A Library for Constraint Consistent Learning

Jeevan Manavalan\textsuperscript{a,+}†, Yuchen Zhao\textsuperscript{a†}, Prabhakar Ray\textsuperscript{a}, Hsiu-Chin Lin\textsuperscript{b} and Matthew Howard\textsuperscript{a}

\textsuperscript{a}King’s College London, Bush House, Strand Campus, 30, Aldwych, London WC2B 4BG - United Kingdom; \textsuperscript{b}McGill University, McConnell Engineering Building, Room 6403480 University Street, Montreal, Quebec, Canada, H3A 0E9

This paper introduces the first, open source software library for Constraint Consistent Learning (CCL). It implements a family of data-driven methods that are capable of (i) learning state-independent and -dependent constraints, (ii) decomposing the behaviour of redundant systems into task- and null-space parts, and (iii) uncovering the underlying null space control policy. It is a tool to analyse and decompose many everyday tasks, such as wiping, reaching and drawing. The library also includes several tutorials that demonstrate its use with both simulated and real world data in a systematic way. This paper documents the methods contained within the library, including the implementations of said methods in tutorials and associated helper methods. The software is made freely available to the community, to enable code reuse and allow users to gain in-depth experience in statistical learning in this area.

\textbf{Keywords:} Software Library; Constraints; Learning from Demonstration; Model Learning

1. Introduction

Constraint Consistent Learning (CCL) is a family of methods for learning different parts of the equations of motion of redundant and constrained systems in a data-driven fashion [1-3]. It is able to learn representations of self-imposed or environmental constraints [3-5], decompose the movement of redundant systems into task- and null-space parts [2, 6], and uncover the underlying null space control policy [7, 8]. CCL enables:

1. Learning the constraints encountered in many everyday tasks through various representations.
2. Learning the underlying control behaviour from movements observed under different constraints.

In contrast to many standard learning approaches that incorporate \textit{the whole of the observed motions into a single control policy estimate} [9], CCL separates the problem into learning (i) the constraint representations and (ii) the underlying control policy. This provides more flexibility to the robot in the reproduction of the behaviour, for example, in face of demonstration data of one behaviour recorded under different constraints, CCL can learn a single policy that generalises across the constraints [8].

In terms of (1), the type of constraints may fall into the categories of \textit{state independent} or \textit{state dependent} constraints. For example, for state-space represented as end-effector coordinates, when wiping a table (see Fig. 1(b)), the flat surface acts as a hard restriction on the actions available (motions perpendicular to the surface will be eliminated by the constraint), regardless of where the effector is located on the surface, so represents a state independent constraint. When wiping or stirring soup in a bowl (see Fig. 1(c)), the curved surface introduces a state dependency in the constraint, since the restriction of motion is dependent on the location of the effector. The ability to predict how the constraints can...
influence the outcomes of control actions can speed-up learning of the new skills and enhance safety (e.g., the learned constraints can prevent exploration actions that cause excessive force). Furthermore, the availability of constraint knowledge can reduce the dimension of the search space when optimising behaviours [10].

In terms of (2), the observable movements may contain partial information about the control policy masked by the constraint, or higher priory task objectives [2]. For example, in a reaching task (see Fig. 1(a)), humans move their arms towards the target (primary task) while minimising effort, for instance, by keeping the elbow low (secondary objective). The imperative of executing the primary task represents a self-imposed constraint, that restricts the secondary objective. Interaction with environmental constraints, can also mask the actions applied by the demonstrator. For example, when grasping an object on a table such as a pen, the fingers slide along the table surface resulting in a discrepancy between the observed motion and that predicted by the applied actions. CCL can help uncover these components, enabling the intended actions to be reconstructed and applied to new situations [11].

This paper introduces the CCL library as a community resource for learning in the face of kinematic constraints, alongside tutorials to guide users who are not familiar with learning constraint representations and underlying policies. As its key feature, CCL is capable of decomposing many everyday tasks (e.g., wiping [3], reaching [2] and gait [12]) and learn subcomponents (e.g., null space components [2], constraints [3] and unconstrained control policy [8]) individually. The methods implemented in the library provide means by which these quantities can be extracted from the data, based on a collection of algorithms developed since CCL was first conceived in 2007 [13]. It presents the first unified Application Programmers Interface (API) implementing and enabling use of these algorithms in a common

Figure 1.: Typical examples in CCL. (a) Three different tasks: moving the finger to an \( x, y, z \) position, liquid pouring and wiping a surface. In each case, redundancy is resolved in the same way. (b) A table wiping task is subject to a state-independent flat environment constraint using a robot arm. (c) A bowl wiping task subject to a state-dependent curvature constraint.
Figure 2.: Policy based modelling. Demonstration data is given as pairs of states and actions which are used to form a model of the policy represented here as a vector field. To then reproduce the movement from the resulting model, a corresponding mapping is defined which maps the demonstrators states $x$ and actions $u$ to the imitators $x'$ and $u'$ [25].

2. Background

CCL addresses the problem of learning from motion subject to various constraints for redundant systems. The behaviour of constrained and redundant systems is well-understood from an analytical perspective [14], and a number of closely related approaches have been developed for managing redundancy at the levels of dynamics [15], kinematics [16, 17], and actuation [18, 19]. However, a common factor in these approaches is the assumption of prior knowledge of the constraints, the control policies, or both. CCL addresses the case where there are uncertainties in these quantities and provides methods for their estimation.

2.1. Direct Policy Learning

One of the most fundamental approaches to modelling movements is through Direct Policy Learning (DPL) [9, 24, 1]. Given data of observed movements as states $x$ and actions $u$, this approach, usually formulated as a supervised learning problem, estimates a function of the policy $\pi$ through direct regression, which is a mapping between states and actions

$$u = \pi(x), \quad \pi : \mathbb{R}^P \rightarrow \mathbb{R}^Q$$

where $x \in \mathbb{R}^P$ and $u \in \mathbb{R}^Q$ are the state and action spaces, respectively. Once demonstration data is used in learning, this results in a reproduced movement by an imitator as illustrated in Fig. 2. As shown, data is given as pairs of states and actions. A model of the policy is then produced which results in corresponding pairs of state-dependent actions, with the goal being to approximate the policy as close as possible. One way to obtain a mapping between the states and actions is by minimising the error of

$$E_{DPL} = \sum_{n=1}^{N} ||u_n - \bar{\pi}_n||^2$$

where $\bar{\pi}$ is an estimated function of the policy $\pi$ [1]. As pointed out in [1], when learning policies through DPL, demonstrations are either performed in free space such as doing sign language or under consistent

---

1 For brevity, here, and throughout the thesis, the notation $a_n$ may be used to denote the quantity $a$ evaluated on the $n$th sample. For example, if $a$ is a vector quantity computed from the state $x$, then $a_n = a(x_n)$.
constraints such as interacting in an unchanging environment with obstacles. These two categories that fall under demonstrations that follow DPL are referred to as unconstrained or consistently constrained, respectively. Moreover, DPL being more of a traditional and older planning algorithm, is deterministic in nature and suffers from poor generalisation when dealing with undemonstrated states. It simply assumes that each action leads to a particular state, however this ideology does not always apply in the real world due to the ever changing environment \[24\]. Nonetheless, there are extensions to this approach to mitigate its lack of robustness to generalisation such as through exploration policies which typically implements a reward function. However, this too brings up another issue involving finding a balance between Exploration vs. Exploitation, i.e., deciding how much more varying data is required compared to being satisfied with the policy modelled up to a certain point in time, with the risk lying in excess time spent at improving a policy not justified by the rate of improvement of said policy. Ultimately, other approaches exist which focus on generalisation without the same shortcomings.

2.2. Constraint Formalism

As discussed in [2.1] there are implicit assumptions set by DPL regarding demonstrations such that these are expected to be either unconstrained or consistently constrained [11]. However, this makes it infeasible when it comes to modelling many everyday tasks due to variability in the real world. Interactions with the ever changing environment tend to involve some type of constraint subject to variability and this may have an impeding factor to the success of a task, such as when walking on uneven terrain or how a surface of a table limits movement of a hand at various angles during a wiping task (Fig. 3) [6]. Not being able to take such variability into account which should be expected in a real world environment shows a limitation of DPL [11]. While constraints are generally viewed as an impeding factor to a task, they can also be used to aid systems in interactions such as by exploiting environmental constraints in order to help facilitate grasping [34].

Starting with the DPL based approach (2.1) which describes autonomous systems of the form

\[
u(t) = \pi(x(t)), \quad \pi : \mathbb{R}^P \to \mathbb{R}^Q
\]

where demonstration data is given as \( N \) pairs of observed states \( x \in \mathbb{R}^P \) and actions \( u \in \mathbb{R}^Q \), and \( \pi \) is a direct mapping between the two. When including a constraint, the model is subject to a set of \( P \)-dimensional constraints where \( P < Q \)

\[
A(x)u(x) = b(x)
\]

where \( x \in \mathbb{R}^P \) represents state and \( u \in \mathbb{R}^Q \) represents the action. \( A(x) \in \mathbb{R}^{S \times Q} \) is the constraint matrix which projects the task space policy onto the relevant part of the control space. The vector term \( b(x) \in \mathbb{R}^S \), if present, represents constraint-imposed motion (in redundant systems, it is the policy in the task space).
Inverting (2), results in the relation
\[ u(x) = A(x)\dagger b(x) + N(x)\pi(x) \quad (3) \]

\( A^\dagger = A^\top (AA^\top)^{-1} \) is the Moore-Penrose pseudo-inverse of \( A \). \( N(x) := I - A(x)\dagger A(x) \in \mathbb{R}^{Q \times Q} \) is the null space projection matrix that projects the null space policy \( \pi(x) \) onto the null space of \( A \). Note that, the constraint can be state dependent (\( A(x) \)) or state independent (\( A \)). Here, \( I \in \mathbb{R}^{Q \times Q} \) denotes the identity matrix. \( v(x) \equiv A(x)\dagger b(x) \) and \( w(x) \equiv N(x)\pi(x) \) are termed the task-space and the null-space component, respectively. The task space can refer to the DoF (Degree of Freedom) required for learning approaches \[ n \] that do not consider the composition of the data in terms of the constraints are prone to poor performance and modelling errors. For example, applying direct regression to learn policies where there are variations in the constraints can result in model averaging effects that risk unstable behaviour \[ n \]. This is due to factors such as (i) the non-convexity of observations under different constraints, and (ii) degeneracy in the set of possible policies that could have produced the movement under the constraint. However, increasingly more methods are being developed which overcome the aforementioned issues such as by learning what movements can be performed in the task space \[ n \] as well as others which separate the task and null space component \[ n \] and then learn the constraint from either the task space \[ n \] or null space \[ n \]. Providing an open-source collection of software tools suitable for application to this class of learning problems, can help those working in the field to avoid the common pitfalls and thus the CCL library is presented which implements CCL methods such that they can be applied directly to such problems. Implementation details are provided in \[ n \].

3. Learning Functions

A number of algorithms exist for the estimation of the quantities \( A, v, w \) and \( \pi \). The following provides a brief summary of the data structures, evaluation metrics and methods relating to CCL provided in the CCL library looking at the theoretical basis for each algorithm.

3.1. Data Structures

CCL methods are tested on data that is given as tuples \( \{x_n, u_n\}_{n=1}^{N} \) of observed states and constrained actions. It is assumed that all the actions \( u \) are generated using the same underlying policy \( \pi(x) \). In particular, \( u_n = A_n b_n + N_n \pi_n \), where \( A_n, b_n \) and \( \pi_n \) are not explicitly known. The observations are assumed to be grouped into \( K \) subsets of \( N \) data points, each (potentially) recorded under different (task or environmental) constraints.

Learning is more effective when the data contains sufficiently rich variations in one or more of the quantities defined in \( n \), since methods learn the consistency by teasing out the variations. For instance, when learning \( v \), variations in \( w \) are desirable \[ n \]. For learning \( \pi \), observations subject to multiple...
constraints (variation in \( A \)) are necessary \([1]\). For learning constraint \( A \), variations in \( \pi \) are desirable \([6]\).

### 3.2. Evaluation Criteria

For testing different aspects of the learning performance, a number of evaluation criteria have been defined in the CCL literature. These include metrics comparing estimation against the ground truth (if known), as well as those that provide a pragmatic estimate of performance based only on observable quantities. Table \([1]\) shows an overview of all key functions as well as their respective equations for learning and evaluation which are covered. Moreover, the implementation of these performance measures are discussed in \([4, 6]\).

#### 3.2.1. Normalised Projected Policy Error (NPPE)

If the ground truth constraint matrix \( A_n \) and unconstrained policy \( \pi_n \) are known, then the (NPPE) provides the best estimate of learning performance \([3]\). It measures the difference between the policy under the real constraints and the policy under the estimated constraints \([3]\). NPPE in this case is defined as:

\[
E_{PPPE} = \frac{1}{N} \sigma_u^2 \sum_{n=1}^{N} ||N\pi_n - \hat{N}\pi_n||^2
\]  

where \( N \) is the number of data points, \( \pi_n \) are samples from the policy, \( N \) is the true projection matrix and \( \hat{N} \) is the learnt projection matrix. The error is normalised by the variance of the observations under the true constraints \( \sigma_u^2 \). This method is only used for validating the results, as it requires the ground truth \( \pi_n \) and \( N \) which is assumed to be unknown.

#### 3.2.2. Normalised Projected Observation Error (NPOE)

In the absence of ground truth \( A_n \) and \( \pi_n \), the NPOE must be used \([3]\). In a practical setting, constraint learning may be performed without prior knowledge of the \( \pi_n \) or \( N \), thus the NPOE plays a vital role:

\[
E_{POE} = \frac{1}{N} \sigma_w^2 \sum_{n=1}^{N} ||w_n - \hat{N}w_n||^2
\]  

#### 3.2.3. Null space Projection Error (NPE)

To evaluate the predictions of the null-space component model \( \mathbf{\hat{w}}_k(x) \), the NPE can be used \([2]\). Note that, use of this function assumes knowledge of the ground truth \( w_n \).

\[
E_{PE} = \frac{1}{N} \sigma_p^2 \sum_{n=1}^{N} ||w_k,n - \hat{\pi}_k(x_n)||^2
\]  

#### 3.2.4. Normalised Unconstrained Policy Error (NUPE)

To evaluate the estimated unconstrained control policy model \( \pi(x) \), the NUPE is used \([8]\). It assumes access to the ground truth unconstrained policy \( \pi_n \) and works by directly comparing the true policy against the learnt policy

\[
E_{UPE} = \frac{1}{N} \sigma_p^2 \sum_{n=1}^{N} ||\pi_n - \hat{\pi}_n||^2
\]
Table 1.: Key Functions in the CCL Library

<table>
<thead>
<tr>
<th>Function Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learn Constraint</td>
<td>Learn Alpha</td>
<td>Learning state dependent projection $N$</td>
</tr>
<tr>
<td></td>
<td>Learn Lambda</td>
<td>Learning state dependent selection matrix $\Lambda$</td>
</tr>
<tr>
<td></td>
<td>Learn Nhat</td>
<td>Learning state independent constraint $N$</td>
</tr>
<tr>
<td>Learn Nullspace Component</td>
<td>Learn ncl</td>
<td>Learn the nullspace component of $u = v + w$</td>
</tr>
<tr>
<td>Learn unconstrained control</td>
<td>Learn CCL Pi</td>
<td>Learn null space policy</td>
</tr>
<tr>
<td>Policy</td>
<td>Evaluate Constraint error</td>
<td>Normalised projected policy error</td>
</tr>
<tr>
<td></td>
<td>Evaluate Nullspace component error</td>
<td>Nullspace projection error</td>
</tr>
<tr>
<td></td>
<td>Evaluate unconstrained control policy</td>
<td>Normalised unconstrained policy error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Normalised constrained policy error</td>
</tr>
</tbody>
</table>

3.2.5. Normalised Constrained Policy Error (NCPE)

To evaluate the estimated constrained control policy model $\tilde{\pi}(x)$, NCPE is used [8]. It assumes that the true $w_n$ is known. In some cases, variations in constraints may be insufficient to fully estimate the true policy $\pi$. In such cases, where constraints show little variation, it may not even be necessary to uncover the hidden components of the fully unconstrained policy as these can usually be eliminated by the constraints. Thus the following approach is suggested

$$E_{CPE} = \frac{1}{N\sigma^2} \sum_{n=1}^{N} ||w_n - N\tilde{\pi}_n||^2$$  (8)

which measures the difference between the data and the estimated policy, when the latter is projected by the same constraints as in the training data.

3.3. Learning the Constraint Matrix

Several methods for estimating the constraint matrix $A$ are presented. These assume that (i) each observation contains a constraint $A \neq 0$, (ii) the observed actions take the form $u = w = N\pi$, (iii) $u$ are generated using the same null space policy $\pi$, and (iv) $A$ is not explicitly known for any observation (optionally, features in form of candidate rows of $A$, may be provided as prior knowledge—see §3.3.2).

The methods for learning $A$ under these assumptions, are based on the insight that the projection of $u$ also lies in the same image space [3], i.e.,

$$(I - A^\dagger A)u = Nu = u.$$  (9)

If the estimated constraint matrix $\hat{A}$ is accurate, it should also obey this equality. Hence, the estimate can be formed by minimising the difference between the left and right hand sides of (9), as captured by the error function

$$E[\tilde{N}] = \sum_{n=1}^{N} ||u_n - \tilde{N}_n u_n||^2$$  (10)

where $\tilde{N}_n = I - \hat{A}_n^\dagger \hat{A}_n$. This can be expressed in simplified form

$$E[\tilde{A}] = \sum_{n=1}^{N} ||u_n^\top \hat{A}_n^\dagger \hat{A}_n u_n||^2.$$  (11)

Note that, in forming the estimate, no prior knowledge of $\pi$ is required. However, an appropriate repre-
sentation of the constraint matrix is needed. As outlined in §1, in the context of learning $A$, there are two distinct cases depending on whether the constraints are state-dependent or not.

### 3.3.1. Learning State Independent Constraints

The simplest situation occurs if the constraint is unvarying throughout the state space. In this case, the constraint matrix can be represented as the concatenation of a set of unit vectors

$$
\hat{A} = (\hat{a}_1^\top, \hat{a}_2^\top, \ldots, \hat{a}_S^\top)^\top
$$

(12)

where the $s$th unit vector $\hat{a}_s$ is constructed from the parameter vector $\theta \in \mathbb{R}^{Q-1}$ that represents the orientation of the constraint in action space (see [3] for details). A visualisation of the simplest case of a one-dimensional constraint in a system with two degrees of freedom where $A = \hat{a} \in \mathbb{R}^{1 \times 2}$ is shown in Fig. 4. It can be seen how $\theta$ (in this case, $\hat{a} = (\cos \theta, \sin \theta)$) corresponds to different estimates of the constraint’s associated null-space.

In [3], the use of this method is demonstrated in both simulation and a real world experiment. It is first evaluated in a toy example with the goal of learning the null-space projections from constrained data. For this, a two-dimensional system with a one-dimensional constraint is set-up. This constrained system is tested using three different control policies which include, a linear policy, limit-cycle and sinusoidal policy where in all cases errors for the NPPE and NPOE are below $10^{-5}$. This approach is then evaluated in the real world on a 7DoF Kuka Lightweight Robot (LWR-III) to see how it performs when learning from data with a higher dimensionality as well as with more complex constraints. All results for experiments following this set-up give a NPPE and NPOE less than $10^{-3}$. Further details on the theory behind this approach and its experiments can be found in [3].

### 3.3.2. Learning State Dependent Constraint

Another proposed method for learning a state-dependent constraint, i.e., the constraint $A(x)$ depends on the current state of the robot $x$. For learning $A(x)$, two scenarios may be encountered where (i) there is prior knowledge of the constraint in form of candidate rows (see [6]), or (ii) no prior knowledge is available.

In the case of (i),

$$
A(x) = \tilde{A}(x)\Phi(x).
$$

(13)

Note: While the pseudoinverse operation has conditioning problems when close to a rank-deficient configuration, the constraint learning methods deal with this during the inversion by ignoring all singular values below a certain threshold.

---

Figure 4.: Three one-dimensional unit vector constraints $\hat{a}$ where (a) $\theta = 0^\circ$, (b) $\theta = 45^\circ$, and (c) $\theta = 90^\circ$ (reproduced from [3]).
Φ(x) ∈ ℝ^{R×Q} is the feature matrix and Λ(x) ∈ ℝ^{S×R} is the selection matrix specifying which rows represent constraints. The feature matrix can take the form \( \Phi(x) = J(x) \), where \( J \) is the Jacobian mapping from the joint space to the end-effector task space. Similar to (12), \( \tilde{\lambda} \) represent constraints. The feature matrix can take the form \( \Phi = \Lambda(\theta)^\top \), \( \tilde{\lambda} \) a set of feature vectors in higher dimensional space. Substituting \( R_{\lambda} \) where

\[
\Lambda(x) = (\lambda_1(\theta)^\top, \lambda_2(\theta)^\top, \cdots, \lambda_S(\theta)^\top)^\top
\]

where \( \lambda_s \in \mathbb{R}^S \) is the \( s \)th dimension in the task space and \( \lambda_i \perp \lambda_j \) for \( i \neq j \). Parameter \( \theta_s \in \mathbb{R}^{S-1} \) is used for representing the constraint vector \( \lambda \). Each \( \theta_s \) is modelled as \( \theta_s = \omega_s \beta(x) \) where \( \omega_s \in \mathbb{R}^{(Q-1)×G} \) is the weight matrix, and \( \beta(x) \in \mathbb{R}^G \) is the vector of \( G \) basis functions that transform \( x \) into a set of feature vectors in higher dimensional space. Substituting \( \Lambda_n = \Lambda_n \Phi_n \), (11) can be written as

\[
E[\Lambda] = \sum_{n=1}^{N} ||u_n^\top (\Lambda_n \Phi_n)^\dagger \Lambda_n \Phi_n u_n||^2.
\]

The optimal \( \Lambda \) can be formed by iteratively searching the choice of \( \lambda_s \) that minimises (15).

In [6], experiments to validate this method are presented in a simulated toy example and three link planar arm as well as a real world humanoid hand. The three link system looks at constraints affecting the end-effector of a system where various combinations of constraints are applied on a 2D Cartesian space as well as the orientation of the system’s end-effector. Finally, it is also tested using an AR10 Robotic hand holding a remote of an air conditioner, in a problem where the task-space controls the pressing of a button using its thumb and the null-space controls moving the middle finger to a comfort posture. The goal of this experiment is to extract the task-space component in face of the distracting null-space movements so that the action of pressing a button can be generalised to other buttons (details can be found in [6]).

### 3.4. Learning the Task and Null Space Components

When the unconstrained control policy is subject to both constraint and some task space policy (\( b(x) \neq 0 \)), it is often useful to extract the task and null space components (\( v(x) \) and \( w(x) \), respectively) of the observed actions \( u(x) \). These assume that the underlying null-space policy and the task constraint are consistent.

Assuming that the underlying null-space policy \( \pi(x) \) and the task constraint \( \Lambda(x) \) are consistent across the data set, the null space component \( w(x) \) should satisfy the condition \( N(x)u(x) = w(x) \) as noted in [2], in this case, \( w(x) \) by minimising

\[
E[\tilde{w}] = \sum_{n=1}^{N} ||\tilde{P}_n u_n - \tilde{w}(x_n)||^2
\]

with \( \tilde{P}_n = \tilde{w}_n \tilde{w}_n^\top / ||\tilde{w}_n||^2 \), where \( \tilde{w}_n = \tilde{w}(x_n) \) projects an arbitrary vector onto the same space of \( w \), and \( u_n \) is the \( n \)th data point. This error function eliminates the task-space components \( v(x) \) by penalising models that are inconsistent with the constraints, i.e., those where the difference between the model, \( \tilde{w}(x) \), and the observations projected onto that model is large.

The approach is validated in simulation of a 2D system, three link planar arm as well as a 7DoF Kuka Lightweight arm. It is also tested in a comparative study against the at the time state-of-the-art DPL where it is shown to perform better when estimating null-space policies (for details, see [2]).
Figure 5.: A system diagram of the CCL library. The complete dependency tree can be found in the package documentation. The library has been written under three categories: learn A, learn null-space component and learn policy. The arrows links the dependencies of the implementations to the main functions.

3.5. Learning the Unconstrained Control Policy

Another situation where CCL plays a role is when there is a need to estimate the underlying unconstrained control policy. These assume either (i) \( b = 0 \) no additional task is involved or (ii) \( w \) is learnt using the method proposed in §3.4.

As shown in [8], an estimate \( \tilde{\pi}(x) \) can be obtained by minimising the inconsistency error

\[
E_\pi = \sum_{n=1}^{N} \| u_n - P_n \tilde{\pi}(x_n) \|_2^2; \quad P_n = \frac{u_n u_n^\top}{\| u_n \|_2^2}.
\]

The risk function (17) is compatible with many regression models (Details see [2]).

4. Tutorials

The API of the CCL library provides functions for a series of algorithms for the estimation of the quantities \( A, v, w \) and \( \pi \). Moreover, the CCL library package provides a number of tutorial scripts to illustrate its use. The following provides a summary of the implementation and multiple examples are provided which demonstrate learning using both simulated and real world data, including, learning in a simulated 2-link arm reaching task, and learning a wiping task. Included in this is a toy example which is described in detail, alongside a brief description of other included demonstrations which were tested with real world systems. Further details of the latter are included in the documentation.

To help put these functions into context, the majority of examples provided in this section are explained for use with a simple, illustrative toy example, in which a two-dimensional system is subject to a one-dimensional constraint. In this, the learning of (i) the null space components, (ii) the constraints, and (iii) the null space policy are demonstrated.

The toy example code has been split into three main sections for learning different parts of the equations. Detailed comments of the functions can be found in the Matlab script. The procedure for generating data is dependant on the part of the equation you wish to learn. Details can be found in the documentations.
4.1. Implementation

A system diagram of the CCL library is shown in Fig. [5]. The naming convention is following `ccl_xxx_yxx` indicating the category and functionality of the implementations. The following provides brief notes on the implementation languages and library installation, explains the data structures used in toolbox.

4.1.1. Language and Installation Notes

The library is implemented in both C and Matlab and is available for download from Github[1] The CCL library is provided under the GNU General Public License v3.0, and documentation is provided online[2]. To use the Matlab package, the user can just add the CCL library into the current path. Installation of the C package has been tested on Ubuntu 14.04 Linux systems and is made easy through use of the autoconf utility. For Windows systems, only the Matlab package is currently tested and supported. The C package's only dependency is on the 3rd-party, open-source GNU Scientific Library (GSL)[3].

A detailed explanation of the functions follows, alongside tutorial examples that are provided to enable the user to easily extend use of the library to their chosen application (see [4]).

4.2. Data Structures

The methods implemented in the toolbox work on data that is given as tuples \( \{x_n, u_n\}_{n=1}^N \) of observed states and constrained actions. The CCL library uniformly stores the data tuples in a data structure reflecting this problem structure. The latter includes the data fields incorporating samples of the input state (x), action (u), actions from unconstrained policy (π), task space component (v) and null space component (w). The major data structure and data fields are listed in Table 2.

4.3. Learning Null Space Components w

Here, guidance is given on how to use the library for learning W. This method is applicable to data when demonstrations consist of both v and w. As learning is performed on the null-space component, it is necessary to separate these. However, if all demonstrations are already recorded in the null-space, then data is given as w and it’s therefore not necessary to execute this step.

In the CCL library, this functionality is implemented in the function

\[
[\text{fun}, J] = \text{ccl_obj_ncl} \ (\text{model}, W, \text{BX}, U)
\]

where U is the observed actions U. The outputs are fun (function handle) and J (analytical Jacobian of the objective function [16]).

Minimisation of (16) penalises models that are inconsistent with the constraints, i.e., those where the difference between the model, \( \tilde{w}_k(x) \), and the observations projected onto that model is large (for details, see [2]). In the CCL library, this functionality is implemented in the function

\[
\text{model} = \text{ccl_learnv_ncl}(X, U, \text{model})
\]

where the inputs are X (input state), Y (observed actions combined with task and null space components) and model (model parameters). The output is the learnt model model.

A sample Matlab snippet is shown in Table 3 and explained as follows. Firstly, the user needs to generate training data given the assumption that w is fixed but v is varying in your demonstration. The unconstrained policy is a limit cycle policy [4]. For learning, the centre of the Radial Basis Functions are chosen according to K-means and the variance as the mean distance between the centres. A parametric model with 16 Gaussian basis functions is used. Then the null-space component model can be
Table 2.: Main data structures of the CCL library. dim* indicates the data fields for definition of the data dimensionality.

<table>
<thead>
<tr>
<th>Struct</th>
<th>Main Fields</th>
<th>Definition</th>
<th>Other Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>data X</td>
<td>X = (x₁, ..., x_N) ∈ ℝ^{P×N}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>U u</td>
<td>u = (u₁, ..., u_N) ∈ ℝ^{Q×N}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pi Π</td>
<td>Π = (π₁, ..., π_N) ∈ ℝ^{Q×N}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ts V</td>
<td>V = (v₁, ..., v_N) ∈ ℝ^{Q×N}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Un W</td>
<td>W = (w₁, ..., w_N) ∈ ℝ^{Q×N}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>model M</td>
<td>Nullspace component Model weighting parameters ω.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Σ</td>
<td></td>
<td></td>
<td>dim*</td>
</tr>
<tr>
<td>(θ₁, ..., θ_N*ₜ)</td>
<td>Normalised mean square error.</td>
<td>dim*</td>
<td></td>
</tr>
<tr>
<td>f_proj</td>
<td>Function handle for projection matrix.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TolFun</td>
<td>Tolerance for residual.</td>
<td></td>
<td>dim*</td>
</tr>
<tr>
<td>TolX</td>
<td>Tolerance for model parameters.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MaxIter</td>
<td>Maximum iterations for optimiser.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stats nmse</td>
<td>Normalised mean square error.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mse</td>
<td>Mean squared error.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>var</td>
<td>Variance.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.4. Learning Null Space Constraints A

Once data is in the correct format to proceed with learning, it is important to know how to select the appropriate functions from the library to learn a state-independent A and state dependent A(x), where b(x) = 0. If the provided data contains a linear constraint (i.e., state-independent), then the use of ccl_learn_nhat is suggested. On the other hand, if the constraint is non-linear (i.e., state-dependant), then it is necessary to use ccl_learn_alpha in order to correctly estimate the constraint.)

4.4.1. Learning State Independent Constraints

For learning a linear constraint problem, constant A is used and ccl_learn_nhat is implemented. The CCL library provides the function

```
optimal = ccl_learna_nhat(Un)
```

to form the estimate A in this scenario, where Un is the collated set of samples of u (see Table 2) and optimal is the learnt model. The C counterpart of this function is

```
void learn_nhat(const double *Un, const int dim_u, const int dim_n, NHAT_Model *optimal)
```

where the input argument Un and output argument optimal in the C library is identical with Matlab implementation. The arguments dim_u and dim_n are used for defining the dimensionality of the array Un.

The function works by searching for rows of the estimated constraint matrix A using the function

```
[model, stats] = ccl_learna_sfa (UnUn, Un, model, search)
```

Finally, the evaluation metrics NUPE and NPE are used to report the learning performance.

---

1The C library implements all learning functions, but for the remainder of the paper, Matlab functions will be used throughout for consistency.
Table 3.: Sample code snippet for learning null-space component from Matlab

```matlab
Data = ccl.data_gen(settings); 
model.c = ccl.math.gc(X, model.dim.basis); 
model.s = mean(mean(sqrt(ccl.math.distances(model.c, model.c))))^2; 
model.phi = @(x)ccl.basis_rbf(x, model.c, model.s); 
model = ccl.learnv_ncl(X, U, model); 
f_ncl = @(x) ccl.learnv_pred_ncl(model, x); 
NSp = f_ncl(X); 
NUPE = ccl.error_nupe(Un, Unp); 
NPE = ccl.error_npe(Un, Unp); 
```

the arguments model and search contain the initial model (untrained) model, parameters controlling the search, respectively and UnUn is the matrix \( W^\top W \) where \( W \) is a matrix containing samples of \( w \). This is pre-calculated prior to the function call for speed. The function returns the learnt model model and learning statistics stats (see Table 2).

A sample Matlab script is provided in Table 4 and explained as the following. Firstly, it simulates a problem where the user faces is learning a fixed constraint in which the system’s null-space controller \( \pi \) varies.

### 4.4.2. Learning State dependent Constraints

The objective function for (11) and (15) are embedded in function

```matlab
[fun] = ccl_obj_AVn(model,W,BX,RnUnUnn) 
```

where fun is the objective function handle, \( M \) is the model parameter \( \omega \), BX is the feature vectors in high dimension space \( \beta(x) \), and RnUnUnn is the pre-rotated \( UnUn_n \).

The implementation of case (i) discussed in §3.3.2 is

```matlab
[optimal] = ccl_learn_lambda(Un,X,Phi,options) 
```

where Phi is the feature matrix \( \Phi \).

The implementation of case (ii) where

\[
A(x) = (a_1(\theta)^\top, a_2(\theta)^\top, \cdots, a_S(\theta)^\top)^\top 
\]

is

```matlab
[optimal] = ccl_learn_alpha(Un,X,options) 
```

The non-linear parameter optimisation solver used for learning \( \theta_s \) calculates the direction in which the parameter is improved, for this the Levenberg-Marquardt (LM) solver is used. The latter is embedded in the function

```matlab
[xf,S,msg] = ccl_math_solve_lm(varargin) 
```

where the inputs varargin are the objective function (fun), initial guess of model parameters (xc, i.e., \( \omega \)) and training options (options).

As an example for learning a parabolic constraint, a state-dependent constraint \( A(x) \) of the form \( A(x) = [-2ax, 1] \) is used in the library. For this, ccl.learn_alpha is implemented. Finally, nPPE and nPOE are used to evaluate the learning performance.

### 4.5. Learning Null Space Policy \( \pi \)

Looking at learning an unconstrained control policy \( \pi \). Data may be given as motions which are subject to some constraint and it may be of interest to separate the control policy of the system from the constraint such that this policy can be applied to new situations. To do so ccl.learnpi can be used.

This applies to the use case where \( \pi \) is consistent but \( A \) is varying. The CCL library implements the
Table 4.: To do: Sample code snippet for learning state independent and dependent constraint from matlab

```
Data = ccl_data_gen(settings);  
model = ccl_learn_nhat (Un); % for learning A
model = ccl_learn_alpha (Un, X, settings); % for learning A(x)
nPPE = ccl_error_ppe(U, model.P, Pi);  
nPOE = ccl_error_poe(U, model.P, Pi);
```

Table 5.: Sample code snippet for learning null-space control policy from Matlab

```
Data = ccl_data_gen(settings);  
model.c = ccl_math_gc (X, model.dim_basis);  
model.s = mean(mean(sqrt(ccl_math_distances(model.c, model.c))))^2;  
model.phi = @(x)ccl_basis_rbf ( x, model.c, model.s);  
model = ccl_learnp_pi (X, Un, model);  
fp = @(x) ccl_learnp_pred_linear ( model, x );  
Pi = fp (X);  
NUPE = ccl_error_nupe(Un, Unp);  
NCPE = ccl_error_ncpe (Un, Unp);
```

learning of these models through the functions

```matlab
model = ccl_learnp_pi (X,U,model)
```

where inputs X and U are observed states and actions. The outputs are the learnt model model.

In the CCL library, the risk function (17) is implemented as a parametric policy learning scheme. Where \( \tilde{\pi}(x) = W\beta(x) \), \( W \in \mathbb{R}^{Q \times M} \) is a matrix of weights and \( \beta(x) \in \mathbb{R}^M \) is a vector of fixed basis functions (e.g., linear features or Gaussian radial basis functions). A locally-weighted linear policy learning is also implemented in the library to improve the robustness of the policy learning (Details see [2]). A reinforcement learning scheme (with possibly deep network structure) can also be used to learn more complex policies [21–23].

A sample Matlab script is provided in Table 5 and explained as the following. For learning, 10 Radial Basis Functions are used and learned using the same K-means algorithm. ccl_learnp_pi is then implemented for training the model. Finally, NUPE and NCPE are calculated to evaluate the model’s performance. In the library, a locally weighted policy learning method is also implemented in both Matlab and C. For details please refer to the documentation.

Other examples such as a 2-link arm and wiping examples are also implemented which follow a similar procedure to the toy example but in a higher dimension to aid novice users wanting to take advantage of the CCL to learn the kinematic redundancy of a robot. Moreover, users can easily adapt these learning methods, where the provided demonstrations such as wiping and real data are set-up to use the 7 DoF KUKA LWR3 and Trakstar 3D electromagnetic tracking sensor, respectively, are reconfigured to suit their own systems and requirements.

4.6. Implementation of Evaluation Criteria

Once a constraint is learnt, results can be validated by using some function to assess the learnt model’s performance. In the context of evaluating the constraints, the CCL library provides implementations of several performance metrics which have been discussed in §5.2.

4.6.1. NPPE

In the CCL library, The NPPE is computed through the function

```matlab
[nPPE,vPPE,uPPE] = ccl_error_ppe (U_t,N_p,Pi)
```

where \( U_t \) is the true null space component \( W \), \( N_p \) is the learned projection matrix \( N \), and \( Pi \) is the
unconstrained control policy $\pi$. The outputs are the NPPE, variance and (non-normalised) mean-squared PPE, respectively.

4.6.2. NPOE

As mentioned before, the NPOE is used in the absence of ground truth $A_n$ and $\pi_n$. The function implemented for computing the Normalised Projected Observation Error is

$$[nPOE,vPOE,uPOE] = ccl_error_poe(U_t,N_p,Pi)$$

where the input and output conventions are similar to those of the functions computing the NPPE.

4.6.3. NPE

The NPE is implemented in the function

$$[umse,v,nmse] = ccl_error_npe(Un,Unp)$$

where $Un$ and $Unp$ are the true and predicted null space components. The return values are the NPE, variance and (non-normalised) mean squared projection error.

4.6.4. NUPE & NCPE

The NUPE and NCPE are implemented in the functions

$$[umse,v,nmse] = ccl_error_nupe(F,Fp)$$

and

$$[umse,v,nmse] = ccl_error_ncpe(F,Fp,P)$$

where input $F$ is the true unconstrained control policy commands, $F_p$ is the learned unconstrained control policy and $P$ is the projection matrix. The outputs are NUPE (respectively, NCPE), the sample variance, and the mean squared Unconstrained Policy Error (respectively, Constrained Policy Error).

5. Conclusion

This paper introduced the CCL library, an open-source collection of software tools for learning different components of constrained movements and behaviours. The implementations of the key functions have been explained throughout the paper, and interchangeable and expandable examples have been demonstrated using both simulated and real world data for their usage. For the first time, the library brings together a diverse collection of algorithms developed over the past ten years into a unified framework with a common interface for software developers in the robotics community. In the future work, Matlab and python wrappers will be released by taking advantage of the fast computation routine implemented in C.

References


