A Generic Solver for Unconstrained Control Problems with Integral Functional Objectives

Shih-Hao Tseng

Abstract— **We present a generic solver for unconstrained control problems (UCPs) whose objectives take the form of an integral functional of the controllers. The solver generalizes and improves upon the algorithm in [1] for the Witsenhausen's counterexample, which provides the best-known results. In essence, we show that minimizing the objective implies minimizing the marginal cost functions almost everywhere, and we perform the latter task pointwisely by the adaptive minimization technique, which speeds up the computation. We implement single-threaded and parallelized versions of the proposed algorithm. Our implementation runs $30\times$ faster than the algorithm in [1] on the Witsenhausen's counterexample, and we demonstrate the applicability of the solver and discuss the possible generalization to constrained problems and multi-dimensional controllers through three more examples.**

I. Introduction

In this work, we focus on the unconstrained control problem (UCP) with the objective

$$\min \ \mathcal{J}[U]$$

where $U = \{u_0(y_0), u_1(y_1), \ldots, u_{M-1}(y_{M-1})\}$ is the set of controllers $u_m : \mathbb{R} \to \mathbb{R}$. We assume that the objective functional $\mathcal{J}[U]$ can be expressed as

$$\mathcal{J}[U] = \int L_m(u_m(y_m), y_m)\ dy_m + \mathcal{R}_m[U_{-m}] \quad (1)$$

for all $m = 0, \cdots, M-1$ (in other words, the functional can be expanded with respect to each m), where $L_m : \mathbb{R}\times\mathbb{R} \to \mathbb{R}$ and $\mathcal{R}_m[U_{-m}]$ is the *residual functional* depending only on the controllers other than u_m. UCP is "unconstrained" in the sense that we require the ability to embed all constraints, e.g., system dynamics, in (1). Although omitted, we remark that L_m might also depend on U_{-m} and the decomposition of L_m and $\mathcal{R}_m$ is not necessarily unique.

Finding the optimal controller of a control problem is a daunting task, even when the problem imposes no constraints. The traditional approach to a control problem is to analyze its structure and conclude some useful properties that would help find the optimal controller. However, as demonstrated by the famous Witsenhausen's counterexample [2], preferred properties, such as linearity, does not hold in general. As a result, deriving the optimal controller becomes craftsmanship relying on keen observations.

Fortunately, we often need a near-optimal controller rather than an exact optimal one in practice. There are two approaches, the analytical and the numerical, to design a near-optimal controller. The former approach examines some specific controller structure based on the problem characteristics. Again, the performance of the design highly depends on the sophisticated understanding of the problem. On the other hand, the numerical approach develops techniques to approximate the problem and obtain good approximations of the optimal controllers. Correspondingly, the main challenges lie on computation efficiency and approximation quality.

It used to be the computation-demanding nature of the numerical methods which impedes the adoption. But luckily, the advancing technologies in the past decades have hugely reshaped the research landscape: Cheaper computation resources and parallelization techniques facilitate the development of image classification [3], machine learning [4], [5], and genomics processing [6]. Increased computation power not only grants us higher efficiency but also enables finer sampling granularity and potentially better approximation quality. Therefore, we argue that numerical methods have great potential in the upcoming computation-rich era.

Although numerical methods could potentially compute faster with plenty of computation resources, its effectiveness plays a central role in getting closer to the optimum. Blindly adopting numerical methods can be ineffective. Accordingly, it is critical to ask how to design numerical methods that are effective for general problems, in particular, for general UCPs.

Our approach to tackling general UCPs evolves from the algorithm proposed for the Witsenhausen's counterexample in 2017 [1]. The algorithm in [1] outperforms all previous attempts on the well-known Witsenhausen's counterexample (e.g., [7]–[9]), and its mechanism does not depend on the property of the given objective functional. Although the algorithm finds the controllers that result in the record-low cost, it is computationally demanding and requires a special math tool called calculus of variation.

A. Contribution and Organization

We examine UCP and provide a generic algorithm to find a near-optimal controller numerically. The proposed algorithm generalizes the algorithm in [1] with the following improvements: First, it presents a new angle viewing the local Nash minimizing phase in the algorithm in [1] which does not involve calculus of variation. In summary, our analysis reveals that UCP leads to a per-point marginal cost optimization problem, and the two methods used in the algorithm in [1], local Nash minimizing and local denoising, approach the problem using different candidate sets. We also apply the adaptive minimization technique to speed up the convergence significantly. Furthermore, our proposed

Shih-Hao Tseng is with the Division of Engineering and Applied Science, California Institute of Technology, Pasadena, CA 91125, USA. Email: shtseng@caltech.edu

algorithm adopts a unified termination criterion applicable to arbitrary UCPs.

We then implement a generic solver based on the proposed algorithm in C++ and demonstrated that it converges $3\times$ faster than the algorithm in [1] on Witsenhausen's counterexample. Since the proposed algorithm is parallelizable, we enhance the single-threaded C++ implementation using NVIDIA CUDA and observe another $10\times$ computation speed up. We also demonstrate how the solver works on the zero-delay source-channel coding problem, inventory control problem, and 2-dimensional Witsenhausen's counterexample. And we open source the tool for future research.

The paper is organized as follows. We first provide a brief overview of the algorithm in [1] in Section II. Section III introduces the ideas of marginal cost functions, local update, partial exhaustion, and adaptive minimization. Those ideas contribute to the design of our solver. We then implement the solver and examine its performance improvement in Section IV. In Section V, we demonstrate that how we can use the solver to approach different problems. Finally, we conclude the paper in Section VI.

B. Notation

By convention, we denote the state by x, control by u, observation by y, and disturbance by w. Let $\mathbb{E}_A$ be the expected value with respect to random variable A. We omit A when the expectation is taken with respect to all random variables. We denote by $A \sim \mathcal{N}(\mu, \sigma^2)$ a Gaussian random variable A with mean μ and variance σ^2 and by $A \sim \mathcal{U}(a, b)$ a uniform random variable distributed over $[a, b]$. Given a number M, we slightly abuse the notation to denote $m = 0, \dots, M-1$ by $m \in M$. For a multivariate function $F(a, b)$, we introduce the shorthand notation $F'(a, b) = \frac{\partial F(a,b)}{\partial a}$ to denote the partial derivative with respect to the first variable.

Given a functional $\mathcal{J}[U]$, we denote by $\frac{\delta \mathcal{J}[U]}{\delta u}(y)$ the functional derivative of $\mathcal{J}[U]$ with respect to the function $u(y)$, which is derived from the Taylor expansion below:

$$\begin{aligned}\mathcal{J}[U + \epsilon\delta u] = \mathcal{J}[U] \\ + \epsilon \int \frac{\delta \mathcal{J}[U]}{\delta u}(y)\delta u(y)dy \\ + \frac{\epsilon^2}{2} \int \frac{\partial}{\partial u}\frac{\delta \mathcal{J}[U]}{\delta u}(y)\delta u^2(y)dy + O(\epsilon^3)\end{aligned}$$

where $U + \epsilon\delta u$ refers to the set $(U \backslash \{u(y)\}) \cup \{u(y) + \epsilon\delta u(y)\}$, $\delta u(y)$ is a bounded function for variation, and $O(\epsilon^3)$ represents the residual terms of order ϵ^3 or higher.

II. Existing Algorithm and its Limitation

We first explain the algorithm in [1], which attains the state-of-the-art best results for Witsenhausen's counterexample, in Section II-A. We then discuss the limitations of the algorithm in Section II-B.

A. Basic Structure

In [1], Witsenhausen's counterexample is deemed an optimization problem minimizing a given functional. To obtain the best controllers, the algorithm in [1] introduces two main components: local Nash minimizers and local denoising.

1) Local Nash Minimizers: [1] shows that an optimal controller must be a local Nash minimizer. Using calculus of variation, the first order condition (FOC) and the second order condition (SOC)

$$\frac{\delta \mathcal{J}[U]}{\delta u_m}(y_m) = 0, \quad \frac{\partial}{\partial u_m}\frac{\delta \mathcal{J}[U]}{\delta u_m}(y_m) \geq 0$$

are then derived for local Nash minimizers. Combining FOC and SOC, the algorithm in [1] repeats revised Newton's method to seek for a local Nash minimizer.

2) Local Denoising: The most important observation given by [1] is that finding local Nash minimizers numerically would land in a "noisy" controller. As a result, [1] introduces the idea of "denoising," i.e., for each u_m, we denoise for all y_m by

$$u_m(y_m) \leftarrow \underset{u_m(y):y\in B_r(y_m)}{\operatorname{argmin}} C_m(u_m(y), y_m),$$

where $B_r(a)$ is a ball centering at a with radius r.

B. Limitations

Although the algorithm in [1] is demonstrated effective for Witsenhausen's counterexample and it is applicable to other similar problems such as inventory control, there are still few issues left by [1]. First, albeit a universal approach to finding local Nash minimizers, calculus of variation is not a simple idea/operation for the people who know little about functional analysis. For local denoising, [1] obtains functions C_m by observation. It would be more rigorous to have a standard procedure to derive C_m. Meanwhile, despite its great performance in terms of the final cost it achieves, the algorithm runs slow in practice. And it is not clear how the termination criterion used in [1] can be easily generalized for arbitrary problems.

III. Generic Algorithm Design

In this section, we study UCP and illustrate how to overcome the limitations stated in Section II-B so that we can improve the ideas in [1] to solve UCPs.

A. Marginal Cost Functions

We start the analysis with Lemma 1, which is a necessary condition for optimal U, followed by the derivation of the *marginal cost function* C_m.

Lemma 1. *If U minimizes $\mathcal{J}$, we have*

$$u_m(y_m) = \underset{u\in\mathbb{R}}{\operatorname{argmin}} L_m(u, y_m)$$

almost everywhere for all $m \in M$.

The lemma can be derived from (1) and the derivation is straightforward. Suppose Lemma 1 is not true, there must exist some $\Delta > 0$ such that

$$L_m(u_m(y_m), y_m) \geq \Delta + \min_{u\in\mathbb{R}} L_m(u, y_m)$$

over some set Y_Δ with non-zero measure $\mathcal{M}(Y_\Delta)$. As such, we can set $u_m(y_m)$ as in Lemma 1 for all $y_m \in Y_\Delta$, which results in the reduction of the functional value by at least

$\mathcal{M}(Y_\Delta)\Delta > 0$. However, it leads to a contradiction as U minimizes $\mathcal{J}$.

Lemma 1 relates the optimal $u_m(y_m)$ with $L_m(u, y_m)$. However, a UCP is usually specified by $\mathcal{J}$ and its decomposition to $L_m(u, y_m)$ is in general not unique. We would prefer to base our solver on Lemma 1 with respect to a more deterministic expression than L_m. Therefore, we derive the *marginal cost function* C_m as follows.

From (1), we know

$$L_m(u_m(y_m), y_m) = \frac{\partial \mathcal{J}[U]}{\partial y_m} - \frac{\partial \mathcal{R}_m[U_{-m}]}{\partial y_m}$$

which implies that

$$L_m(u, y_m) = \left(\frac{\partial \mathcal{J}[U]}{\partial y_m} - \frac{\partial \mathcal{R}_m[U_{-m}]}{\partial y_m} \right)\Bigg|_{u_m(y_m)=u}.$$

Substitute it into Lemma 1, we have

$$\begin{aligned} u_m(y_m) &= \operatorname*{argmin}_{u\in\mathbb{R}} \left(\frac{\partial \mathcal{J}[U]}{\partial y_m} - \frac{\partial \mathcal{R}_m[U_{-m}]}{\partial y_m} \right)\Bigg|_{u_m(y_m)=u} \\ &= \operatorname*{argmin}_{u\in\mathbb{R}} \frac{\partial \mathcal{J}[U]}{\partial y_m}\Bigg|_{u_m(y_m)=u} \end{aligned}$$

because $\mathcal{R}_m[U_{-m}]$ does not depend on u. Thus, we define the *marginal cost function* C_m for each $m \in M$ by

$$C_m(u, y_m) = \frac{\partial \mathcal{J}[U]}{\partial y_m}\Bigg|_{u_m(y_m)=u}$$

and we can rephrase Lemma 1 as

Corollary 1. *If U minimizes $\mathcal{J}$, we have*

$$u_m(y_m) = \operatorname*{argmin}_{u\in\mathbb{R}} C_m(u, y_m)$$

almost everywhere for all $m \in M$.

Now, once we have the objective $\mathcal{J}$ specified, C_m can be derived using simple partial derivation. Actually, the definition of C_m also matches with the observation in [1].

B. Local Update and Partial Exhaustion

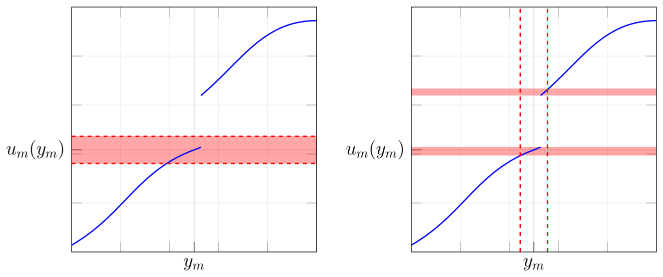
Corollary 1 is not only a necessary constraint but also a way to improve a non-optimal controller. Essentially, we can always improve a non-optimal solution by

$$u_m(y_m) \leftarrow \operatorname*{argmin}_{u\in\mathbb{R}} C_m(u, y_m).$$

To do so, we need to find the minimizer of $C_m(u, y_m)$.

To find the minimizer, the most effective method is to exhaust all possible $u \in \mathbb{R}$ and choose the best one. A full exhaustion is rarely practical as the computational cost would be intolerable. Instead, we would prefer computationally economic methods, such as *local update* and *partial exhaustion*.

Local update methods include gradient descent, Newton's method, and their variations. Those methods rely on local information at y_m, such as derivatives, to update u_m. Partial exhaustion takes a different approach. It avoids the heavy computational load in a full exhaustion by searching within only a subset of candidate values.



$\{u : u \in B_r(u_m(y_m))\}$ $\{u_m(y) : y \in B_r(y_m)\}$

(a) Local update (b) Partial exhaustion

Fig. 1. The candidate sets used in different techniques are marked by shaded areas. We depict the function u_m by the thick blue line and mark the range of B_r by red dashed lines. The candidate set can be disjoint in partial exhaustion when $u_m(y_m)$ is discontinuous.

Not surprisingly, these two methods correspond to the two main components in the algorithm in [1]. Since

$$\begin{aligned} \frac{\delta \mathcal{J}[U]}{\delta u_m}(y_m) &= C'_m(u_m(y_m), y_m), \\ \frac{\partial}{\partial u_m}\frac{\delta \mathcal{J}[U]}{\delta u_m}(y_m) &= C''_m(u_m(y_m), y_m), \end{aligned}$$

finding local Nash minimizers using the revised Newton's method in [1] is equivalent to minimizing the function $C_m(u, y_m)$ at each y_m by a revised Newton's method.

On the other hand, local denoising is a partial exhaustion method that searches within the candidate set

$$\{u_m(y) : y \in B_r(y_m)\} \tag{2}$$

to minimize $C_m(u, y_m)$. This is an important feature of the algorithm in [1]: It searches over the domain instead of the range of the function u_m. Usually, a partial exhaustion method searches the argument value locally, i.e., it considers the candidate set

$$\{u : u \in B_r(u_m(y_m))\}.$$

Such a partial exhaustion method plays a similar role as a local update method since they both try to update

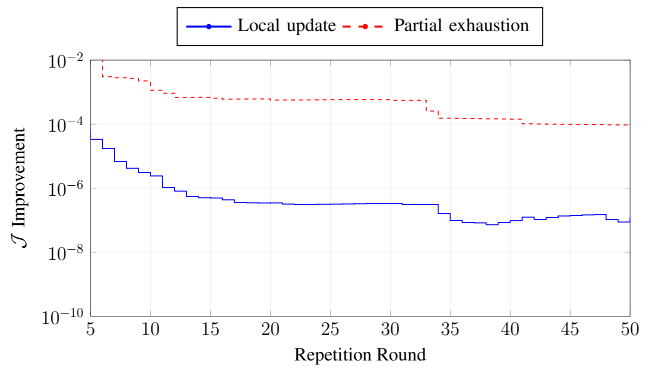
$$u_m(y_m) \leftarrow \operatorname*{argmin}_{u:u\in B_r(u_m(y_m))} C_m(u, y_m)$$

for some small r. As a result, we do not benefit much from combining the methods together. However, (2) can be a disconnected set when u_m is noncontinuous as in Fig. 1, which allows searching and "leaping" to another region.

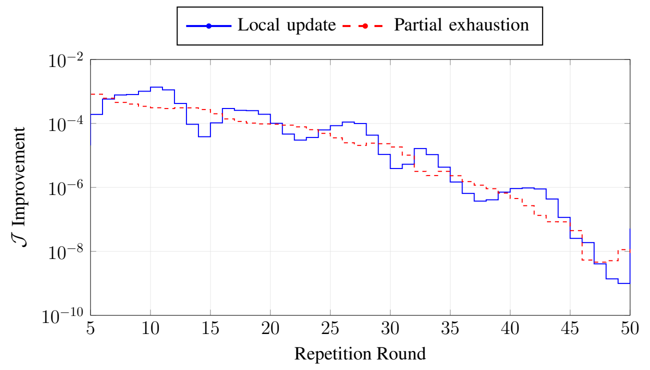
C. Adaptive Minimization

Both local update and partial exhaustion methods are adopted alternatively in the algorithm in [1]: In each repetition round, it denoises the controllers after repeating the revised Newton's method several times. This hybrid strategy converges to the best-known results, but it progresses quite slowly. The reason is that the two methods improve $\mathcal{J}$ in different ways and one may be more effective than the other at different searching phase.

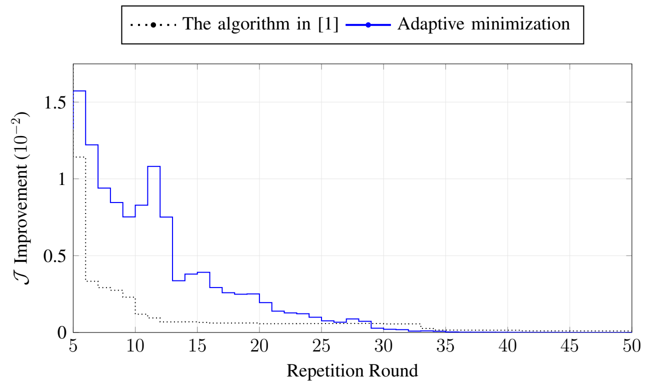
In Fig. 2(a), we consider the Witsenhausen's counterexample as in [1] and plot the average $\mathcal{J}$ value improvement



(a) The algorithm in [1] runs local update 19 times and performs 1 partial exhaustion per round. Partial exhaustion is more effective than local update.



(b) Adaptive minimization allocates 20 iterations in each round to the methods according to the improvement in the last round. As such, both methods improve $\mathcal{J}$ comparably.



(c) Comparison of the overall improvement. Adaptive minimization outperforms the algorithm in [1] significantly.

Fig. 2. The improvement of the objective functional $\mathcal{J}$ in the Witsenhausen's counterexample for each method. Each round has 20 iterations.

of the local update method (revised Newton's method) and the partial exhaustion method (denoising) for a series of repetition rounds, each round consisting of 19 local update iterations and 1 partial exhaustion iteration. Fig. 2(a) shows that partial exhaustion performs much better than local update, but local update runs much more times than partial exhaustion. As a result, the overall improvement at each round is low as in Fig. 2(c).

To improve the efficiency, we introduce *adaptive minimization*. The basic idea is that we will adapt the number of iterations according to the performance of the method. The more effective method gets more iterations to run. Fixing the total iterations to be 20 per round, we perform adaptive minimization and Fig. 2(b) shows that the average improvements of the two methods are comparable. Furthermore, the overall improvement is boosted by adaptive minimization as shown in Fig. 2(c).

Algorithm 1: Generic Solver

```
Input: Number of iterations per round $N$ and
       precision $p \geq 0$.
1:  Initialize controllers $U$.
2:  $\mathcal{J}_c \leftarrow \mathcal{J}[U]$.                          ▹ Initial objective value
3:  $I_L \leftarrow p$, $I_P \leftarrow p$.                      ▹ Initial improvements
4:  $N_L \leftarrow \lfloor \frac{N}{2} \rfloor$, $N_P \leftarrow N - N_L$.          ▹ Initial iterations
5:  while $I_L + I_P > p$ do                                   ▹ Main loop
6:      for $N_L$ iterations do
7:          for $m = 0$ to $M - 1$ do
8:              LocalUpdate($u_m$)
9:          end for
10:     end for
11:     $\mathcal{J}_c \leftarrow \mathcal{J}[U]$, $I_L \leftarrow |I_L - \mathcal{J}_c|$.
                                         ▹ Get local update improvement
12:     for $N_P$ iterations do
13:         for $m = 0$ to $M - 1$ do
14:             PartialExhaustion($u_m$)
15:         end for
16:     end for
17:     $\mathcal{J}_c \leftarrow \mathcal{J}[U]$, $I_P \leftarrow |I_P - \mathcal{J}_c|$.
                                   ▹ Get partial exhaustion improvement
18:     $N_L = \min\left\{\max\left\{\left\lfloor \frac{I_L N}{I_L + I_P} \right\rfloor, 1\right\}, N - 1\right\}$,
        $N_P \leftarrow N - N_L$.                           ▹ Adaptive minimization
19: end while
```

D. Generic Algorithm

We combine the methods described in Section III-B and Section III-C to construct Algorithm 1 for UCP. In summary, Algorithm 1 approximates the controllers U by step functions over a given sample range. Then, it adaptively minimizes $\mathcal{J}[U]$ using local update and partial exhaustion methods. Adaptive minimization is repeated until some precision factor is met. The whole procedure can be partitioned into five main parts, and we give more detailed descriptions below.

1) Initialization (line 1 – 4): Given some sampling range $[a, b] \subseteq \mathbb{R}$ and number of samples d, we create a linearly spaced vector $\mathbb{Y} \in \mathbb{R}^d$ spanning over $[a, b]$ as the domain. Each controller u_m is then approximated by a function mapping $\mathbb{Y}$ to $\mathbb{R}$. As suggested in [1], we initialize u_m by an identity map:

$$u_m(y_m) \leftarrow y_m \quad \text{for all} \quad y_m \in \mathbb{Y}.$$

Also, we initialize the current objective value $\mathcal{J}_c$, improvement I_L, I_P, and iterations N_L, N_P for adaptive minimization. We use subscript L to denote the variable for local update and subscript P for partial exhaustion.

2) Main Loop (line 5): We repeat adaptive minimization in the main loop, and we adopt a unified termination criterion: The loop terminates when the overall improvement in the current round is smaller than the precision p. This criterion improves upon the one adopted in [1] as it is valid regardless of the objective $\mathcal{J}$.

3) Local Update (line 6 – 11): In this phase, we aim to solve

$$u_m(y_m) \leftarrow \operatorname*{argmin}_{u:u\in B_r(u_m(y_m))} C_m(u, y_m)$$

for all $y_m \in \mathbb{Y}$ using some local update methods.

In our implementation, we adopt a modified Newton's method slightly different from the one in [1]: We only apply Newton's method

$$u_m(y_m) \leftarrow u_m(y_m) - \frac{C'_m(u_m(y_m), y_m)}{C''_m(u_m(y_m), y_m)}$$

when

$$C''_m(u_m(y_m), y_m) > 0$$

which implies a local minimum. Otherwise, we apply simple gradient method

$$u_m(y_m) \leftarrow u_m(y_m) - \tau C'_m(u_m(y_m), y_m)$$

where τ is the given step size.

4) Partial Exhaustion (line 12 – 17): We perform partial exhaustion based on the sampled candidate set (2):

$$u_m(y_m) \leftarrow \operatorname*{argmin}_{u_m(y):y\in B_r(y_m)\cap\mathbb{Y}} C_m(u_m(y), y_m)$$

for all $y_m \in \mathbb{Y}$. Essentially, it is equivalent to the local denoising procedure with denoising radius r. The radius r should be chosen such that it allows the algorithm to examine at least 1 sample point each direction along $\mathbb{R}$. Meanwhile, r should not be too large as the larger r leads to higher computation load and potentially traps the local update phase around suboptimal local minima. In our implementation, we choose r such that we check 2 points per direction.

5) Adaptive Minimization (line 18): We perform a per-round adaptive minimization by updating the iterations according to the improvements in the last round. Starting with fair sharing as in line 11, we calculate the improvements gained from each of the methods by line 11 and 17. In line 18, we allocate the N iterations in the next round proportional to the contribution of each method while ensuring each method will be run at least once.

IV. Performance

To demonstrate the performance, we implement Algorithm 1 in two different versions: the single-threaded solver in C++ (denoted by *Single*) and the parallelized one in NVIDIA CUDA (denoted by *Parallel*). The solvers are open sourced at [10]. The solvers are run under the number of iterations per round $N = 20$ and the precision $p = 10^{-10}$.

We compare our solver with the algorithm in [1] on the Witsenhausen's counterexample, for which the formulation

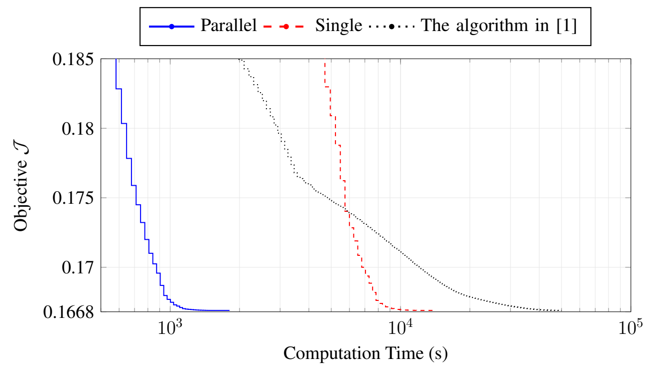


Fig. 3. Comparison of the convergence time among different methods. the algorithm in [1] converges faster in the beginning but it converges much slower when approaching to the optimum. On the contrary, both single-threaded and parallelized versions of Algorithm 1 converge much faster towards the optimum.

is deferred to Section V-A, under the parameters $k = 0.2$ and $\sigma = 5$.

For fair comparison, we impose the same termination criterion and total iterations per round for the algorithm in [1]: It performs local Nash minimizing 19 times followed by 1 local denoising. All experiments are run on a desktop with Intel Xeon W-2145 (CPU), NVIDIA Quadro P1000 (GPU), and Samsung 970 EVO NVMe M.2 (SSD).

A. Parallelization

To reduce the computation time, we examine Algorithm 1 and identify the parts that can be parallelized.

In both the local update and partial exhaustion phases, we revise u_m one at a time, which may change $C_{m'}$ for some $m' \in M, m' \neq m$. As a result, such a dependency prevents the computation being done in parallel. On the other hand, updating $u_m(y_m)$ does not affect C_m, and hence the calculation for each $y_m \in \mathbb{Y}$ can be parallelized to reduce the computation time.

B. Computation Time

Fig. 3 shows how fast the solvers and the algorithm in [1] converges along the computation time under 14000 sample points. the algorithm in [1] converges quickly at the beginning since it computes u_1 directly through a closed form expression. On the contrary, Algorithm 1 does not utilize any closed form expression. It simply searches for u_0 and u_1 using local update and partial exhaustion. Although the algorithm in [1] is more effective in computing u_1, adaptive minimization allows Algorithm 1 to select and use the more effective method. As a result, the single-threaded version converges slower at the beginning, but it improves $\mathcal{J}$ much more effectively than the algorithm in [1] when getting closer to the limit. After parallelization, the convergence speed is further improved by $10\times$, shifting the curve to the left in Fig. 3.

C. Scalability

Another property we examine is scalability. Scalability is important for a numerical method as it implies how precise

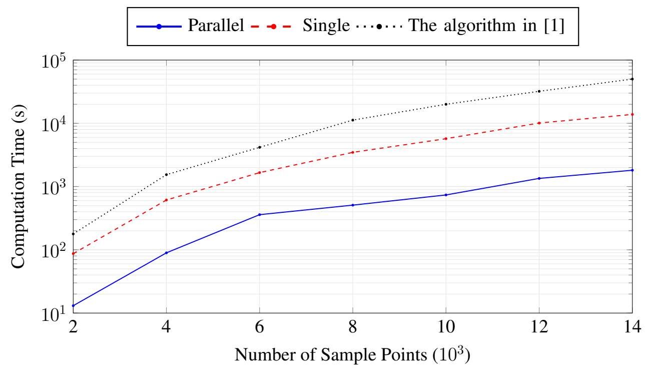


Fig. 4. Algorithm 1 scales much better than the algorithm in [1]. The parallelized version generates the controllers $30\times$ faster than the algorithm in [1].

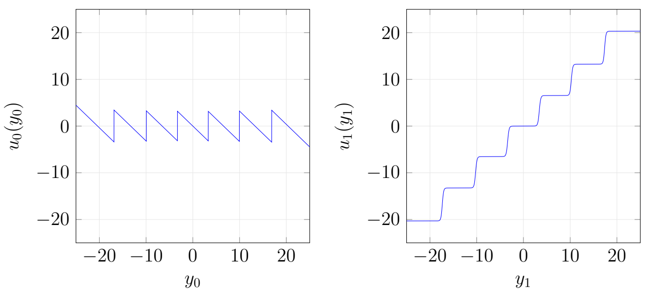


Fig. 5. Controllers for Witsenhausen's counterexample ($k = 0.2, \sigma = 5$), $\mathcal{J}[U] = 0.166897$.

the approximation can be and how large the problem the solver can handle. To demonstrate the scalability, we vary the number of sample points for each controller and measure the computation time needed before reaching the required precision level. In Fig. 4, the single-threaded solver runs about $3\times$ faster than the algorithm in [1]. After parallelization, the computation time is further reduced by $10\times$, and hence it enjoys $30\times$ performance improvement over the algorithm in [1].

V. Examples

We apply our solver to three different examples: Witsenhausen's counterexample (Section V-A), zero-delay source-channel coding (Section V-B), and inventory control (Section V-C). Since inventory control problem imposes additional constraints on the controllers, we discuss how our solver might be generated for the problem with constraints.

A. Witsenhausen's Counterexample

Witsenhausen's counterexample [2] aims to minimize the objective

$$\min\ \mathcal{J}[U] = \mathbb{E}\left[k^2 u_0(y_0)^2 + x_2^2\right]$$

subject to the system dynamic

$$\begin{aligned} x_1 &= x_0 + u_0(y_0), & y_0 &= x_0, \\ x_2 &= x_1 - u_1(y_1), & y_1 &= x_1 + w, \end{aligned}$$

where $x_0 \sim \mathcal{N}(0, \sigma^2)$ and $w \sim \mathcal{N}(0, 1)$.

We use our solver to find controllers in Fig. 5. The result is very close to the best-known controller in [1].

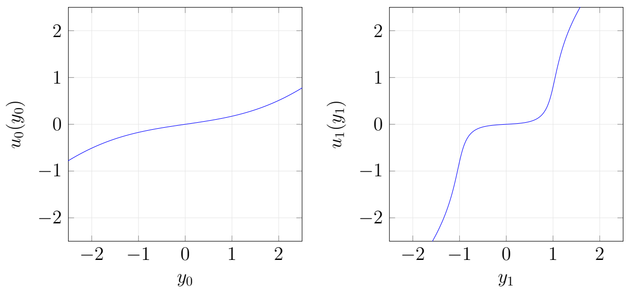


Fig. 6. Controllers for zero-delay source-channel coding problem ($\lambda = 2$), $\mathcal{J}[U] = 0.890756$.

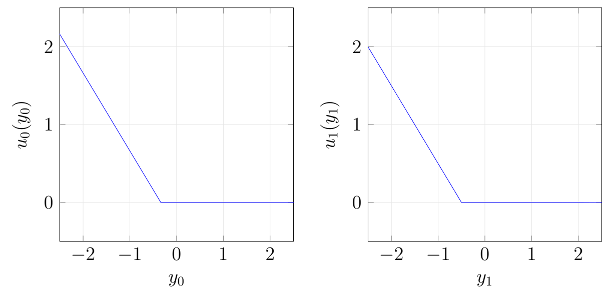


Fig. 7. Controllers for the inventory control problem using the same setting as in [1].

B. Zero-Delay Source-Channel Coding

The zero-delay source-channel coding problem [11] has the objective

$$\min\ \mathcal{J}[U] = \mathbb{E}\left[\lambda u_0(x_0)^2 + (u_1(x_1) - x_0)^2\right]$$

and the system dynamic

$$y_0 = x_0, \quad x_1 = u_0(y_0) + w, \quad y_1 = x_1,$$

where $x_0 \sim \mathcal{N}(0, 1)$ and $w \sim \mathcal{U}(-1, 1)$.

Fig. 6 shows the non-linear controllers found by our solver.

C. Inventory Control and Constrained Controller

The inventory control problem has the objective

$$\min\ \mathcal{J}[U] = \mathbb{E}\left[\sum_{m=0}^{M} \xi u_m(x_m) + \gamma(x_{m+1})\right]$$

where γ is some cost function attaining the minimum at 0.

The system dynamic is

$$y_m = x_m, \quad x_{m+1} = x_m + u_m(x_m) - w_m,$$

where $x_0 \sim \mathcal{U}(-1, 1)$ and $w_m \sim \mathcal{U}(-1, 1)$ for all $m \in M$.

The inventory control problem imposes an additional constraint: $u_m \geq 0$. We can enforce the constraint by performing $u_m \leftarrow \max\{0, u_m\}$ after each local update and partial exhaustion. With the same settings as in [1], our solver finds the same controllers as in Fig. 7. It is reported in [1] that this enforcement successfully finds the optimal controllers. Therefore, we conjecture that it is possible to generalize this generic UCP solver for the problems with constraints by projecting the result back to the feasible region after each local update and partial exhaustion.

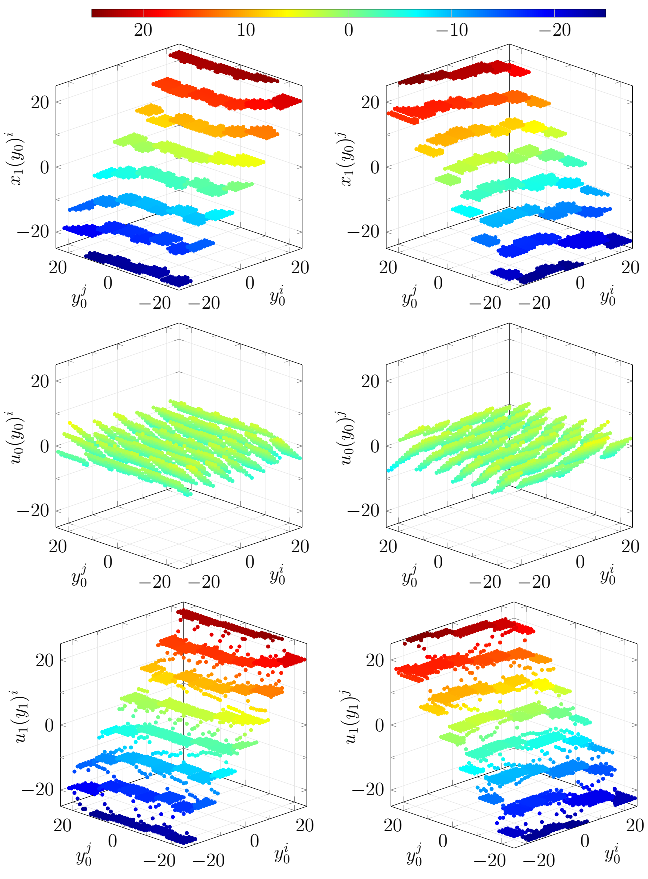


Fig. 8. State x_1 and controllers u_0 and u_1 for 2-dimensional Witsenhausen's counterexample ($k = 0.2, \sigma = 5$). We denote the two dimensions by superscripts i and j. With only 25600 sample points over y_0 and y_1, we can obtain the result $\mathcal{J}[U] = 0.166719$, which is close to the best known one 0.1527 by machine learning and sophisticated heuristics in [13].

D. Multi-Dimensional Controller: 2-Dimensional Witsenhausen's Counterexample

We can also apply the solver to multi-dimensional controllers. For example, 2-dimensional Witsenhausen's counterexample [12], [13] aims to minimize the objective

$$\min \ \mathcal{J}[U] = \frac{1}{2}\mathbb{E}\left[k^2\|u_0(y_0)\|^2 + \|x_2\|^2\right]$$

under the same state dynamic as the Witsenhausen's counterexample [2]. The difference is that the state x, controller u, and output y are all 2-dimensional vectors.

We apply our solver with only $160 \times 160 = 25600$ sample points over y_0 and y_1 and obtain the results in Fig. 8. The resulting objective value is very close to the best-known one in [13].

VI. Conclusion

The paper examines the unconstrained control problems and proposes an effective generic solver which can serve as the benchmark for the future UCP research. On the other hand, a lot of control problems do impose constraints on either the states of the controllers. As a result, it would be of interest to generalize the solver to constrained control problems.

References

[1] S.-H. Tseng and A. Tang, "A local search algorithm for the Witsenhausen's counterexample," in *Proc. IEEE CDC*, dec 2017.

[2] H. S. Witsenhausen, "A counterexample in stochastic optimum control," *SIAM J. Control*, vol. 6, no. 1, pp. 131–147, 1968.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. NIPS*, 2012, pp. 1097–1105.

[4] I. Goodfellow *et al.*, "Generative adversarial nets," in *Proc. NIPS*, 2014, pp. 2672–2680.

[5] M. Abadi *et al.*, "Tensorflow: A system for large-scale machine learning," in *Proc. USENIX OSDI*, 2016, pp. 265–283.

[6] Y. Turakhia, G. Bejerano, and W. J. Dally, "Darwin: A genomics co-processor provides up to 15,000× acceleration on long read assembly," in *Proc. ASPLOS*. ACM, 2018, pp. 199–213.

[7] N. Li, J. R. Marden, and J. S. Shamma, "Learning approaches to the Witsenhausen counterexample from a view of potential games," in *Proc. IEEE CDC*, 2009, pp. 157–162.

[8] J. Karlsson *et al.*, "Iterative source-channel coding approach to Witsenhausen's counterexample," in *Proc. IEEE ACC*, 2011, pp. 5348–5353.

[9] M. Mehmetoglu, E. Akyol, and K. Rose, "A deterministic annealing approach to Witsenhausen's counterexample," in *Proc. IEEE ISIT*, 2014, pp. 3032–3036.

[10] UCP solver. [Online]. Available: https://github.com/shih-hao-tseng/UCP-Solver

[11] E. Akyol *et al.*, "On zero-delay source-channel coding," *IEEE Trans. Inf. Theory*, vol. 60, no. 12, pp. 7473–7489, 2014.

[12] P. Grover, S. Y. Park, and A. Sahai, "Approximately optimal solutions to the finite-dimensional Witsenhausen counterexample," *IEEE Trans. Autom. Control*, vol. 58, no. 9, pp. 2189–2204, 2013.

[13] V. Subramanian *et al.*, "Some new numeric results concerning the Witsenhausen counterexample," in *Proc. Allerton*. IEEE, 2018, pp. 413–420.