}_0 = (0,0)$ with a random initial velocity.

The acceleration controller itself is tuned incorrectly and therefore overshoots. However, that's the desired behavior we want to track (shown in blue). If we simply pipe the desired accelerations through the very approximate inverse dynamics model, the behavior we get is abysmal (shown in red). On the one hand the dynamics model is a severe approximation, and on the other hand, we have no advanced knowledge of the strange frictional pattern, so the robot drifts upward and then oscillates during the final approach. But when we turn on adaptation (shown in green), the system is able to compensate for all of that and we get very good tracking behavior.

This implementation didn't simulate noise. But the real-world implementation discussed in the next section shows the efficacy of the above described learning tricks under noisy acceleration estimates sent back by the physical robot.

8 Real-world experiments

We have a full implementation of this algorithm working for the Apollo platform for both simple Cartesian space controllers and for a full continuous optimization (MPC-type) motion generation system. The underlying adaptive control parameterization is the same in all cases—we tune once and that enables the execution of any number of acceleration policies.

The following three videos show examples of the dynamics adaptation algorithm running on the Apollo manipulation platform. In the first two videos, Apollo is directly executing an acceleration policy designed to generate Cartesian motion moving his finger to a fixed point in space. In the final video, Apollo is executing optimized grasping behaviors.

1. **Robustness to perturbation:** `https://youtu.be/clldz75ToVI`
   In this video, Apollo is repeatedly perturbed away from the fixed point and allowed to return under the control of the acceleration policy. The behavior is shaped by the desired accelerations, which Apollo is able to accurately reproduce by running the online learning adaptive control method outlined above to track how the dynamics model error shifts throughout the execution.

2. **Bounce tests:** `https://youtu.be/m0i5oHQeqA8`
   In this video, Apollo is put through a series of more aggressive bounce tests in an attempt to incite oscillation modes. The robustness measures outlined above successfully combat the maneuvers and Apollo's behavior remains natural throughout the attempt.

3. **Grasp tasks:** `https://youtu.be/LQABeK2IO80`
   In this video, Apollo is executing grasp task motions optimized on on the fly. Each motion is sent down to control as a sequence of affine acceleration policies which are directly executed using the online learning adaptive control methodology described in this document. No vision is used; the sequence of object locations is planned out in advance. The system, however, uses force control in the grasps (reading from the fingertip's strain gauges) to be robust to variations in size and specific positioning of the objects.

9 Generalization: Interpreting PID terms as online learning

The true acceleration of a system can be described as

$$\ddot{\boldsymbol{q}}_a(\boldsymbol{w}) = \boldsymbol{M}^{-1}\Big(\big(\widehat{\boldsymbol{M}}\ddot{\boldsymbol{q}}_d + \widehat{\boldsymbol{h}}\big) + \boldsymbol{w} - \boldsymbol{h}\Big), \tag{25}$$

where $\ddot{\boldsymbol{q}}_a$ is the actual acceleration the system undergoes under the torques calculated by an estimated inverse dynamics model $\widetilde{\boldsymbol{\tau}} = \widehat{\boldsymbol{M}}\ddot{\boldsymbol{q}}_d + \widehat{\boldsymbol{h}}$ with torque offset $\boldsymbol{w}$. The true dynamics of the classical mechanical system (represented under the Lagrangian mechanics framework) is $\ddot{\boldsymbol{q}}_a = \boldsymbol{M}^{-1}(\boldsymbol{\tau} - \boldsymbol{h})$, which is an inversion of the common form $\boldsymbol{\tau} = \boldsymbol{M}\ddot{\boldsymbol{q}}_a + \boldsymbol{h}$. Note that because we representing the applied torque $\boldsymbol{\tau}_{\text{applied}} = \big(\widehat{\boldsymbol{M}}\ddot{\boldsymbol{q}}_d + \widehat{\boldsymbol{h}}\big) + \boldsymbol{w}$ as an inverse dynamics term plus the offset, this model automatically is using a feed forward term: $\boldsymbol{\tau}_{\text{ff}} = \widehat{\boldsymbol{M}}\ddot{\boldsymbol{q}}_d + \widehat{\boldsymbol{h}}$. The learning technique is finding $\boldsymbol{w}$ as an offset to that over time. Note that we would have equivalently used simply $\boldsymbol{\tau}_{\text{applied}} = \boldsymbol{w}$ so that $\boldsymbol{w}$ represented the entire applied torque. That case would be equivalent to not using the feed forward term.

We can rewrite PID control as

$$\boldsymbol{\tau} = \boldsymbol{\tau}_{\text{ff}} + \alpha(\boldsymbol{q}_d - \boldsymbol{q}) + \beta(\dot{\boldsymbol{q}}_d - \dot{\boldsymbol{q}}) + \gamma \int (\boldsymbol{q}_d - \boldsymbol{q})\ dt, \tag{26}$$

so we have three terms to explain: the position integral, the position error, and the velocity error. We'll start with the position integral term and progress to the position and velocity error terms.

9.1 Position integral term

If we desire to be at $\boldsymbol{q}_d$, and are currently at $\boldsymbol{q}_{t-1}$, we can choose an acceleration for this current time step to try to minimize that difference. The position the system will be in after Δt seconds executing an acceleration $\ddot{\boldsymbol{q}}$ is

$$\boldsymbol{q}_{t+1} = \boldsymbol{q}_t + \Delta t\ \dot{\boldsymbol{q}}_t + \frac{1}{2}\Delta t^2 \ddot{\boldsymbol{q}}. \tag{27}$$

Thus, if $\ddot{\boldsymbol{q}}_a(\boldsymbol{w}_t)$ defines the actual realized acceleration applied at time step t to get the system to time step $t+1$, we can write the resulting location

$$\boldsymbol{q}_{t+1}(\boldsymbol{w}_t) = \boldsymbol{q}_t + \Delta t\ \dot{\boldsymbol{q}}_t + \frac{1}{2}\Delta t^2 \ddot{\boldsymbol{q}}_a(\boldsymbol{w}_t), \tag{28}$$

as a function of the current weight vector. Given a desired position signal $\boldsymbol{q}_t^d$, we can write down an objective measuring the discrepancy between where we are at time t and where we wanted to be:

$$l_t^{\text{pos}}(\boldsymbol{w}_t) = \frac{1}{2}\|\boldsymbol{q}_{t+1}^d - \boldsymbol{q}_{t+1}(\boldsymbol{w}_t)\|^2_{\boldsymbol{M}_t}, \tag{29}$$

where $\boldsymbol{M}_t$ is the mass matrix at time t which defines the real system dynamics at that time. The derivative is

$$\nabla_{\boldsymbol{w}} l_t^{\text{pos}}(\boldsymbol{w}_t) = -\left[\frac{\partial \boldsymbol{q}_{t+1}(\boldsymbol{w}_t)}{\partial \boldsymbol{w}}\right]^T \boldsymbol{M}_t\Big(\boldsymbol{q}_{t+1}^d - \boldsymbol{q}_{t+1}(\boldsymbol{w}_t)\Big) \tag{30}$$

$$\propto -\Big(\boldsymbol{q}_{t+1}^d - \boldsymbol{q}_{t+1}(\boldsymbol{w}_t)\Big), \tag{31}$$

since $\frac{\partial \boldsymbol{q}_{t+1}(\boldsymbol{w}_t)}{\partial \boldsymbol{w}} = \frac{1}{2}\Delta t^2 \boldsymbol{M}_t^{-1}$. Notice that this gradient is a difference of positions, so taking gradient steps in an online learning fashion gives the traditional integral term on the positions:

$$\sum_{t=0}^{T} \gamma l_t^{\text{pos}}(\boldsymbol{w}_t) = \gamma \sum_{t=1}^{T} \Big(\boldsymbol{q}_t^d - \boldsymbol{q}_t^a\Big). \tag{32}$$

In other words, the typical I term of a PID controller is gradient desent on the objective given by Equation 29.

9.2 Position error

Similarly, we can define the actual velocity at time $t+1$ in terms of the integrated velocity from time t under the true system acceleration $\ddot{\boldsymbol{q}}_t^a(\boldsymbol{w}_t)$ at that time:

$$\dot{\boldsymbol{q}}_{t+1}^a(\boldsymbol{w}_t) = \dot{\boldsymbol{q}}_t + \Delta t \ \ddot{\boldsymbol{q}}_t^a(\boldsymbol{w}_t). \tag{33}$$

The following velocity error objective

$$l_t^{\mathrm{vel}}(\boldsymbol{w}_t) = \frac{1}{2}\|\dot{\boldsymbol{q}}_{t+1}^d - \dot{\boldsymbol{q}}_{t+1}^a(\boldsymbol{w}_t)\|_{\boldsymbol{M}_t}^2, \tag{34}$$

therefore, has the desired derivative

$$\nabla_{\boldsymbol{w}} l_t^{\mathrm{vel}}(\boldsymbol{w}_t) = -\Big(\dot{\boldsymbol{q}}_{t+1}^d - \dot{\boldsymbol{q}}_{t+1}^a\Big). \tag{35}$$

Summing up the negative gradients over time (under constant step size of $\widetilde{\alpha} = \alpha\Delta t$), therefore, defines an Euler integration of these velocities, which is equivalent to a position difference:

$$\sum_{t=0}^{T} \widetilde{\alpha} \ \nabla_{\boldsymbol{w}} l_t^{\mathrm{vel}}(\boldsymbol{w}_t) = \alpha\left(\sum_{t=1}^{T} \Delta t \ \dot{\boldsymbol{q}}_t^d - \sum_{t=1}^{T} \Delta t \ \dot{\boldsymbol{q}}_t^a\right) \tag{36}$$

$$= \alpha\left(\boldsymbol{q}_T^d - \boldsymbol{q}_T^a\right). \tag{37}$$

9.3 Velocity error

And finally, we can do the same thing for acceleration error objectives:

$$l_t^{\mathrm{acc}}(\boldsymbol{w}_t) = \frac{1}{2}\|\ddot{\boldsymbol{q}}_t^d - \ddot{\boldsymbol{q}}_t^a(\boldsymbol{w}_t)\|_{\boldsymbol{M}_t}^2, \tag{38}$$

with gradient

$$\nabla_{\boldsymbol{w}} l_t^{\mathrm{acc}}(\boldsymbol{w}_t) = -\Big(\ddot{\boldsymbol{q}}_t^d - \ddot{\boldsymbol{q}}_t^a(\boldsymbol{w}_t)\Big). \tag{39}$$

Gradient descent with step size $\widetilde{\alpha} = \alpha\Delta t$ again defines an an Euler integration of these acceleration difference terms, which this time amounts to a position error:

$$\sum_{t=0}^{T} \widetilde{\alpha} \ \nabla_{\boldsymbol{w}} l_t^{\mathrm{acc}}(\boldsymbol{w}_t) = \alpha\left(\sum_{t=0}^{T} \Delta t \ \ddot{\boldsymbol{q}}_t^d - \sum_{t=0}^{T} \Delta t \ \ddot{\boldsymbol{q}}_t^a\right) \tag{40}$$

$$= \alpha\left(\dot{\boldsymbol{q}}_T^d - \dot{\boldsymbol{q}}_T^a\right). \tag{41}$$

9.4 The full PID objective

In short, PID control is online gradient descent on the objective

$$\begin{aligned} l_t(\boldsymbol{w}_t) &= l_t^{\mathrm{pos}}(\boldsymbol{w}_t) + l_t^{\mathrm{vel}}(\boldsymbol{w}_t) + l_t^{\mathrm{acc}}(\boldsymbol{w}_t) \\ &= \frac{1}{2}\|\boldsymbol{q}_{t+1}^d - \boldsymbol{q}_{t+1}^a(\boldsymbol{w}_t)\|_{\boldsymbol{M}_t}^2 + \frac{1}{2}\|\dot{\boldsymbol{q}}_{t+1}^d - \dot{\boldsymbol{q}}_{t+1}^a(\boldsymbol{w}_t)\|_{\boldsymbol{M}_t}^2 + \frac{1}{2}\|\ddot{\boldsymbol{q}}_t^d - \ddot{\boldsymbol{q}}_t^a(\boldsymbol{w}_t)\|_{\boldsymbol{M}_t}^2, \end{aligned}$$

since its negative gradient is

$$-\nabla_{\boldsymbol{w}} l_t(\boldsymbol{w}_t) = \left(\boldsymbol{q}^d_{t+1} - \boldsymbol{q}^a_{t+1}\right) + \left(\dot{\boldsymbol{q}}^d_{t+1} - \dot{\boldsymbol{q}}^a_{t+1}\right) + \left(\ddot{\boldsymbol{q}}^d_t - \ddot{\boldsymbol{q}}^a_t\right). \tag{42}$$

This perspective allows us to understand this method of control in the broader context of machine learning, specifically an online learning method explictly attempting to patch up the inverse dynamics model by running gradient descent on objective terms leveraging three sources of information: desired and actual positions, velocities, and accelerations.

10 What does all of this mean?

Industrial robotic systems use PID control extensively because it works. Precision is paramount, and in most cases, to combat the real world realities of inaccurate models, the PID gains need to be set so high that it obviates entirely the need for inverse dynamics. Initially, that sounds naïve. We know something about the system: it, for the most part, is a rigid body system, and we can compute it's dynamics extremely fast using the Recursive Newton-Euler algorithm. Why not use that? Surely we'd get better results.

But the analysis in this paper shows that the real reason why that's unnecessary is: machine learning works really well. The system is running an online learning algorithm to minimize position, velocity, and acceleration errors over time, and it's getting a stream of data from the robot, likely at the rate of 1000 points per second. Even for high-speed manipulators, that's (apparently—we have a lot of empirical evidence of this) plenty of data to adjust for the changing nonlinear dynamics model.

The above discussion enables us to understand existing real-world control methods as online learning, and it provided a useful framework for developing a novel adaptive control algorithm that enables accurate tracking of acceleration policies, but all of this discussion has centered around a very restrictive model of the dynamics offset. All of the above algorithms model that offset as just a single vector, with no generalization as a function of position, velocity, and desired acceleration. The real power of this connection to machine learning comes from the generalization beyond simple offset models to more sophisticated models, such as deep learners or linear models. In these cases, rather than defining applied torques as $\boldsymbol{\tau} = \widehat{\boldsymbol{M}}(\boldsymbol{q})\ddot{\boldsymbol{q}}_d + \widehat{\boldsymbol{h}}(\boldsymbol{q}, \dot{\boldsymbol{q}}) + \boldsymbol{w}$, using a constant ofset $\boldsymbol{w}$ as we did before. We simply replace $\boldsymbol{w}$ with a more sophisticated function approximator $f_{\text{offset}}(\boldsymbol{q}, \dot{\boldsymbol{q}}, \ddot{\boldsymbol{q}}_d, \boldsymbol{w})$ as a function of $\boldsymbol{q}$, $\dot{\boldsymbol{q}}$, and $\ddot{\boldsymbol{q}}$ as well as the parameter vector $\boldsymbol{w}$. The only difference is that now, instead of the gradient being just a difference between the desired and actual accelerations as given in Equation 11, the new gradient is

$$\nabla_{\boldsymbol{w}} f(\boldsymbol{w}) = -\boldsymbol{J}^T\left(\ddot{\boldsymbol{q}}_d - \ddot{\boldsymbol{q}}_a(\boldsymbol{w})\right), \tag{43}$$

where $\boldsymbol{J} = \frac{\partial}{\partial \boldsymbol{w}} f_{\text{offset}}$ is the Jacobian of the offset function. In other words, we just push the original gradient through the function approximator's Jacobian

This observation opens up an entirely new class of algorithms that calculate not only just the current torque to apply instantaneously, but an entire local model generalizing that information and telling us what torques to apply for nearby positions, velocities, and desired accelerations.

Appendix

A The relationship between momentum updates and exponential smoothing

Exponential smoothing is

$$\begin{aligned}\boldsymbol{g}_t &= \nabla f(\boldsymbol{w}_t^e)\\ \boldsymbol{w}_{t+t} &= \boldsymbol{w}_t - \eta \boldsymbol{g}_t\\ \boldsymbol{w}_{t+1}^e &= \gamma \boldsymbol{w}_t^e + (1-\gamma)\boldsymbol{w}_{t+1},\end{aligned}$$

where $\boldsymbol{w}_t$ is the sequence of weight vectors generated by the algorithm and $\boldsymbol{w}_t^e$ is the exponentially smoothed evaluation point. Similar in structure is the momentum update, which takes the form

$$\begin{aligned}\boldsymbol{g}_t &= \nabla f(\boldsymbol{w}_t^m)\\ \boldsymbol{u}_{t+1} &= \gamma \boldsymbol{u}_t + (1-\gamma)\boldsymbol{g}_t\\ \boldsymbol{w}_{t+1}^m &= \boldsymbol{w}_t^m - \eta \boldsymbol{u}_t.\end{aligned}$$

This time the gradient is evaluated at the momentum point and then smoothed with past gradients before the step is taken.

With some algebra, they both expand out to be of the form:

$$\boldsymbol{w}_{t+1}^e = \boldsymbol{w}_{t+1}^m = -\eta\Big(\ldots(1+\gamma+\gamma^2)g_{t-2} + (1+\gamma)g_{t-1} + g_t\Big). \tag{44}$$