Sampled Differential Dynamic Programming

Joose Rajamäki\textsuperscript{1}, Kourosh Naderi\textsuperscript{1}, Ville Kyrki\textsuperscript{2} and Perttu Hämäläinen\textsuperscript{1}

\textbf{Abstract}—We present SaDDP, a sampled version of the widely used differential dynamic programming (DDP) control algorithm. We contribute through establishing a novel connection between two major branches of robotics control research, that is, gradient-based methods such as DDP, and Monte Carlo methods such as path integral control (PI) that utilize random simulated trajectory rollouts. One of our key observations is that the Taylor-expansion, central to DDP, can be reformulated in terms of second-order statistics computed from the sampled trajectories. SaDDP makes few assumptions about the controlled system and works with black-box dynamics simulations with non-smooth contacts. Our simulation results show that the method outperforms PI and CMA-ES in both a simple linear-quadratic problem, and a multilink arm reaching task with obstacles.

I. INTRODUCTION

Trajectory optimization is an important problem in robotics. However, non-linear control problems still pose serious challenges. More efficient methods are needed, especially for solving non-linear trajectory optimization problems in (near) real-time.

This paper presents a novel trajectory optimization method we call sampled differential dynamic programming (SaDDP). SaDDP builds on two classes of previous algorithms: 1) methods based on differential dynamic programming (DDP), i.e., local optimization based on a Taylor expansion around a single nominal trajectory, and 2) Monte Carlo algorithms based on statistics from multiple random trajectory rollouts, i.e., simulations of the system state evolution up to a planning horizon, given some sampled control values for each time step. Our SaDDP method generates such rollouts similar to previous approaches. The key difference is that we use Gaussian distributions to approximate DDP’s Taylor expansion utilizing Laplace approximation. This procedure yields control sampling distributions with state-dependent closed-loop terms that are omitted in related Monte Carlo methods such as Path Integral (PI) control [1].

SaDDP is based on two main observations. The first one is that the Taylor-expansion used in DDP can be expressed in the terms of the sampled trajectories’ second order statistics. Second, we note that Covariance Matrix Adaptation Evolution Strategy (CMA-ES) can produce the statistics, with the added benefit of adapting the exploration variance. Essentially, we use CMA-ES as a robust, adaptive replacement for the finite difference computations traditionally used to compute DDP’s Taylor expansion.

This paper proceeds as follows. In Section II we briefly review relevant related work. Section III reviews the essentials of differential dynamic programming and CMA-ES to help in understanding our approach. Section IV explains our SaDDP algorithm and provides derivations for the quantities needed in it. Sections V and VI finish up with experimental results and conclusions. Our simulation results indicate that SaDDP outperforms both Path Integral control and naive CMA-ES optimization of the whole trajectory in both a linear-quadratic toy problem and a multilink arm test with obstacles.

II. RELATED WORK

Trajectory optimization has drawn much attention during the last decade. In addition to sampling based methods, progress has been made in derivative based methods, such as the iterative linear-quadratic regulator (iLQR) [2], that produce locally optimal trajectories. Furthermore, iLQR has inspired many recent algorithms [3][4] based on the linear-quadratic regulator (LQR) theory. These algorithms are computationally light because they do not compute the second order derivatives of the state dynamics, unlike the more general differential dynamic programming (DDP) [5]. Control limits in the iLQR/DDP framework had been largely an open problem until recently [6] and they still require some extra effort. In sampling based methods, control limits can be trivially added by rejection sampling.

New theory in the general duality between optimal control and estimation [7] has made it possible to use concepts of estimation theory for control problems. One remarkable recent advance in LQR-theory was made in approximate inference control (AICO) [8] that uses the duality of optimal control and estimation to optimize the trajectory time-instant at a time instead of iterating with the full trajectories. This improved the convergence of LQR-based algorithms [8].

Monte Carlo methods have also been developing rapidly in the last years. Path integral control (PI) and the methods derived from it, utilize the observation that for certain classes of control problems there is a transformation that linearizes the Hamilton-Jacobi-Bellman equation [1]. Thus, the backward computation of optimal control can be replaced by a stochastic integration of a forward-diffusion process. This approach was further extended for parametric policies by policy improvement with path integrals (PI\textsuperscript{2}) [9]. PI\textsuperscript{2} has gained a lot of popularity in the recent years. However, in this paper we refrain from the parametric policies.

Besides PI and PI\textsuperscript{2} there have been Monte Carlo algorithms that are derived directly from the principles of differential dynamic programming, e.g., probabilistic differential dynamic programming (PDDP) [10]. PDDP forms a
transition model of the system dynamics using Gaussian processes. The principles of DDP have been utilized also with general function approximators that build a model of the dynamics and perform DDP style cost minimization with this model [11][12]. Our method differs from all of these by not building a model of the dynamics but rather computing the required components of the Taylor-expansion directly from the samples’ statistics.

Compared to standard DDP, our approach does not depend on existence of analytic derivatives, or finite differences approximations of derivatives, which are unpredictable in situations with contact discontinuities. Considering Monte Carlo methods, our method is perhaps closest to PI. The differences are that we introduce an additional closed-loop term in the control sampling distributions, and our exploration variance adapts through the use of CMA-ES. While existing animation and control literature has used CMA-ES [13], [14], [15], [16], the parameters of a trajectory are typically optimized at once, whereas we employ a separate CMA-ES for each time step, which greatly reduces problem dimensionality.

III. BACKGROUND

A. Dynamic programming principle

Let us consider the (time-dependent) system dynamics

\[ x_{t+1} = f_t(x_t, u_t) \]  

where \( f \) is the state transition function, \( x \) is the system state and \( u \) is the control. Our goal is to find a series of controls i.e. a policy \( U = \{u_0, u_1, ..., u_{T-1}\} \) that produces the sequence of states \( \{x_0, x_1, ..., x_T\} \) minimizing the cost functional:

\[ J_k(x, U) = \ell_T(x_T) + \sum_{i=k}^{T-1} \ell_i(x_i, u_i) \]  

Here \( J_k(x_k, U) \) is the cost-to-go at time \( k \) given the state \( x_k \) and policy \( U \). \( \ell_i(x_i, u_i) \) is the instantaneous cost, i.e. the cost of taking action \( u_i \) in state \( x_i \) and \( \ell_T(x_T) \) is the cost of the final state. The minimum cost is customarily referred to as ‘value’, \( V_t(x_i) \equiv \min_U J_t(x_i, U) \).

All dynamic programming is based on the Bellman optimality principle also known as dynamic programming principle [5]:

“An optimal policy has the property that whatever the initial state and initial decisions are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.”

We can apply the Bellman principle to the functional of Equation (2). Let us denote a partial policy as \( U_t = \{u_t, u_{t+1}, ..., u_{T-1}\} \). The cost functional of Equation (2) can be decomposed as:

\[ J_t(x_t, U_t) = \ell(x_t, u_t) + J_{t+1}(x_{t+1}, U_{t+1}) \]

\[ = \ell(x_t, u_t) + J_{t+1}(f_t(x_t, u_t, t), U_{t+1}) \]  

According to the Bellman principle, the following applies to the minimum cost, i.e. the value:

\[ V_t(x) = \min_U [\ell(x_t, u_t) + J_{t+1}(f_t(x_t, u_t, t), U_{t+1})] \]  

\[ = \min_{u_t} [\ell(x_t, u_t) + V_{t+1}(f_t(x_t, u_t, t))] \]  

B. Differential dynamic programming

Equation (5) has the consequence that optimization can be performed time-step by time-step. It thus justifies the following iteration that is used in differential dynamic programming (DDP) and the algorithms derived from it:

1) Given a policy, compute the future states.
2) Compute the optimal control for the last time step, then compute the optimal control to the second last time step and so on.
3) Replace the policy with the computed values and repeat until convergence.

How step 2 of the previous iteration is performed may vary in different algorithms. The general idea in all DDP based algorithms is to form the second order Taylor-expansion \( Q_t(\Delta x, \Delta u) \) of the cost \( J_t(x_t, U_t) \) and minimize it:

\[ Q_t(\Delta x, \Delta u) = J_t(x_t + \Delta x, \{u_t + \Delta u, U_{t+1}\}) - J_t(x_t, \{u_t, U_{t+1}\}) \]

\[ \approx \frac{1}{2} \begin{bmatrix} \Delta x \\ \Delta u \end{bmatrix}^T \begin{bmatrix} Q_x & Q_{xu} \\ Q_{ux} & Q_{uu} \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta u \end{bmatrix} \]  

This is minimized by:

\[ \Delta u = \arg \min_{\Delta u} Q(\Delta x, \Delta u) \]

\[ = -Q_{uu}^{-1}(Q_u + Q_{ux}\Delta x) \]  

This expression has the open-loop term \( k = -Q_{uu}^{-1}Q_u \) and the state-dependent feedback term \( K\Delta x \), where \( K = -Q_{uu}^{-1}Q_{ux} \) is the feedback gain. DDP is based on the observation that \( Q_t(\Delta x, \Delta u) \) can be formed with \( Q_{t+1}(\Delta x, \Delta u) \) using an approximation of the dynamics [17]. This process is straightforward but involves much computation. As it is unnecessary for the understanding of our algorithm we omit the details here.

C. Covariance matrix adaptation evolutionary strategy

The covariance matrix adaptation evolutionary strategy (CMA-ES) is a numerical derivative-free second order optimization method. As an evolutionary algorithm it finds the minimum of an objective function by the interplay of sampling candidates for the minimum point and selecting the best candidates for parameter updates. It is a well established algorithm, and introductory material to the workings of the algorithm exists [18]. The main steps of one CMA-ES
within one iteration of the algorithm. Hence the name covariance matrix adaptation. The covariance of the search distribution. The directions where to look \( V \).

We used these default values except for the forgetting factor \( \delta \) can be used to adapt the step-size \( \sigma \) to the standard normal distribution and its deviation from this conjugate evolution path \( \mu \).

All smaller case letters represent the data, and it varies in the interval \([1, \ldots, K]\). The tuning of all the parameters and good factors and the damping should be adjusted according to the problem at hand. The effective number of samples consists of the newly computed value and how much of the old value is preserved. The effective number of samples' second order statistics using these probabilities. The costs, transforming them to probabilities and computing the samples’ second order statistics using these probabilities. The second order statistics can be accumulated across multiple iterations by CMA-ES. We use these as approximations of the Taylor-expansions in consecutive iterations to sample new control values around the optima of these models. Repeating the previously described two stages can be done until convergence, which is analogous to performing differential dynamic programming. In addition to being computationally cheap our algorithm has the following desirable properties of Monte Carlo algorithms:

- tolerance to non-convexity,
- tolerance to non-existence of first and second order derivatives (i.e., discontinuities),
- ability to work with black-box dynamics with no access to Jacobians or Hessians,
- control-limits can be added trivially to the algorithm.

### A. Notations

We use the curly brackets to signify sets, for example \( \{x_0, x_1, x_3\} \) is a set containing elements \( x_0 \), \( x_1 \) and \( x_3 \). We also use the Matlab-style notation for running indices so that \( t = 0 : 10 \) denotes the variable \( t \) running from zero to ten. The variables \( t \) and \( T \) are used to refer exclusively to time. For example \( x_{t,n} \) is the \( n \)th measurement of \( x \) at time \( t \). We also use combinations of the previous notations. For example \( \{x_{t,n}\}_{t=0:T} \) means a set of \( n \)th measurements of quantity \( x \) at time instants from zero to \( T \) and \( x_{t,1:N} \) is a vector \( x \) at time \( t \) that has the index running from one to \( N \). We start indexing from 1 except for time for which 0 denotes the current time.

### B. Estimating the future of the system

Algorithm 2 presents how the future evolution of the system is evaluated using standard particle filtering techniques [19]. We sample control values, simulate the corresponding future states, and evaluate the associated costs. These costs can be used to compare the likelihood of each trajectory to the others. This is illustrated in Figure 1a with a sample problem. If the probability mass collapses to only a few samples, the computational resources can be reallocated to the more likely trajectories through the standard particle filter process of resampling. This takes place in lines 4-5.

### IV. Sampled Differential Dynamic Programming (SADDP)

Our SaDPP method uses the transition model of Equation (1). We assume only that \( f \) can be evaluated point-wise. Additionally, our algorithm requires us to be able to evaluate the instantaneous cost function \( \ell_t(x_t, u_t) \) point-wise.

The parameters of the algorithm need some clarification. All smaller case letters \( c \) denote forgetting factors. Their sub-index indicates the parameter they account for, and their role is to determine how much every quantity consists of the newly computed value and how much of the old value is preserved. The effective number of samples \( \mu_{\text{eff}} = \left( \sum_{k=1}^K u_k^2 \right)^{-1} \) indicates how many samples actually represent the data, and it varies in the interval \([1, K]\). If only one weight is one and the rest are zero \( \mu_{\text{eff}} = 1 \) and when the weights are equally distributed \( \mu_{\text{eff}} = K \). The conjugate evolution path \( p_s \) should be distributed according to the standard normal distribution and its deviation from this can be used to adapt the step-size \( \sigma \). To prevent the step size from growing too rapidly, damping \( d_p \) is used. The forgetting factors and the damping should be adjusted according to the problem at hand. The tuning of all the parameters and good default values are dealt with in great detail by Hansen [18]. We used these default values except for the forgetting factor \( c_p \) and the step size \( \sigma \), whose values are discussed in Section V.

CMA-ES updates the mean and the covariance matrix of the search distribution. The directions where to look for future candidates are stored in the covariance matrix, hence the name covariance matrix adaptation. The covariance matrix is updated with two terms as seen on line 13. The term \( pp^* \) is the so-called rank-1 update which conveys the information of variation between multiple iterations. The rank-\( \mu \) update \( C_\mu \) stores the information of the variation within one iteration of the algorithm.

---

**Algorithm 1** Covariance matrix adaptation evolutionary strategy

1. Sample candidate solutions \( \{x\}_1:N \) from \( \mathcal{N}(m, \sigma^2C) \).
2. Compute the goodness/score \( g \) of the sampled candidates \( x \).
3. Sort \( \{x\}_1:N \) to descending order according to the scores \( g \).
4. Select \( K \) first candidates \( \{x\}_1:K \) and discard the rest.
5. Compute weights \( \{w\}_1:K \) for the solutions.
6. Normalize the weights \( \{w\}_1:K \) to sum up to one.
7. Store the old mean \( m_{\text{old}} = m \).
8. Compute the new mean \( m = \sum_{k=1}^K w_kx_k \).
9. Compute the evolution path \( p \)
   
   \[ p = (1 - c_p)p + \sqrt{c_p(2 - c_p)}\mu_{\text{eff}} m - m_{\text{old}} \]

10. Compute the conjugate evolution path \( p_s \)
    
    \[ p_s = (1 - c_p)p_s + \sqrt{c_p(2 - c_p)}\mu_{\text{eff}} C - \frac{\mu_{\text{eff}}}{\mu_{\text{eff}} - c_p} m - m_{\text{old}} \]
11. Compute centered samples \( y_k = x_k - m_{\text{old}} \).
12. Compute the rank-\( \mu \) weighted covariance matrix \( C_\mu \)
    
    \[ C_\mu = \sum_{k=0}^K w_k y_k y_k^T \]
13. Update the covariance matrix \( C \)
    
    \[ C = (1 - c_1 - c_\mu)C + c_1pp^* + c_\mu C_\mu \]
14. Update the step size \( \sigma \)
    
    \[ \sigma = \sigma \exp \left( \frac{T}{t_d} \left( \frac{\|p_s\|}{\|Ax(0,N)\|} - 1 \right) \right) \]

---

The problem of resampling. This takes place in lines 4-5. The
Algorithm 2 System evolution estimation

Input:
- initial state $x_0$
- control distributions $U_{t,n}$
- resampling threshold $N_{res}$

Output:
- control estimates $\{\{u_{t,n}\}_{t=0:T-1}\}_{n=1:N}$
- system cost estimates $\{\{\ell(x_{t,n},u_{t,n},t)\}_{t=0:T-1}\}_{n=1:N}$

1: Set all $\{x_{0,n}\}_{1:N}$ to $x_0$
2: for Time $t = 0 : T - 1$ do
   //Resample when only a few samples represent the data.
   3: Compute effective number of samples $\mu_{eff} = \frac{1}{\sum_{n=1}^{N_{res}} \omega_{t,n}^2}$
   4: if $\mu_{eff} < \frac{1}{N_{res}}$ then
      5: Sample states to be continued $\{x_t\}_n$ using $w_{t,1:N}$ as distribution
      6: for Sample $n = 1 : N$ do
         7: Sample $u_{t,n}$ from $U_{t,n}$
         8: Evaluate $x_{t+1,n} = f(x_{t,n}, u_{t,n}, t)$
         9: Evaluate $\ell_{t,n} = \ell(x_{t,n}, u_{t,n}, t)$
      //Scaled softmax function transforms the cost estimates to pseudo-probabilities. See Section IV-D for its details and for the details of choosing its scale parameter $s$.
      10: Compute scale $s$ using Equation (23)
      11: $\bar{w}_{t,1:N} = \text{scaled}_\text{softmax}(\bar{\ell}_{t,1:N}, s)$
      12: $w_{t,n} = \bar{w}_{t,1:N} \omega_{t-1,n}$
      13: Normalize $w_{t,1:N}$ to sum up to one
   return control estimates $\{\{u_{t,n}\}_{t=0:T-1}\}_{n=1:N}$, state estimates $\{\{x_{t,n}\}_{t=0:T-1}\}_{n=1:N}$ and cost estimates $\{\{\ell(x_{t,n},u_{t,n},t)\}_{t=0:T-1}\}_{n=1:N}$

Algorithm 3 Building the Taylor-expansions

Input:
- set of sampled control values $\{\{u\}_{0:T-1}\}_{1:N}$
- set of estimated states $\{\{x\}_{0:T-1}\}_{1:N}$
- corresponding set of cost estimates $\{\{\ell\}_{0:T-1}\}_{1:N}$

Output:
- state-control estimates $\{\{x_{t,k}^T, u_{t,k}^T\}_{t=0:T-1} \}$
- control-state covariance estimates $\{C_{t}\}_{0:T-1}$

1: for Time $t = T - 1 : 0$ do
   //Evaluate the upper-bound for the cost of each state depending on whether the trajectory was pruned or not.
   2: for each sample $n$ do
      3: if $t == T - 1$ then
         4: $V_{t,n} = \bar{V}_{t,n}$
      5: else if sample $n$ has successors then
         6: $V_{t,n} = \bar{V}_{t,n} + \min_{k \in \text{successor}(n)} \left(\bar{V}_{t+1,k}\right)$
      7: else
         8: $V_{t,n} = \infty$
   9: for Time $t = 0 : (T - 1)$ do
   10: Concatenate the vectors $z_{t,n} = [x_{t,n}^T, u_{t,n}^T]^T$
   11: Sort pairs $\{z_{t,n}, V_{t,n}\}_{n=1:N}$ to ascending order according to $V_{t,1:N}$
   12: Select $K$ first pairs $\{z_{t,k}, V_{t,k}\}_{1:K}$ and discard the rest.
   13: Discard $\{z_t, V_t\}$ with $V_t = \infty$
   14: Compute scale $s$ using Equation (23)
   15: $w_{t,1:K} = \text{scaled}_\text{softmax}(\bar{V}_{t,1:K}, s)$
   //See Algorithm 1.
   16: Run one iteration of CMA-ES with data $\{z_{t,k}\}_{1:K}$ and weights $w_{t,1:K}$.
   return mean parameters of CMA-ES $\{x_{t,k}^T, u_{t,k}^T\}_{0:T-1}^T$ and covariance matrices of CMA-ES $\{C_{t}\}_{0:T-1}$

C. Forming the Taylor-expansion

Once we have estimated the future evolution of the system, we estimate required parts of the Taylor-expansion (6). The first step in this process is computing the upper-bound estimates $\bar{V}_t(x_t)$ for $\bar{V}_t(x_t)$. We get $\bar{V}_t(x_t)$ by summing the instantaneous costs $\{\ell\}_{0:T-1,0:N}$, which were computed in the system evolution estimation in Algorithm 2:

$$V_t(x_t) \leq \bar{V}_{t,n}(x_{t,n}) \forall n$$ \hspace{1cm} (9)

$$= \sum_{t'=t}^{T-1} \ell(x_{t',n}, u_{t',n})$$ \hspace{1cm} (10)

These are higher bounds because the minimum cost of any realized trajectory necessarily sets the upper bound for the minimum cost. Figure 1b shows these cost estimates in a sample problem. Our aim is to bring the upper bound estimates $\bar{V}_{t,n}(x_{t,n})$ closer to the actual value $V_t(x_t)$. This is accomplished by selecting the control distributions $U$ of Algorithm 2 such that they produce lower $\bar{V}_{t,n}(x_{t,n})$. To do this we first transform $\bar{V}_{t,n}(x_{t,n})$ to probabilities and, using them, compute a Gaussian model of the states and controls for each time step. These models are used in the subsequent iterations to sample better control values. The Gaussian models can be gathered across many iteration utilizing CMA-ES. Computing the upper-bound cost estimates and forming the Gaussian models is presented in Algorithm 3.

Our Gaussian models represent the Laplace approximations of the distributions that we get by mapping the Taylor-expansions (7) with the scaled_softmax-function. Thus, the control $u_t + \Delta u$ given by Equation (8), which is the zero gradient point of the Taylor-expansion, coincides with the mean control. This means that $u_t + k_t$ are the mean control values returned by Algorithm 3. Similarly, the nominal states $x_t$ are the mean states returned by Algorithm 3.

Good new control values are likely to be found in the directions indicated by $Q_{uu}^{-1}$. The feedback gain $K$ tells which controls are most probable given the difference to the nominal state. Next we see how $Q_{uu}^{-1}$ and $K$ are related to the
covariance matrices returned by Algorithm 3. Let us denote:

\[
Q_H = \begin{bmatrix} Q_{xx} & Q_{xu} \\ Q_{ux} & Q_{uu} \end{bmatrix}
\]

(11)

and:

\[
C_t = \begin{bmatrix} C_{xx} & C_{xu} \\ C_{ux} & C_{uu} \end{bmatrix}
\]

(12)

In general, the covariance matrix of a Gaussian random variable equals the inverse Hessian of the random variable’s negative logarithmic probability density function. This is the central observation for our algorithm because it establishes the following relation of \(Q_H\) and the covariance matrices \(C_t\) returned by Algorithm 3:

\[
Q_H^{-1} = c C_t
\]

(13)

where \(c\) is a constant. To get the different components of \(Q_H\), we may apply the block-wise matrix inversion:

\[
\begin{bmatrix} A & B \\ C & D \end{bmatrix}^{-1} = \begin{bmatrix} A^{-1} + A^{-1}B(D-CA^{-1}B)^{-1}CA^{-1} - A^{-1}B(D-CA^{-1}B)^{-1} \\ -(D-CA^{-1}B)^{-1}CA^{-1} & (D-CA^{-1}B)^{-1} \end{bmatrix}
\]

(14)

Thus:

\[
Q_{uu} = \frac{1}{c} (C_{uu} - C_{ux} C_{xx}^{-1} C_{xu})^{-1}
\]

(15)

and

\[
Q_{ux} = -\frac{1}{c} (C_{uu} - C_{ux} C_{xx}^{-1} C_{xu})^{-1} C_{ux} C_{xx}^{-1}
\]

(16)

We can express the feedback gain \(K\) in terms of the covariance matrices:

\[
K = -Q_{uu}^{-1} Q_{ux}
\]

(17)

\[
= C_{ux} C_{xx}^{-1}
\]

(18)

A notable thing in this is that the feedback gain term does not contain the constant \(c\), as it cancels out. To sum up, we can perform a stochastic sampled version of differential dynamic programming by substituting the control distributions \(\mathcal{U}\) of Algorithm 2 with distributions \(\mathcal{U}_{\text{opt}}\):

\[
\mathcal{U}_{\text{opt}} = \mathcal{N}(\mathbf{u} + K \Delta \mathbf{x}, C_{\text{opt}})
\]

(19)

We have omitted the time indices here for clarity. The matrices \(K\) and \(C_{\text{opt}}\) are given by:

\[
K = \lambda_{\text{fb}} C_{ux} (C_{xx} + \lambda_{\text{state}} I)^{-1}
\]

(20)

\[
C_{\text{opt}} = C_{uu} - C_{ux} (C_{xx} + \lambda_{\text{state}} I)^{-1} C_{xu} + \lambda_{\text{expl}} I
\]

(21)

Incidentally, the control distribution \(\mathcal{U}_{\text{opt}}\) that we get by this choice, is the joint Gaussian distribution of the state and control conditioned at the current state when we set \(\lambda_{\text{fb}} = 1\), \(\lambda_{\text{state}} = 0\) and \(\lambda_{\text{expl}} = 0\). \(u\) designates the mean control returned by Algorithm 3 and \(\Delta \mathbf{x}\) is the difference of the current state and the mean state, also returned by Algorithm 3. The regularization factors \(\lambda_{\text{fb}}, \lambda_{\text{state}}\) and \(\lambda_{\text{expl}}\) do not belong to the algorithm per se. Their role is to maintain the algorithm functional in the situations where it would otherwise fail. First of all, they work as regularization and ensure that all of the relevant matrices are invertible. The regularization factor \(\lambda_{\text{fb}}\) is given by

\[
\lambda_{\text{fb}} = \min \left(1, \frac{1}{\Delta x^T (C_{xx} + \lambda_{\text{state}} I)^{-1} \Delta x} \right)
\]

(22)

\(\lambda_{\text{fb}}\) ensures that the feedback term is not used far from the data that generated the distribution. It thus prevents divergent control. The regularization factor \(\lambda_{\text{state}}\) can be thought of as the smallest precision with which we can know the state. The regularization factor \(\lambda_{\text{expl}}\) on the other hand can be thought of as the smallest allowed amount of exploration per dimension. Thus, in addition to making \(C_{\text{opt}}\) invertible, \(\lambda_{\text{expl}}\) prevents it from becoming a zero matrix. This will otherwise happen easily for example in situations like the one in Figure 1b where the variation of early time steps is small.

**D. The scaled softmax heuristic**

We made use of the scaled softmax function in Algorithms 2 and 3 to transform costs to probabilities that sum up to one. This is of course just one way to map quantities to probabilities and other could be used as well. The scaled softmax function is introduced in Algorithm 4.

**Algorithm 4 The scaled softmax function.**

1: **Input**: vector \(x\), scale \(s\)
2: **Output**: probability vector \(p\)
3: \(x_{\text{max}} = \max (x)\)
4: \(x_{\text{min}} = \min (x)\)
5: for \(i = 0: \text{dim} (x)\) do
6: \(x_i = \frac{x_i - x_{\text{min}}}{x_{\text{max}} - x_{\text{min}}} \cdot s\)
7: \(p_i = \sum x_i e^{x_i}\)
8: *return* \(p\)

At first, the scaled softmax function maps the entries in vector \(x\) to the interval \([s, 0]\), assuming a negative user-specified scale \(s\). These are transformed to probabilities by the exponential function. The scale \(s\) allows for adjusting the aggressiveness of penalizing worse values.

Because mean is sensitive to outliers we do not want to assign too high a weight to the bad samples. This implies choosing a big value for \(s\). On the other hand \(s\) being too big will lead to the weights being zero for all but the first sample which is also non-desirable. Let us assume that we need the scale \(s\) for values \(X = \{x_n\}\). We used the following heuristic for \(s\):

\[
s = -\max (\text{kurtosis}(X), 3).
\]

(23)

We chose this, because kurtosis is small for evenly distributed values and grows when the distribution becomes peaked.
Fig. 1: System evolution and resampling in a case where the controlled object moves at a constant velocity from left (current time) to right (planning horizon), and vertical velocity is controlled. Instantaneous cost equals vertical squared deviation from center. Going through a red barrier causes a high additional instantaneous cost. Figure 1a shows trajectories evaluating the system evolution. The blueness of the trajectory segment corresponds to the weight $w_{t,n}$ of Algorithm 2 and is thus an estimate of the probability of the trajectory given the past. Figure 1b shows the value upper-bound cost estimates $\tilde{V}_{t,n}$ of the trajectory segments, dark color corresponding to high value. Thus, the green color corresponds to the probability of the trajectory given the future. The power of the resampling can be seen in how most of the trajectories going through a wall are pruned and the more successful ones get more “offspring”.

V. RESULTS

To explore the convergence properties of SaDDP, we used two test problems. The results of these test are presented in Figure 3. The first test problem is a two-dimensional 8-link robot arm reaching problem with random goal and four random location obstacles and 60 time-steps as shown in Figure 2. In this problem we simulate the state transition using the Box2D physics engine and we only get to know the end-state after applying a control in a particular state. The state in this problem is 48-dimensional, containing the position, velocity, angle and angular velocity of each of the eight links. The control is 8-dimensional consisting of target velocities for the link. Box2D adjusts the torques internally to match the velocities. The instantaneous cost in this problem is:

$$\ell_t(x_t, u_t) = \|g - x_t^{\text{end}}\|_2 + \frac{1}{1000} u_t^T u_t, \quad (24)$$

where $g$ is the goal point, $x_t^{\text{end}}$ is the position of the end-effector and $u_t \in \mathbb{R}^4$ is the vector containing the target velocities of the simulated joint motors.

The second test problem is a linear-quadratic one, whose optimum solution was computed with standard DDP and for which the standard DDP is exact:

$$x_{t+1} = (I + \delta t A)x_t + \delta t B u_t \quad (25)$$

$$\ell(x_t, u_t) = (x_t - x_{\text{goal}})^T Q (x_t - x_{\text{goal}}) + u_t^T R u_t \quad (26)$$

The state was chosen to have five dimensions $x \in \mathbb{R}^5$ and the control was chosen to be three dimensional $u \in \mathbb{R}^3$. The time step $\delta t$ was selected to be 0.25 and we used 15 time steps. The matrices $A \in \mathbb{R}^{5 \times 5}$ and $B \in \mathbb{R}^{5 \times 3}$ contain randomly selected numbers from uniform distribution $U(-1, 1)$. The state cost matrix $Q \in \mathbb{R}^{5 \times 5}$ and the control cost matrix $R \in \mathbb{R}^{3 \times 3}$ should be positive-semidefinite for the optimal control to be non-divergent. They were thus formed as a product of the correct size matrix $U$ containing uniformly distributed random numbers from distribution $U(-1, 1)$ in the following manner:

$$Q = U U^T \quad (27)$$
$$R = U U^T \quad (28)$$

The initial state $x_0$ and the goal state $x_{\text{goal}}$ were randomly selected from the uniform distribution $U(-10, 10)$.

The test problems were produced randomly but the random number generators were seeded such that the problems were the same for each of the tested methods and parameter settings. This is the case both for the linear-quadratic problem and the reaching task.

A. SaDDP and sampling based methods

We compared SaDDP to Path Integral control, which is the closest previous Monte Carlo algorithm in that it also operates on raw control values instead of parametric policies. As we use CMA-ES in a novel way, we also compared our
CMA-ES optimization of the controls of the full trajectory. We used 32 sampled trajectories per iteration in all our tests. The update batch size parameter $K$ of Algorithm 3 was set to $N/2$. In the linear quadratic test case PI’s exploration noise was set to $\lambda R^{-1}$ as required by the algorithm. In the robot arm reaching task we used $I$ as PI’s exploration variance and swept over the $\lambda$ parameter values, since PI failed to converge using the theoretically valid $\lambda R^{-1}$. We also set the initial control sampling distributions $U_{t,n}$ of Algorithm 2 to standard normal distribution. In subsequent iterations of SaDDP, we used three special distributions. $U_{t,1}$ was Dirac delta that gave us the control of the best trajectory of the previous iteration. $U_{t,2}$ was $U_{\text{opt}}$ with $C_{\text{opt}} = 0$, i.e. it gave us the nominal trajectory. $U_{t,3}$ was $U_{\text{opt}}$ with $C_{\text{opt}} = 0$ and $\lambda_{\text{thr}} = 0$, i.e. it gave us the nominal trajectory disregarding the state dependency. We used $\lambda_{\text{expl}} = 0.0625$ for all SaDDP tests.

Both SaDDP and PI perform better than the full trajectory CMA-ES in the more demanding robot arm test. SaDDP seems to perform best. PI converges to the level dictated by sampling variance. Using PI with $\lambda = 0.01$ means roughly sampling random trajectories and selecting the best. As the variance has a larger effect when fewer samples affect the policy, the levelling happens earlier with smaller $\lambda$. This doesn’t happen to SaDDP since both the sampling covariance matrix and the parameter $s$ of scaled softmax are adaptive. SaDDP converging faster than the full trajectory CMA-ES is in line with standard DDP being similar or slightly better than Newton’s method for the whole trajectory [6].

B. Relationship with standard DDP

The reaching problem of Figure 2 demonstrates our algorithm running with black-box dynamics. This problem demonstrates a set up, in which standard DDP cannot be applied. There are many reasons that render standard DDP inapplicable here. We use hard contacts and the obstacles make the dynamics discontinuous, which is why computation of the gradients and Hessians cannot be done. The second reason is that we only simulate the system forward from the state that it is in. This means that we cannot apply perturbations to the state and simulate the system forward from that state. This would also be necessary to use standard DDP.

The linear-quadratic test problem is solved by one iteration of DDP. As Figure 3a shows, all the sampling based methods take dozens of iterations to converge. One reason for this is that the sampling based methods’ control values lie inside the convex hull of the sampled values whereas standard DDP can take larger steps. On the other hand this prevents wildly divergent trajectories. It’s noteworthy that it would be possible to use a hybrid of SaDDP and standard DDP where the SaDDP is used for example for the time-steps for which collisions occur and standard DDP for the rest. In that sense SaDDP and DDP can be viewed more complementary rather than exclusive approaches.

C. Remarks

The effect of resampling is highly task-dependent. In the LQ-problem it is actually harmful, but helps with initial convergence in the multilink arm case, where resampling allows early pruning of trajectories that hit the obstacles, similar to Figure 1. Using the feedback gain regularization of Equation (22) is beneficial in both tasks.

Quite early on in our tests we noticed that using the step size control of CMA-ES makes SaDDP rather unstable. All of our tests are thus run without using the step size control, i.e. we skipped lines 10 and 14 of Algorithm 1. We also restricted $c_n$ parameter of CMA-ES to being at least 0.75.

There is no mechanism that prevents $C_{\text{opt}}$ from approaching a zero matrix prematurely. This seemed to be problematic particularly for the first time steps of the trajectory where $C_{\text{opt}}$ tends to become $\lambda_{\text{expl}} I$ quickly. This takes SaDDP back to the direction of PI. One could think of other regularization schemes that scale $C_{\text{opt}}$ appropriately and add regularizing noise only to the zero variance directions. These different regularization schemes are outside the scope of this paper and they should be investigated in the future.

VI. CONCLUSIONS

We have introduced Sampled Differential Dynamic Programming (SaDDP), a novel Monte Carlo version of the widely used differential dynamic programming (DDP) control optimization method. In contrast to DDP, SaDDP can be used with hard contacts and other cases with discontinuous dynamics. In our tests, SaDDP outperforms both Path Integral control (PI) and Covariance Matrix Adaptation Evolution Strategy (CMA-ES) optimization of the whole trajectory. Instead of optimizing all trajectory parameters at once, we utilize CMA-ES for each control time step, with the estimated means and covariance matrices utilized to approximate DDP’s Taylor-expansion around the nominal trajectory. The nominal trajectory is in our case estimated as the CMA-ES control and state means for each time step. Compared to Path Integral control, our approach allows for 1) adaptive exploration covariance matrix, and 2) an additional state-dependent closed-loop term to guide the sampling.

A limitation of our work is that our test cases only comprise offline trajectory optimization, i.e., time is not advancing between the iterations. However, in our multilink arm test, one iteration with 300 trajectories and 60 rollout time steps only consumes 330 ms on a single CPU thread. Thus, online operation in the style of Tassa et al. [6] or Hämäläinen et al. [20] should be possible with multithreading, although it may require an extension for expanding the sampling variance when the environment changes and the presently pursued movement strategy is no longer feasible. In future work, we aim to combine SaDDP with other methods and also compare SaDDP against a wider variety of previous methods. Additionally, we aim to expand our test cases to more complex 3D simulations.
(a) Convergence of the linear-quadratic test problem. The cost is the relative cost above the theoretical minimum computed by DDP.

(b) Convergence of the 8-link robot arm reaching. The cost is the relative cost above the minimum found with 300 iterations.

Fig. 3: This figure has the convergence tests. The curves are the averages 100 test cases. Regularization off means having $\lambda_{fb}$ to be always 1. Resampling 25% means resampling when $\mu_{eff}$ is less than 25% of the sample amount. Resampling 0 % means no resampling. The regularization $\lambda_{state}$ was set to $0.01 \cdot \max(\text{diag}(C_{xx}))$. $\lambda_{expl}$ is set to 0.0625.

ACKNOWLEDGMENT

We would like to thank our reviewers for their good comments.

This work has been supported by TEKES (The Finnish Funding Agency For Innovation). Joose Rajamäki additionally thanks KAUTE foundation for the support (Grant 201500537).

REFERENCES


