Differential Dynamic Programming with Temporally Decomposed Dynamics

Akihiko Yamaguchi[1] and Christopher G. Atkeson[1]

***Abstract*— We explore a temporal decomposition of dynamics in order to enhance policy learning with unknown dynamics. There are model-free methods and model-based methods for policy learning with unknown dynamics, but both approaches have problems: in general, model-free methods have less generalization ability, while model-based methods are often limited by the assumed model structure or need to gather many samples to make models. We consider a temporal decomposition of dynamics to make learning models easier. To obtain a policy, we apply differential dynamic programming (DDP). A feature of our method is that we consider decomposed dynamics even when there is no action to be taken, which allows us to decompose dynamics more flexibly. Consequently learned dynamics become more accurate. Our DDP is a first-order gradient descent algorithm with a stochastic evaluation function. In DDP with learned models, typically there are many local maxima. In order to avoid them, we consider multiple criteria evaluation functions. In addition to the stochastic evaluation function, we use a reference value function. This method was verified with pouring simulation experiments where we created complicated dynamics. The results show that we can optimize actions with DDP while learning dynamics models.**

I. Introduction

Manipulation of non-rigid materials is an important challenge to make robots useful in the home. Pouring a range of materials, such as liquids, and granular particles, is a good example. Folding towels is another example [1]. A difficulty is that the behavior of such material is too complicated to make a precise model. Leaning methods are used in those cases, such as reinforcement learning (e.g. [2]). Kormushev *et al.* applied a reinforcement learning method to obtain a policy of flipping a pancake in a frying pan [3].

Yamaguchi *et al.* created a pouring behavior that can generalize well [4]. They used simple learning methods combined with planning methods to achieve generalization. They did a careful primitive skill design which was further improved by learning from practice methods. Pouring is still a difficult task. An example is shown in Fig. 1 where a robot is pouring granular material (dried peas) by shaking. Some peas spilled out of the container. The problem is due to the complicated dynamics of flow. Designing behaviors is very difficult even for professionals.

Therefore we consider a policy learning method for unknown dynamics. There are model-free methods (e.g. [2]) and model-based methods (e.g. [5]) for policy learning with unknown dynamics, but both approaches have problems: in general, model-free methods have less generalization ability, while model-based methods need to gather many samples to make dynamic models.

[1]A. Yamaguchi and C. G. Atkeson are with the Robotics Institute, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh PA 15213, United States `info@akihikoy.net`

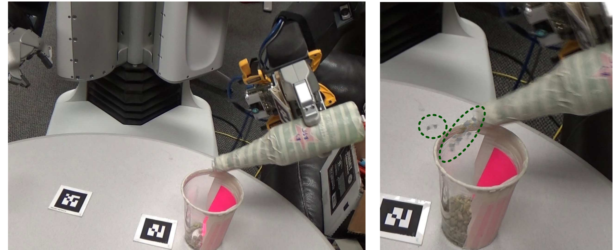


Fig. 1. PR2 pouring materials (dried peas) by shaking. Dotted circles in right image show flow; a little amount spilled out. See the accompanying video or: `https://youtu.be/OrjTHw0CHew`

We explore a temporal decomposition of dynamics to make learning dynamics easier. For example in pouring, the whole dynamics is complicated: the state is positions of containers, types of containers, and material type, the action is grasping parameters and flow-control parameters, and the output is material flow. But a small part of the dynamics, for example the relationship between the mouth position of the source container and the flow trajectory, is easier to model. We believe that humans have a model of ketchup flow since humans can pour ketchup correctly on a target surface. Thus we explore a method to learn temporally decomposed dynamics and plan a behavior with dynamic programming.

Since we are working with continuous actions, we use differential dynamic programming (DDP) which is a gradient based optimization algorithm. Since (1) learned models typically have modeling (prediction) error, and (2) flow is a probabilistic process, we consider probability distributions of states and an expectation of the evaluation function (i.e. stochastic DDP). For simplicity, we use a first-order gradient descent algorithm. Especially when using learned dynamics, there are many local maxima in the evaluation function, a DDP often converges to poor local maxima. In order to avoid them, we consider multiple criteria evaluation functions. In addition to the stochastic evaluation function, we use *reference value functions* made from *reference states*. The states of the system are constrained by the dynamics, but the reference states are optimized while ignoring those constraints. A reference value function is defined as a quadratic centered on a reference state. DDP with reference value functions performs rough optimization. A quadratic value function is easier to optimize, but the solution quality is not good due to dynamics not being taken into account to

obtain the reference states. On the other hand, DDP with the original stochastic evaluation function is useful for fine tuning. Combining them gives better solutions. In addition to this, we test multiple gradient candidates in each iteration of DDP to avoid getting stuck on saddle points.

A remarkable feature of our method is that we consider a dynamics stage even when an action is not taken, which allows us to decompose dynamics more flexibly. Reference states are also considered in such stages. For example, we consider where the flow center should be in the receiving container, which corresponds to a reference state in our method.

We use locally weighted regression (LWR) to learn the dynamics (process) models [6]. In order to obtain an adequate expectation, we provide a simple extension to locally weighted regression.

We applied our method to a pouring task on a dynamics simulator. We used a large bouncing parameter to make the flow less predictable. Through comparing several conditions, we found: (1) temporal decomposition of a process helps to learn dynamics, (2) value functions with reference states help to find a good solution especially when the process models are not learned, (3) our numerical method to compute an expectation with LWR is useful to treat uncertainty, (4) especially when the process models are not learned, a brute force dynamic programming using CMA-ES [7] outputs poor local maxima even with decomposed process models, and (5) multiple criteria optimization is helpful to avoid poor local maxima. Therefore we contribute practical methods for DDP with learned models, and show empirical benefits of dynamics decomposition.

Related Work

Regarding planning behaviors under totally or partially unknown dynamics, there are two approaches: a model-free approach and a model-based approach. In a model-based approach, we learn the dynamics of the system, and then we apply dynamic programming, such as differential dynamic programming [8]. Schaal *et al.* [5] used locally weighted regression to learn a dynamics model, and applied linear quadratic control, which was a successful approach to robot juggling. Morimoto *et al.* [9] used a similar approach to learn a dynamics model, then applied robust DDP where the system disturbances were considered. A problem with model-based approaches is that we need to gather many samples to construct a dynamics model which is especially problematic in higher dimensional systems and/or systems with complicated dynamics such as pouring. In contrast in a model-free approach, we do not learn dynamics models, but learn policies directly. Many reinforcement learning algorithms are proposed in this category [10], [11], [12], [3], [13]. These were successful in learning specific tasks such as flipping a pancake, a ball-in-cup task, and a crawling motion. However their issue is poor generalization ability compared to that of model-based methods, although several attempts are taken to increase generalization ability (e.g. [2]). Another interesting approach is a hybrid of model-based and model-free methods. A well-known architecture is Dyna [14]. Originally it was developed for a discrete state-action domain, and later a new version with a linear function approximator was developed for a continuous domain [15]. Recently Levine *et al.* [16] proposed a more practical approach where a trajectory optimization method for unknown dynamics is combined with a local linear model learned from samples.

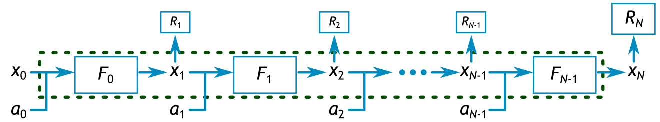


Fig. 2. Temporal decomposition considered in this paper. The dotted box denotes the whole process.

We introduce another insight into this model-based v.s. model-free argument: a temporal decomposition of dynamics. Learning dynamics becomes easier when the relationship between the input and output are simple. Levine's approach [16] makes learning dynamics easier by extracting a series of local dynamics (in both time and state space). We focus on a temporal decomposition. Typically in discrete time DDP, we divide dynamics where an action is taken. In this case dynamics models are considered in a form of $\mathbf{x}_{n+1} = \mathbf{F}_n(\mathbf{x}_n, \mathbf{a}_n)$, a mapping from state $\mathbf{x}_n$ and action $\mathbf{a}_n$ to a next state $\mathbf{x}_{n+1}$. We consider dynamics even when an action is not taken: i.e. we use $\mathbf{x}_{n+1} = \mathbf{F}_n(\mathbf{x}_n)$ as well as $\mathbf{x}_{n+1} = \mathbf{F}_n(\mathbf{x}_n, \mathbf{a}_n)$ in order to make the decomposition more flexible.

Among many DDP methods, our DDP is stochastic; that is, we consider probability distributions of states and expectations of rewards. This design of evaluation functions is similar to recent DDP methods (e.g. [17]). On the other hand, we use a simple gradient descent method to optimize actions, while traditional DDP [8] and recent methods (e.g. [18], [17]) use a second-order algorithm like Newton's method. Our DDP is inferior to second-order DDP algorithms regarding convergence speed. The quality of the solution would be the same as far as we use the same evaluation function. In addition, we introduce practical methods to increase the solution quality: multiple criteria evaluation functions with reference states and a gradient descent method with multiple gradient candidates.

II. Differential Dynamic Programming

We introduce our stochastic DDP with a first order gradient descent algorithm. We assume the dynamics are already decomposed and learned. We also introduce multiple criteria evaluation functions using reference states, and a gradient descent method with multiple gradient candidates.

A. Problem Formulation

We consider a system with N decomposed processes illustrated in Fig. 2. The input of n-th process is a state $\mathbf{x}_n$ and an action $\mathbf{a}_n$, and its output is the next state $\mathbf{x}_{n+1}$. Note that some actions may be omitted. A reward is given

to the outcome state. Let $\mathbf{F}_n$ denote the process model: $\mathbf{x}_{n+1} = \mathbf{F}_n(\mathbf{x}_n, \mathbf{a}_n)$, and R_n denote the reward model: $r_n = R_n(\mathbf{x}_n)$. Typically the most important reward is the final reward $R_N(\mathbf{x}_N)$ and $R_n(\mathbf{x}_n) = 0$ for $n < N$, but we consider a general form $R_n(\mathbf{x}_n)$ so that we can give rewards (or penalties) for intermediate states. We assume that every state is observable.

We use a regression model for each $\mathbf{F}_n$ learned from samples. We assume that the regression model gives the gradients $\partial\mathbf{F}_{\mathrm{X}n} = \frac{\partial \mathbf{F}_n}{\partial \mathbf{x}_n}$, $\partial\mathbf{F}_{\mathrm{A}n} = \frac{\partial \mathbf{F}_n}{\partial \mathbf{a}_n}$ for given $\mathbf{x}_n$, $\mathbf{a}_n$. In addition, for an input distribution $\mathbf{x}_n \sim \mathcal{N}(\bar{\mathbf{x}}_n, \mathbf{\Sigma}_n)$ and a deterministic $\mathbf{a}_n$, the regression model gives the output distribution $\mathbf{x}_{n+1} \sim \mathcal{N}(\bar{\mathbf{x}}_{n+1}, \mathbf{\Sigma}_{n+1})$ as $\bar{\mathbf{x}}_{n+1} = \tilde{\mathbf{F}}_n(\bar{\mathbf{x}}_n, \mathbf{\Sigma}_n, \mathbf{a}_n)$ and $\mathbf{\Sigma}_{n+1} = \mathbf{S}_n(\bar{\mathbf{x}}_n, \mathbf{\Sigma}_n, \mathbf{a}_n)$ (we use $\tilde{\mathbf{F}}_n(\mathbf{x}_n, \mathbf{a}_n)$ and $\mathbf{S}_n(\mathbf{x}_n, \mathbf{a}_n)$ to denote them shortly), and the gradients of $\tilde{\mathbf{F}}_n$ w.r.t. $\bar{\mathbf{x}}_n$ and $\mathbf{a}_n$ as $\partial\tilde{\mathbf{F}}_{\mathrm{X}n} = \frac{\partial \tilde{\mathbf{F}}_n}{\partial \bar{\mathbf{x}}_n}$, $\partial\tilde{\mathbf{F}}_{\mathrm{A}n} = \frac{\partial \tilde{\mathbf{F}}_n}{\partial \mathbf{a}_n}$.

The purpose of dynamic programming for the n-th process is to choose actions $\{\mathbf{a}_n, \dots, \mathbf{a}_{N-1}\}$ so that an evaluation function $J_n(\mathbf{x}_n, \{\mathbf{a}_n, \dots, \mathbf{a}_{N-1}\})$ is maximized. J_n is an expected sum of future rewards, defined as follows:

$$J_n(\mathbf{x}_n, \{\mathbf{a}_n, \dots, \mathbf{a}_{N-1}\}) = \mathrm{E}[\textstyle\sum_{n'=n+1}^{N} R_{n'}(\mathbf{x}_{n'})] = \textstyle\sum_{n'=n+1}^{N} \mathrm{E}[R_{n'}(\mathbf{x}_{n'})] = \textstyle\sum_{n'=n+1}^{N} \tilde{R}_{n'}, \quad (1)$$

where $\tilde{R}_{n'}(\mathbf{x}_{n'}) = \mathrm{E}[R_{n'}(\mathbf{x}_{n'})]$. The expectation is taken over $\{\mathbf{x}_n, \dots, \mathbf{x}_N\}$ where $\mathbf{x}_{n'} \sim \mathcal{N}(\bar{\mathbf{x}}_{n'}, \mathbf{\Sigma}_{n'})$. We also denote $\tilde{R}_{n'}(\mathbf{x}_{n'}) = \tilde{R}_{n'}(\bar{\mathbf{x}}_{n'}, \mathbf{\Sigma}_{n'})$. Note that we can use a different evaluation criteria by changing $\tilde{R}_{n'}(\mathbf{x}_{n'})$. For example, an upper confidence bound (UCB) is given by:

$$\tilde{R}_{n'}(\mathbf{x}_{n'}) = \mathrm{E}[R_{n'}(\mathbf{x}_{n'})] + f_{\mathrm{UCB}}\sqrt{\mathrm{var}[R_{n'}(\mathbf{x}_{n'})]}, \quad (2)$$

where f_{UCB} is a coefficient (such as 1).

B. Simulating the Processes

At the n-th process, we estimate future states and rewards from the state distribution $\mathbf{x}_n \sim \mathcal{N}(\bar{\mathbf{x}}_n, \mathbf{\Sigma}_n)$ and the actions $\{\mathbf{a}_n, \dots, \mathbf{a}_{N-1}\}$. The process models give $\mathbf{x}_{n+1} \sim \mathcal{N}(\bar{\mathbf{x}}_{n+1}, \mathbf{\Sigma}_{n+1}) = \mathcal{N}(\tilde{\mathbf{F}}_n(\mathbf{x}_n, \mathbf{a}_n), \mathbf{S}_n(\mathbf{x}_n, \mathbf{a}_n))$, so we obtain $\mathbf{x}_{n'} \sim \mathcal{N}(\bar{\mathbf{x}}_{n'}, \mathbf{\Sigma}_{n'})$ for $n' = n+1, \dots, N$.

In order to evaluate the expected reward, we consider quadratic models of reward functions. Consider a Taylor series expansion of $R_n(\mathbf{x}_n)$ around $\bar{\mathbf{x}}_n$: $R_n(\bar{\mathbf{x}}_n + \delta\mathbf{x}_n) \approx \delta\mathbf{x}_n^\top \mathbf{A}_n \delta\mathbf{x}_n + \mathbf{b}_n^\top \delta\mathbf{x}_n + R_n(\bar{\mathbf{x}}_n)$. Then the expectation and the variance of R_n are given by:

$$\mathrm{E}[R_n(\mathbf{x}_n)] = R_n(\bar{\mathbf{x}}_n) + \mathrm{Tr}(\mathbf{A}_n\mathbf{\Sigma}_n), \quad (3)$$

$$\mathrm{var}[R_n(\mathbf{x}_n)] = 2\mathrm{Tr}(\mathbf{A}_n\mathbf{\Sigma}_n\mathbf{A}_n\mathbf{\Sigma}_n) + \mathbf{b}_n^\top\mathbf{\Sigma}_n\mathbf{b}_n. \quad (4)$$

The Taylor series expansion is done in each iteration of DDP. Thus we can evaluate J_n of Eq. (1).

C. Gradient of J_n

We apply the gradient chain rule to obtain the gradient of J_n w.r.t. $\{\mathbf{a}_n, \dots, \mathbf{a}_{N-1}\}$. First we consider a case of $N = 2$. The evaluation function at $n = 0$ is given by:

$$J_0(\mathbf{x}_0, \{\mathbf{a}_0, \mathbf{a}_1\}) = \tilde{R}_2(\bar{\mathbf{x}}_2, \mathbf{\Sigma}_2) + \tilde{R}_1(\bar{\mathbf{x}}_1, \mathbf{\Sigma}_1) \quad (5)$$

$$= \tilde{R}_2(\tilde{\mathbf{F}}_1(\tilde{\mathbf{F}}_0(\bar{\mathbf{x}}_0, \mathbf{\Sigma}_0, \mathbf{a}_0), \mathbf{S}_0(\cdots), \mathbf{a}_1), \mathbf{S}_1(\cdots)) + \tilde{R}_1(\tilde{\mathbf{F}}_0(\bar{\mathbf{x}}_0, \mathbf{\Sigma}_0, \mathbf{a}_0), \mathbf{S}_0(\cdots)), \quad (6)$$

where the omitted arguments of $\mathbf{S}_n(\cdots)$ are the same as those of corresponding $\tilde{\mathbf{F}}_n$. We approximate the derivatives by assuming $\frac{\partial \mathbf{S}_n}{\partial \mathbf{a}_n} = \frac{\partial \mathbf{S}_n}{\partial \mathbf{a}_{n-1}} = \cdots = \mathbf{0}$. Then the derivatives of J_0 w.r.t. $\mathbf{a}_0$ and $\mathbf{a}_1$ are obtained by the chain rule:

$$\partial\mathbf{J}_{\mathrm{A}0} = \partial\tilde{\mathbf{F}}_{\mathrm{A}0}\partial\tilde{\mathbf{F}}_{\mathrm{X}1}\partial\tilde{\mathbf{R}}_{\mathrm{X}2} + \partial\tilde{\mathbf{F}}_{\mathrm{A}0}\partial\tilde{\mathbf{R}}_{\mathrm{X}1}, \quad (7)$$

$$\partial\mathbf{J}_{\mathrm{A}1} = \partial\tilde{\mathbf{F}}_{\mathrm{A}1}\partial\tilde{\mathbf{R}}_{\mathrm{X}2}, \quad (8)$$

where $\partial\tilde{\mathbf{R}}_{\mathrm{X}n} = \frac{\partial \tilde{R}_n}{\partial \mathbf{x}_n}$, and $\partial\mathbf{J}_{\mathrm{A}n} = \frac{\partial J_n}{\partial \mathbf{a}_n}$. Note that since $\mathbf{a}_n$ affects only $J_{n'}$ $(n' \geq n)$, $\frac{\partial J_0}{\partial \mathbf{a}_n} = \frac{\partial J_n}{\partial \mathbf{a}_n} = \partial\mathbf{J}_{\mathrm{A}n}$. In a general case $N = N$, we start the calculation from $\mathbf{a}_n = \mathbf{a}_{N-1}$:

$$\partial\mathbf{J}_{\mathrm{A}N-1} = \partial\tilde{\mathbf{F}}_{\mathrm{A}N-1}\partial\tilde{\mathbf{R}}_{\mathrm{X}N},$$

$$\partial\mathbf{J}_{\mathrm{A}N-2} = \partial\tilde{\mathbf{F}}_{\mathrm{A}N-2}\partial\tilde{\mathbf{F}}_{\mathrm{X}N-1}\partial\tilde{\mathbf{R}}_{\mathrm{X}N} + \partial\tilde{\mathbf{F}}_{\mathrm{A}N-2}\partial\tilde{\mathbf{R}}_{\mathrm{X}N-1},$$

$$\partial\mathbf{J}_{\mathrm{A}N-3} = \partial\tilde{\mathbf{F}}_{\mathrm{A}N-3}\partial\tilde{\mathbf{F}}_{\mathrm{X}N-2}\partial\tilde{\mathbf{F}}_{\mathrm{X}N-1}\partial\tilde{\mathbf{R}}_{\mathrm{X}N} + \partial\tilde{\mathbf{F}}_{\mathrm{A}N-3}\partial\tilde{\mathbf{F}}_{\mathrm{X}N-2}\partial\tilde{\mathbf{R}}_{\mathrm{X}N-1} + \partial\tilde{\mathbf{F}}_{\mathrm{A}N-3}\partial\tilde{\mathbf{R}}_{\mathrm{X}N-2},$$

$$\dots$$

$$\partial\mathbf{J}_{\mathrm{A}n} = \partial\tilde{\mathbf{F}}_{\mathrm{A}n}\mathbf{\Omega}_n, \quad (9)$$

$$\text{where} \quad \mathbf{\Omega}_n = \partial\tilde{\mathbf{F}}_{\mathrm{X}n+1}\mathbf{\Omega}_{n+1} + \partial\tilde{\mathbf{R}}_{\mathrm{X}n+1}, \quad \mathbf{\Omega}_N = \mathbf{0}. \quad (10)$$

Thus we can calculate the gradient of J_n w.r.t. $\{\mathbf{a}_n, \dots, \mathbf{a}_{N-1}\}$ by computing backwards from $n = N-1$. The derivatives $\partial\tilde{\mathbf{F}}_{\mathrm{X}n}$, $\partial\tilde{\mathbf{F}}_{\mathrm{A}n}$ are given by the process models. The derivative $\partial\tilde{\mathbf{R}}_{\mathrm{X}n}$ is obtained from a local quadratic reward model (Taylor series expansion around $\bar{\mathbf{x}}_n$); $\partial\tilde{\mathbf{R}}_{\mathrm{X}n} = \mathbf{b}_n$.

D. Value Functions with Reference States

We assume a reference state at state n is given as $\mathbf{x}_n^*$. Then another optimization criterion is choosing an action $\mathbf{a}_n$ so that the next state $\mathbf{x}_{n+1}$ is close to $\mathbf{x}_{n+1}^*$. In this case, we do not need to consider the gradient chain. For a deterministic $\mathbf{x}_n$, we define a value function with a reference state as:

$$V_n(\mathbf{x}_n) = -(\mathbf{x}_n^* - \mathbf{x}_n)^\top \mathbf{W}_{\mathrm{rs}}(\mathbf{x}_n^* - \mathbf{x}_n), \quad (11)$$

where $\mathbf{W}_{\mathrm{rs}}$ is a weight matrix. Note that since $\mathbf{x}_n^*$ is optimized by considering $R_n(\mathbf{x}_n^*)$ and $V_{n+1}(\mathbf{x}_{n+1})$, $V_n(\mathbf{x}_n)$ doest not need to consider $R_n(\mathbf{x}_n)$. The expectation over a state distribution $\mathbf{x}_n \sim \mathcal{N}(\bar{\mathbf{x}}_n, \mathbf{\Sigma}_n)$ is $\mathrm{E}[V_n(\mathbf{x}_n)] = V_n(\bar{\mathbf{x}}_n) - \mathrm{Tr}(\mathbf{W}_{\mathrm{rs}}\mathbf{\Sigma}_n)$. With the value function criteria, we optimize $\mathbf{a}_n$ so that $\mathrm{E}[V_{n+1}(\mathbf{x}_{n+1})] = \mathrm{E}[V_{n+1}(\tilde{\mathbf{F}}_n(\mathbf{x}_n, \mathbf{a}_n))]$ is maximized. The gradient of $\mathrm{E}[V_{n+1}(\mathbf{x}_{n+1})]$ w.r.t. $\mathbf{a}_n$ is obtained by:

$$\frac{\partial \mathrm{E}[V_{n+1}]}{\partial \mathbf{a}_n} = \partial\tilde{\mathbf{F}}_{\mathrm{A}n}\frac{\partial \mathrm{E}[V_{n+1}]}{\partial \mathbf{x}_{n+1}} = \partial\tilde{\mathbf{F}}_{\mathrm{A}n}(2\mathbf{W}_{\mathrm{rs}}(\mathbf{x}_{n+1}^* - \bar{\mathbf{x}}_{n+1})),$$

where we also assumed $\frac{\partial \mathbf{S}_{n+1}}{\partial \mathbf{a}_n} = \mathbf{0}$. Thus, an action is updated only considering making the next state close to the reference state; i.e. further future states are not considered. We expect the numerical stability with such a gradient might be more than a gradient obtained by the chain rule. For multiple criteria optimization, we consider $\mathrm{E}[V_{n+1}(\mathbf{x}_{n+1})]$ as an alternative of J_n, and $\frac{\partial \mathrm{E}[V_{n+1}]}{\partial \mathbf{x}_{n+1}}$ as a corresponding $\mathbf{\Omega}_n$.

E. Gradient Descent Algorithm for Dynamic Programming

The core procedure of a gradient descent algorithm for dynamic programming is *StepGD* listed in Algorithm 1. *StepGD* updates current actions with multiple gradients. Since the gradients are computed backwards from $n = N-1$, *StepGD* calls itself recursively until n reaches $N-1$. Multiple gradients computed in different ways are considered. All gradients are tested and the one which most improves the evaluation function is actually used. We use a gradient of the evaluation function and two gradients of the value functions (best two reference states are used independently). In line A of Algorithm 1, $\{\mathbf{\Omega}_n\} \leftarrow \{\mathbf{b}_{n+1}, 2\mathbf{W}_{\mathrm{rs}}(\mathbf{x}^{*1}_{n+1} - \bar{\mathbf{x}}_{n+1}), 2\mathbf{W}_{\mathrm{rs}}(\mathbf{x}^{*2}_{n+1} - \bar{\mathbf{x}}_{n+1})\}$. In line B of Algorithm 1, $\{\mathbf{\Omega}_{n-1}\} \leftarrow \{\partial\tilde{\mathbf{F}}_{\mathrm{X}n}\mathbf{\Omega}_n + \mathbf{b}_n, 2\mathbf{W}_{\mathrm{rs}}(\mathbf{x}^{*1}_n - \bar{\mathbf{x}}_n), 2\mathbf{W}_{\mathrm{rs}}(\mathbf{x}^{*2}_n - \bar{\mathbf{x}}_n)\}$. The gradients are updated for each iteration. Note that regardless of whether we use the evaluation function or the value function as criteria, we use these gradients.

PlanGD is the higher level routine listed in Algorithm 2, which starts with an initial guess to generate a initial sequence of actions. Then it uses *StepGD* repeatedly until convergence. In the initial guess, we search for good action samples from a database storing every state-action-reward from past executions. Some additional samples are generated randomly if not many samples are found in the database. The best action sample is chosen from these samples as an initial value of actions.

PlanGD has two loops. The inside loop is to run *StepGD* repeatedly until it converges. The outside loop is to apply several different criteria. In early stage of planning when the actions are far from optimal, optimizing the actions w.r.t. value functions might be efficient because of the quadratic objective function. On the other hand, using the evaluation function as the optimization criterion may improve the quality of actions since this criterion is the true purpose of dynamic programming. Thus using several criteria must be better numerically. We use the value function criterion twice then use the evaluation function criterion several times[1]. In *PlanGD*, in order to avoid a poor local maxima, Gaussian noise is added to an action (line C of Algorithm 2).

F. Planning Reference States

We also apply a gradient descent algorithm to update the reference states and actions. Though the states are constrained in the actual processes, we optimize the states as well as the actions. Here we consider deterministic variables only. The objective criterion for optimizing a state $\mathbf{x}^*_n$ and an action $\mathbf{a}^*_n$ is maximizing the reward $R_n(\mathbf{x}^*_n)$ and making the next state close to a reference value. This objective is described as:

$$L_n(\mathbf{x}^*_n, \mathbf{a}^*_n) = R_n(\mathbf{x}^*_n) - (\mathbf{x}^*_{n+1} - \mathbf{F}_n(\mathbf{x}^*_n, \mathbf{a}^*_n))^\top \mathbf{W}_{\mathrm{rs}}(\mathbf{x}^*_{n+1} - \mathbf{F}_n(\mathbf{x}^*_n, \mathbf{a}^*_n)). \quad (12)$$

[1]The combination and repeat number of criteria are decided from preliminary experiments. Increasing the repeat number will increase the solution quality but also increase the computation time. The evaluation function criterion should be used last.

Algorithm 1: *StepGD*

Input: n, $\mathbf{x}_n \sim \mathcal{N}(\bar{\mathbf{x}}_n, \mathbf{\Sigma}_n)$, $\mathbf{a}_{n:N-1} = \{\mathbf{a}_n, \ldots, \mathbf{a}_{N-1}\}$

1: **if** $n = N-1$ **then**
A: Calculate $\{\mathbf{\Omega}_n\}$
3: **else**
4:   Get $\mathbf{x}_{n+1}$: $\bar{\mathbf{x}}_{n+1} \leftarrow \tilde{\mathbf{F}}_n(\mathbf{x}_n, \mathbf{a}_n)$, $\mathbf{\Sigma}_{n+1} \leftarrow \mathbf{S}_n(\mathbf{x}_n, \mathbf{a}_n)$
5:   $\mathbf{a}'_{n+1:N-1}, \{\mathbf{\Omega}_n\} \leftarrow$ *StepGD*$(n+1, \mathbf{x}_{n+1}, \mathbf{a}_{n+1:N-1})$
6: $e_{\mathrm{best}} \leftarrow -\infty$
7: **for each** $\mathbf{\Omega}_n$ in $\{\mathbf{\Omega}_n\}$ **do**
8:   $\mathbf{a} \leftarrow \mathbf{a}_n + \alpha\partial\tilde{\mathbf{F}}_{\mathrm{A}n}\mathbf{\Omega}_n$
9:   **if** $J_n(\mathbf{x}_n, \{\mathbf{a}, \mathbf{a}'_{n+1:N-1}\}) > e_{\mathrm{best}}$ **then**
10:     $\mathbf{a}'_n \leftarrow \mathbf{a}$
11:     $e_{\mathrm{best}} \leftarrow J_n(\mathbf{x}_n, \mathbf{a}'_{n:N-1})$
12: **if** $n > 0$ **then**
B: Calculate $\{\mathbf{\Omega}_{n-1}\}$
14:   **return** $\mathbf{a}'_{n:N-1}, \{\mathbf{\Omega}_{n-1}\}$
15: **else**
16:   **return** $\mathbf{a}'_{n:N-1}, \{\}$

Algorithm 2: *PlanGD*

Input: n, $\mathbf{x}_n \sim \mathcal{N}(\bar{\mathbf{x}}_n, \mathbf{\Sigma}_n)$, $\{criteria\}$

1: Initial guess: $\mathbf{a}_{n:N-1}$
2: $\mathbf{a}^{\mathrm{best}}_{n:N-1} \leftarrow \mathbf{a}_{n:N-1}$
3: **for each** $criteria_j$ **in** $\{criteria\}$ **do**
4:   **loop**
5:     $\mathbf{a}'_{n:N-1} \leftarrow$ *StepGD*$(n, \mathbf{x}_n, \mathbf{a}_{n:N-1})$
6:     **if** $J_n(\mathbf{x}_n, \mathbf{a}'_{n:N-1}) \leq J_n(\mathbf{x}_n, \mathbf{a}_{n:N-1})$ **then**
7:       **break**
8:     $\mathbf{a}_{n:N-1} \leftarrow \mathbf{a}'_{n:N-1}$
9:   **if** $J_n(\mathbf{x}_n, \mathbf{a}_{n:N-1}) > J_n(\mathbf{x}_n, \mathbf{a}^{\mathrm{best}}_{n:N-1})$ **then**
10:     $\mathbf{a}^{\mathrm{best}}_{n:N-1} \leftarrow \mathbf{a}_{n:N-1}$
11:   **if** $criteria_j = criteria_{j+1}$ **then**
C: $\mathbf{a}_{n:N-1} \leftarrow \mathbf{a}^{\mathrm{best}}_{n:N-1} + Gaussian\ noise$
13:   **else**
14:     $\mathbf{a}_{n:N-1} \leftarrow \mathbf{a}^{\mathrm{best}}_{n:N-1}$
15: **return** $\mathbf{a}^{\mathrm{best}}_{n:N-1}$

Algorithm 3: *StepGDRef*

Input: n, $\{\mathbf{x}^{*i}_{n'}, \mathbf{a}^{*i}_{n'}\}$ $(n' = n, \ldots, N,\ i = 1, \ldots)$

1: **if** $n = N$ **then**
2:   **for each** i **do**
3:     $\mathbf{x}^{*i}_n \leftarrow \mathbf{x}^{*i}_n + \alpha\partial\mathbf{R}_{\mathrm{X}n}$
4:   **return**
5: Call *StepGDRef* for $n+1$
6: **for each** i **do**
7:   $\mathbf{x}^{*i}_n \leftarrow \mathbf{x}^{*i}_n + \alpha(\partial\mathbf{R}_{\mathrm{X}n} + 2\partial\mathbf{F}_{\mathrm{X}n}\mathbf{W}_{\mathrm{rs}}(\mathbf{x}^{*i}_{n+1} - \mathbf{F}_n(\mathbf{x}^{*i}_n, \mathbf{a}^{*i}_n)))$
8:   $\mathbf{a}^{*i}_n \leftarrow \mathbf{a}^{*i}_n + 2\alpha\partial\mathbf{F}_{\mathrm{A}n}\mathbf{W}_{\mathrm{rs}}(\mathbf{x}^{*i}_{n+1} - \mathbf{F}_n(\mathbf{x}^{*i}_n, \mathbf{a}^{*i}_n))$

The second term becomes zero for $n = N$. The partial derivatives of L_n w.r.t. $\mathbf{x}^*_n$ and $\mathbf{a}^*_n$ are given by:

$$\frac{\partial L_n}{\partial \mathbf{x}^*_n} = \partial\mathbf{R}_{\mathrm{X}n} + 2\partial\mathbf{F}_{\mathrm{X}n}\mathbf{W}_{\mathrm{rs}}(\mathbf{x}^*_{n+1} - \mathbf{F}_n(\mathbf{x}^*_n, \mathbf{a}^*_n)), \quad (13)$$

$$\frac{\partial L_n}{\partial \mathbf{a}^*_n} = 2\partial\mathbf{F}_{\mathrm{A}n}\mathbf{W}_{\mathrm{rs}}(\mathbf{x}^*_{n+1} - \mathbf{F}_n(\mathbf{x}^*_n, \mathbf{a}^*_n)). \quad (14)$$

The process models give $\partial\mathbf{F}_{\mathrm{X}n}$, $\partial\mathbf{F}_{\mathrm{A}n}$, and a Taylor series expansion around the current reference state gives $\partial\mathbf{R}_{\mathrm{X}n}$.

In order to avoid poor local maxima, we use a multi point search. Each point is updated with a gradient descent method. Some search points that have smaller scores (L_n)

are replaced by newly generated points, since these points are probably getting stuck in poor local maxima. The new search points are generated from the database and randomly.

Algorithm 3 describes the one step update routine *StepGDRef*. Since the objective function includes the next state, *StepGDRef* calls itself recursively until n reaches $n = N$, then updates states and actions backwards.

G. Complete Action Selection

The complete action selection procedure consists of updating reference states and *PlanGD*. First we apply the replacement of search points of reference states and actions. In our experiments, we used 6 points for each state and action, three of them are replaced by database samples, and one of them is replaced by a random value[2]. Then we apply *StepGDRef* to update the reference states and actions. *StepGDRef* is repeated several times (e.g. 3). Finally we use *PlanGD* to obtain an action sequence. For exploration purpose in an on-line learning setting, we add a zero-mean Gaussian noise whose variance is proportional to the uncertainty of the reward $\sum \mathrm{var}[R_n(\mathbf{x}_n)]$.

III. Learning Process Models

In order to model each process, we use locally weighted regression (LWR) [6]. LWR is a memory-based method: for a query point, weights between every sample are calculated by a kernel, then a local model is obtained by a weighted regression. The kernel width is decided for each sample point: assigning a small value for dense points, and a large value for coarse points.

A. Prediction and Gradient

Let $\mathbf{x}_1, \dots, \mathbf{x}_M$ denote input sample vectors and $\mathbf{y}_1, \dots, \mathbf{y}_M$ corresponding output sample vectors. For a query point $\mathbf{x}$, an output $\mathbf{y}$ is obtained by:

$$\mathbf{y} = \boldsymbol{\beta}^\top \mathbf{x}, \tag{15}$$

$$\boldsymbol{\beta} = (\mathbf{X}^\top \mathbf{W} \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^\top \mathbf{W} \mathbf{Y}, \tag{16}$$

where $\mathbf{X} = [\mathbf{x}_1, \dots, \mathbf{x}_M]^\top$, $\mathbf{Y} = [\mathbf{y}_1, \dots, \mathbf{y}_M]^\top$, $\mathbf{W} = \mathrm{diag}(\phi(\mathbf{x}, \mathbf{x}_1, c_1), \dots, \phi(\mathbf{x}, \mathbf{x}_M, c_M))$, λ is a regularization parameter, $\mathbf{I}$ is an identity matrix, and $\phi(\mathbf{x}, \mathbf{x}_k, c_k)$ is a kernel function with a width c_k. Typically we append 1 to each input vector of both samples and a query in order to represent a constant term. We assume that the prediction error of LWR is a zero-mean normal distribution $\mathcal{N}(\mathbf{0}, \mathbf{Q}(\mathbf{x}))$ whose covariance matrix is given by:

$$\mathbf{Q}(\mathbf{x}) = (\mathbf{X}\boldsymbol{\beta} - \mathbf{Y})^\top \mathbf{W} (\mathbf{X}\boldsymbol{\beta} - \mathbf{Y}) / (\mathrm{Tr}(\mathbf{W})(1 - D/M)), \tag{17}$$

where D is a number of dimensions of $\mathbf{x}$.

In our problem setting, an input $\mathbf{x}$ of LWR is a concatenated vector of a state $\mathbf{x}_n$, an action $\mathbf{a}_n$, and 1: $\mathbf{x}^\top = [\mathbf{x}_n^\top, \mathbf{a}_n^\top, 1]$, and an output $\mathbf{y}$ is the next state $\mathbf{x}_{n+1}$. We use an LWR for each process. Training an LWR is simple: add a sample of an input and an output to a memory, and update the kernel widths.

[2]These values are chosen from preliminary experiments.

We ignore the dependency of $\mathbf{x}$ on $\boldsymbol{\beta}$ to obtain the gradient; i.e. the gradient around a query point is given by $\boldsymbol{\beta}$. $\boldsymbol{\beta}$ is a matrix consisting of gradients w.r.t. a state $\mathbf{x}_n$ and an action $\mathbf{a}_n$, and a constant term: $\boldsymbol{\beta}^\top = [\mathbf{F}_{\mathrm{X}n}, \mathbf{F}_{\mathrm{A}n}, \mathbf{F}_{0n}]$. Thus $\partial \mathbf{F}_{\mathrm{X}n} = \mathbf{F}_{\mathrm{X}n}^\top$ and $\partial \mathbf{F}_{\mathrm{A}n} = \mathbf{F}_{\mathrm{A}n}^\top$. For an input distribution $\mathbf{x}_n \sim \mathcal{N}(\bar{\mathbf{x}}_n, \boldsymbol{\Sigma}_n)$ and a deterministic $\mathbf{a}_n$, we compute the output distribution $\mathbf{x}_{n+1} \sim \mathcal{N}(\bar{\mathbf{x}}_{n+1}, \boldsymbol{\Sigma}_{n+1})$ with

$$\bar{\mathbf{x}}_{n+1} = \boldsymbol{\beta}^\top [\bar{\mathbf{x}}_n^\top, \mathbf{a}_n^\top, 1]^\top, \tag{18}$$

$$\boldsymbol{\Sigma}_{n+1} = \mathbf{F}_{\mathrm{X}n} \boldsymbol{\Sigma}_n \mathbf{F}_{\mathrm{X}n}^\top + \mathbf{Q}_n(\bar{\mathbf{x}}_n, \mathbf{a}_n), \tag{19}$$

where we considered a local linear model, and we used the MAP estimate $\mathbf{Q}_n(\bar{\mathbf{x}}_n, \mathbf{a}_n)$ although the covariance $\mathbf{Q}_n(\mathbf{x}_n, \mathbf{a}_n)$ depends on $\mathbf{x}_n$.

B. Improvement of Expectation

Computing the expectation using a local linear model (e.g. Eq. (18)) or a local quadratic model (e.g. Eq. (3)) sometimes does not work, for example when the original function is a step function: $f(x) = 1$ if $|x| < 0.5$ otherwise 0. For $x \sim \mathcal{N}(\bar{x}, 1)$, the expectation $\mathrm{E}[f(x)]$ computed with Eq. (18) or (3) will take 1 if $|x| < 0.5$, 0 if $|x| > 0.5$, and a strange value if $|x| \approx 0.5$. The true expectation smoothly changes from 0 to 1 around $|x| = 0.5$. Applying a numerical Taylor series expansion to $f(x)$ with a wider window will reduce this issue, but in order to obtain a good result, we need to obtain many samples to compute derivatives. A numerical sampling-based computation of the expectation is also time consuming.

We solve this issue with LWR. Our simple approach is that we use a Gaussian kernel with a diagonal covariance matrix $\mathrm{diag}(c_k^2, \dots, c_k^2)$ ($c_k \in \mathbb{R}$ is the width of the original kernel), and modify it with diagonal elements of $\mathrm{cov}[\mathbf{x}]$ where $\mathbf{x}$ is a multi dimensional query point. That is, we use a diagonal covariance matrix $\mathrm{diag}(c_k^2, \dots, c_k^2) + \mathrm{diag}(\mathrm{cov}[\mathbf{x}])$ to compute a weight of a query point w.r.t. each sample k. Finally we use Eq. (18) and (19) with the new $\boldsymbol{\beta}$.

This method has two advantages. One is that the computational complexity is just a single evaluation of LWR. The other is that the gradient $\boldsymbol{\beta}$ obtained with the modified covariance matrix also takes into account the uncertainty of a query point, i.e. $\mathrm{cov}[\mathbf{x}]$, which contributes a lot in our gradient descent dynamic programming.

Fig. 3 shows an example where $f(x) = 1$ if $|x| < 0.5$ otherwise 0. LWR is trained with 50 samples from $f(x)$ without noise. The other curves entitled with "E[f]" are expectations taken with variance 0.02 computed with three different methods: (1) $\mathrm{E}[f(x)]$ obtained numerically (sampling based), (2) $\mathrm{E}[f(x)]$ computed similarly with (3) where the Taylor series expansion is used numerically with a window $\sqrt{0.02}$, and (3) $\mathrm{E}[f(x)]$ computed with LWR considering the query uncertainty. (3) is closest to (1) (they are almost overlapping), which is showing that the above expectation improvement method works.

IV. Experiments

We verify our method in simulated experiments. Although our final goal is to improve actual robot performance, we use

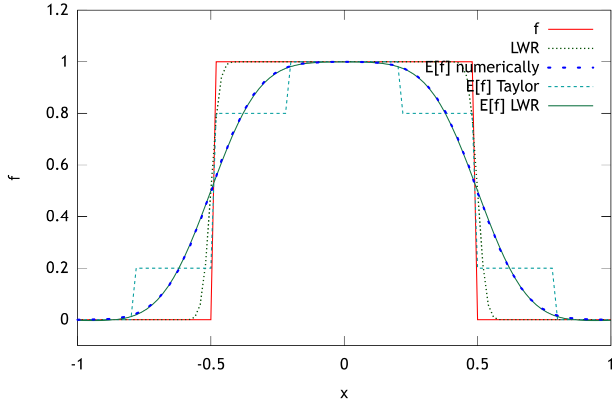


Fig. 3. Comparison of three calculations of $\mathrm{E}[f(x)]$.

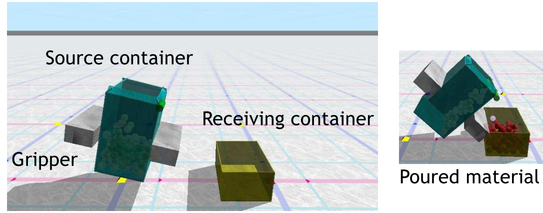


Fig. 4. Simulation environment. Right figure is after pouring material.

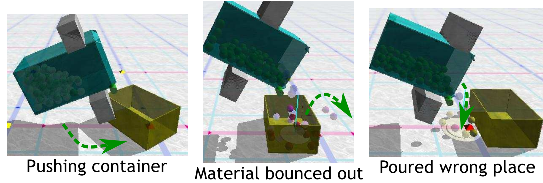


Fig. 5. Typical failures of pouring.

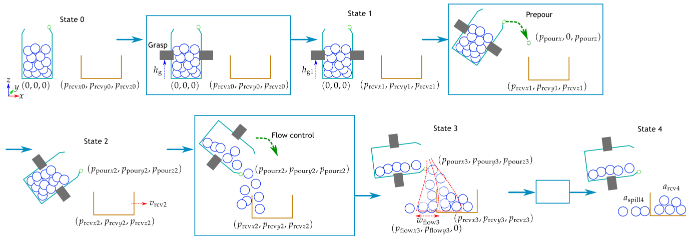


Fig. 6. Process model of pouring simulation experiments.

a simulator in order to compare various conditions.

We built a simulator using Open Dynamics Engine [19] where we simulate source and receiving containers, poured material, and a robot gripper grasping the source container (Fig. 4). We simulate poured material with many (100) spheres. The gripper is modeled as fixed blocks on the source container; we can change the grasping position, but it does not affect the grasp quality. This gripper possibly pushes the receiving container during pouring. In order to simulate the complicated behavior of material such as tomato sauce during shaking, we modified some contact model parameters; such as increasing the bouncing parameters. As a result, although the spheres are rigid objects, their trajectories during flow are complicated and sometimes unpredictable. Typical failure cases are shown in Fig. 5. See also the accompanying video or: `https://youtu.be/OrjTHw0CHew`

We design state machines for pouring, which is a simplified version of [4]. Those state machines have some parameters: a grasping height and pouring position. These parameters are planned.

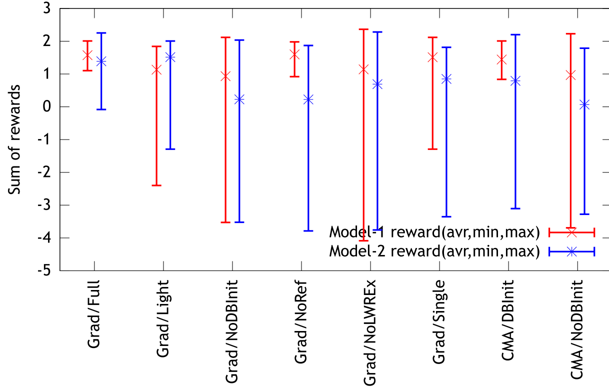
The process for planning is illustrated in Fig. 6. There are five decomposed processes: $\mathbf{x}_0 = (p_{\mathrm{rcv}x0}, p_{\mathrm{rcv}y0})$, $\mathbf{a}_0 = (h_{\mathrm{g}})$, $\mathbf{x}_1 = (p_{\mathrm{rcv}x1}, p_{\mathrm{rcv}y1}, h_{\mathrm{g}1})$, $\mathbf{a}_1 = (p_{\mathrm{pour}x}, p_{\mathrm{pour}z})$, $\mathbf{x}_2 = (\langle p_{\mathrm{rcv}x2} - p_{\mathrm{rcv}x1}\rangle, \langle p_{\mathrm{rcv}y2} - p_{\mathrm{rcv}y1}\rangle, \langle v_{\mathrm{rcv}2}\rangle, p_{\mathrm{pour}x2} - p_{\mathrm{rcv}x2}, p_{\mathrm{pour}z2} - p_{\mathrm{rcv}z2})$, $\mathbf{a}_2 = ()$, $\mathbf{x}_3 = (\langle p_{\mathrm{flow}x3} - p_{\mathrm{rcv}x3}\rangle, \langle p_{\mathrm{flow}y3} - p_{\mathrm{rcv}y3}\rangle, w_{\mathrm{flow}3}, p_{\mathrm{pour}z3} - p_{\mathrm{rcv}z3})$, $\mathbf{a}_3 = ()$, and $\mathbf{x}_4 = (a_{\mathrm{rcv}4}, -a_{\mathrm{spill}4})$, where $(p_{\mathrm{rcv}x}, p_{\mathrm{rcv}y}, p_{\mathrm{rcv}z})$ is the receiving container position, h_{g} is the grasping height, $(p_{\mathrm{pour}x}, p_{\mathrm{pour}y}, p_{\mathrm{pour}z})$ is the pouring position that the source container is rotated around during flow control, v_{rcv} is the speed of receiving container, $(p_{\mathrm{flow}x}, p_{\mathrm{flow}y}, 0)$ is the filtered flow position, w_{flow} is the filtered flow variance, a_{rcv} is the amount in the receiving container, and a_{spill} is the amount spilled out of the receiving container. The reason we are using relative values in $\mathbf{x}_2$ and $\mathbf{x}_3$ is to reduce the modeling complexity. We apply the arctangent function to some elements enclosed with $\langle\rangle$ since sometimes they become large, which causes undesirable learning results.

The reward consists of the amount $a_{\mathrm{rcv}4}$, the spilled penalty $-a_{\mathrm{spill}4}$, and the penalty for the movement of the receiving container $-(\langle p_{\mathrm{rcv}x2} - p_{\mathrm{rcv}x1}\rangle^2 + \langle p_{\mathrm{rcv}y2} - p_{\mathrm{rcv}y1}\rangle^2 + \langle v_{\mathrm{rcv}2}\rangle^2)$.

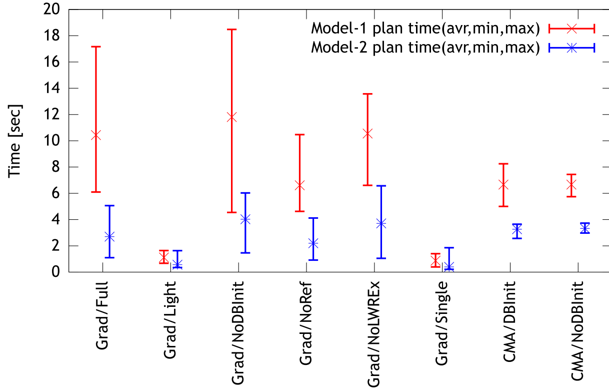
A. Dynamic Programming Comparison

First we compare the performance of DDP with parametric optimization. Here we use process models learned beforehand; a set of process models is well learned with 90 samples (Model-1) and another set is not well learned with 30 samples (Model-2). We apply our DDP with several conditions, and also apply CMA-ES [7] directly to optimize the evaluation function w.r.t. an action sequence. The conditions are: `Grad/Full`: our method with five criteria at the minimum (two value function based, and three evaluation function based), `Grad/Light`: our method with two criteria at the minimum (one value function based, and one evaluation function based), `Grad/NoLWREx`: `Grad/Full` without the expectation modification with LWR, `Grad/NoDBInit`: `Grad/Full` without the initial guess using a database, `Grad/NoRef`: `Grad/Full` without the reference states (no value function based update, and no gradient candidates using reference states), `Grad/Single`: our method with a single criterion (evaluation function based only), `CMA/DBInit`: brute force CMA-ES with the initial guess using the database, and `CMA/NoDBInit`: brute force CMA-ES without the initial guess using the database.

Fig. 7(a) and 7(b) show the result, acquired reward and planning time respectively. For each condition, we executed



(a) Comparison of reward (planning quality).



(b) Comparison of computation time.

Fig. 7. Results of dynamic programming with well-learned models (Model-1) and not well-learned models (Model-2). Average, maximum, and minimum values are plotted.

20 trials. Briefly, using well-learned process models (Model-1) is better than the other (Model-2). Using the database for an initial guess is useful. By comparing `Grad/Full` with other methods, we can find that: (1) value functions with reference states helps to find a better solution especially in the Model-2 case, (2) the expectation modification with LWR is useful to increase the stability to obtain the solutions, (3) CMA-ES [7] sometimes outputs poor local maxima especially in the Model-2 case.

Regarding the computation time, the gradient based methods with a small number of criteria are fast (around or less than one second). However, `Grad/Full` takes longer than CMA-ES. Since by reducing the number of criteria significantly decreases the computation time, we think we can optimize the criteria combination to make it faster than CMA-ES. The reason why Model-1 cases take a longer time than Model-2 cases is that the prediction with LWR depends on the number of samples.

B. On-line Learning

Next we explore an on-line setting. We start with no samples, and apply the action selection and updating alternately. We compare the following conditions and methods: `Sum/Grad`: our method with the sum of expected reward as the evaluation function, `Sum/Grad/NoNoise`: `Sum/Grad` without the exploration noise in the action selection, `Sum/CMA`: brute force CMA-ES with the sum of expected reward, and `UCB/Grad`: our method with the UCB as the evaluation function.

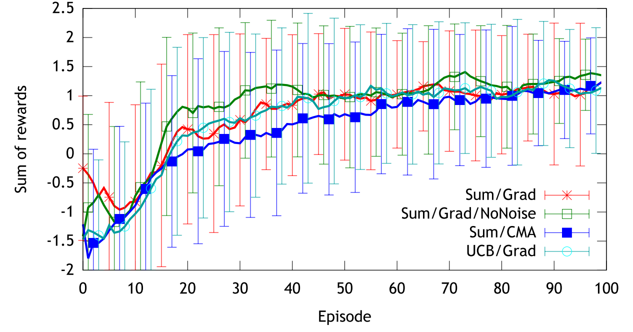


Fig. 8. Learning curves of on-line learning. Moving average filter with 10 episode window is applied.

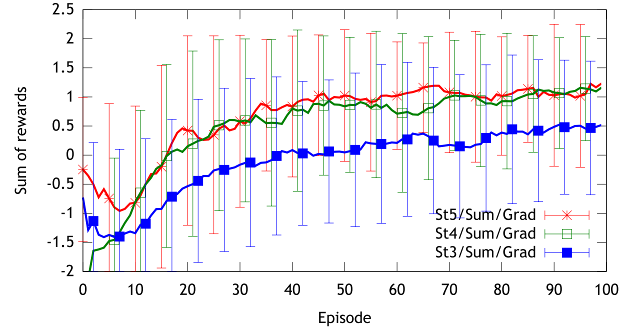


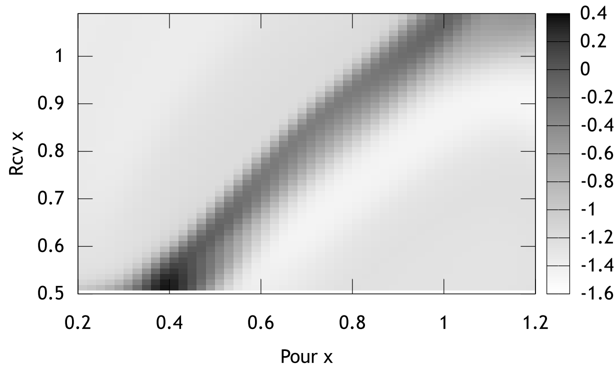
Fig. 9. Learning curves in different decompositions.

Fig. 8 shows the average learning curves of 10 trials. There are no significant difference, but in the early stages of learning (around 30 episodes), `Sum/Grad/NoNoise` is better than `Sum/CMA`. This result is the same as the previous section: especially when the process models are not learned well, a brute force CMA-ES goes to poor local maxima.

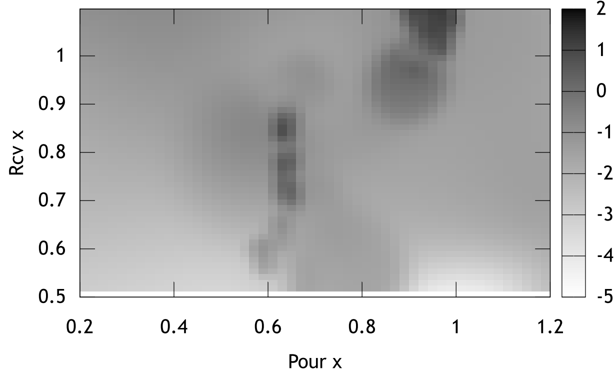
C. Comparison of Dynamics Decomposition

We investigate the effect of temporal decomposition of dynamics by comparing different patterns. We conduct an on-line setting using the same learning method, and compare the following decomposition patterns: `St5/Sum/Grad`: the same setup mentioned so far, `St4/Sum/Grad`: $\mathbf{x}_3$ is replaced by $(a_{\mathrm{rcv4}} - a_{\mathrm{spill4}})$ which is equivalent to the reward for the final state, and $\mathbf{x}_4$ is removed, `St3/Sum/Grad`: $\mathbf{x}_2$ is replaced by $(a_{\mathrm{rcv4}} - a_{\mathrm{spill4}} - (\langle p_{\mathrm{rcv}x2} - p_{\mathrm{rcv}x1}\rangle^2 + \langle p_{\mathrm{rcv}y2} - p_{\mathrm{rcv}y1}\rangle^2 + \langle v_{\mathrm{rcv2}}\rangle^2))$ which is equivalent to the total rewards, and $\mathbf{x}_3$ and $\mathbf{x}_4$ are removed.

Fig. 9 shows the learning curves averaging 10 trials. There is no significant difference between `St4` and `St5` since the process complexity does not change by removing the state $\mathbf{x}_3$. However `St3` has a big difference. The states $\mathbf{x}_2$, $\mathbf{x}_3$, and $\mathbf{x}_4$ of the original setup are not necessary to choose the actions $\mathbf{a}_0$ and $\mathbf{a}_1$ since only the final reward



(a) Using the result of `St5/Sum/Grad`.



(b) Using the result of `St3/Sum/Grad`.

Fig. 10. Plot of estimated evaluation at $\mathbf{x}_1$. Though actually the evaluation is computed for $\mathbf{x}_1 = (p_{\mathrm{rcv}x1}, p_{\mathrm{rcv}y1}, h_{\mathrm{g}1})$ and $\mathbf{a}_1 = (p_{\mathrm{pour}x}, p_{\mathrm{pour}z})$, we changed only $p_{\mathrm{rcv}x1}$ (`Rcv x`) and $p_{\mathrm{pour}x}$ (`Pour x`), and used a median value of samples for the others.

is important to optimize. These three conditions have the same dynamics regarding $\mathbf{x}_0$, $\mathbf{a}_0$, $\mathbf{x}_1$, $\mathbf{a}_1$, and rewards, thus if the learning methods were perfect, the results would be the same. Fig. 10 shows an estimated evaluation at $\mathbf{x}_1$ plotted for varying $p_{\mathrm{rcv}x1}$ (an element of the state $\mathbf{x}_1$) and $p_{\mathrm{pour}x}$ (an element of the action $\mathbf{a}_1$). $p_{\mathrm{pour}x}$ should take a proportional value to $p_{\mathrm{rcv}x1}$; `St5/Sum/Grad` could learn this, but `St3/Sum/Grad` could not. Thus, the bad performance of `St3` is caused by the inaccuracy of learned process models. This result empirically supports the idea of temporal decomposition of dynamics.

V. Conclusion

We explored a temporal decomposition of dynamics in order to enhance policy learning with unknown dynamics. We expected decomposed processes to be easier to learn, especially when we considered complicated dynamics, for example pouring a range of materials like water, ketchup, and sugar. To obtain a policy, we applied differential dynamic programming using a gradient decent method with multiple criteria. In order to take advantage of model-free methods, we used a reference value function, which is a quadratic centered on a reference state. Reference states are obtained through an optimization without considering dynamics constraints. A remarkable feature of our method is that we consider a dynamics decomposition even when an action is not taken, which allows us to decompose dynamics more flexibly. In order to learn process models, we used locally weighted regression with an improved expectation computation. To verify the method, we conducted simulation experiments where we used many spheres with high bouncing parameters to simulate a complicated flow. Despite the complexity of the dynamics, our method worked and we got good results.

Acknowledgment

We thank Professor Maxim Likhachev's Search-based Planning Lab in Carnegie Mellon University for making their PR2 robot available for our research.

References

[1] J. Maitin-Shepard, M. Cusumano-Towner, J. Lei, and P. Abbeel, "Cloth grasp point detection based on multiple-view geometric cues with application to robotic towel folding," in *the IEEE International Conference on Robotics and Automation (ICRA'10)*, 2010, pp. 2308–2315.

[2] J. Kober, A. Wilhelm, E. Oztop, and J. Peters, "Reinforcement learning to adjust parametrized motor primitives to new situations," *Autonomous Robots*, vol. 33, pp. 361–379, 2012.

[3] P. Kormushev, S. Calinon, and D. G. Caldwell, "Robot motor skill coordination with EM-based reinforcement learning," in *the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'10)*, 2010, pp. 3232–3237.

[4] A. Yamaguchi, C. G. Atkeson, and T. Ogasawara, "Pouring skills with planning and learning modeled from human demonstrations," *International Journal of Humanoid Robotics*, vol. 12, no. 3, p. 1550030, 2015.

[5] S. Schaal and C. Atkeson, "Robot juggling: implementation of memory-based learning," in *the IEEE International Conference on Robotics and Automation (ICRA'94)*, vol. 14, no. 1, 1994, pp. 57–71.

[6] C. G. Atkeson, A. W. Moore, and S. Schaal, "Locally weighted learning," *Artificial Intelligence Review*, vol. 11, pp. 11–73, 1997.

[7] N. Hansen, "The CMA evolution strategy: a comparing review," in *Towards a new evolutionary computation*, J. Lozano, P. Larrañaga, I. Inza, and E. Bengoetxea, Eds. Springer, 2006, vol. 192, pp. 75–102.

[8] D. Mayne, "A second-order gradient method for determining optimal trajectories of non-linear discrete-time systems," *International Journal of Control*, vol. 3, no. 1, pp. 85–95, 1966.

[9] J. Morimoto, G. Zeglin, and C. Atkeson, "Minimax differential dynamic programming: Application to a biped walking robot," in *the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'03)*, vol. 2, 2003, pp. 1927–1932.

[10] J. Kober and J. Peters, "Policy search for motor primitives in robotics," *Machine Learning*, vol. 84, no. 1-2, pp. 171–203, 2011.

[11] E. Theodorou, J. Buchli, and S. Schaal, "Reinforcement learning of motor skills in high dimensions: A path integral approach," in *the IEEE International Conference on Robotics and Automation (ICRA'10)*, may 2010, pp. 2397–2403.

[12] D. Ernst, P. Geurts, and L. Wehenkel, "Tree-based batch mode reinforcement learning," *Journal of Machine Learning Research*, vol. 6, pp. 503–556, 2005.

[13] A. Yamaguchi, J. Takamatsu, and T. Ogasawara, "DCOB: Action space for reinforcement learning of high dof robots," *Autonomous Robots*, vol. 34, no. 4, pp. 327–346, 2013.

[14] R. S. Sutton, "Integrated architectures for learning, planning, and reacting based on approximating dynamic programming," in *the Seventh International Conference on Machine Learning*. Morgan Kaufmann, 1990, pp. 216–224.

[15] R. S. Sutton, C. Szepesvári, A. Geramifard, and M. Bowling, "Dyna-style planning with linear function approximation and prioritized sweeping," in *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence*, 2008, pp. 528–536.

[16] S. Levine, N. Wagener, and P. Abbeel, "Learning contact-rich manipulation skills with guided policy search," in *the IEEE International Conference on Robotics and Automation (ICRA'15)*, 2015.

[17] Y. Pan and E. Theodorou, "Probabilistic differential dynamic programming," in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, Eds. Curran Associates, Inc., 2014, pp. 1907–1915.

[18] S. Levine and V. Koltun, "Variational policy search via trajectory optimization," in *Advances in Neural Information Processing Systems 26*, C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Weinberger, Eds. Curran Associates, Inc., 2013, pp. 207–215.

[19] R. Smith, *Open dynamics engine (ODE)*, http://www.ode.org/.