Precise Piecewise Affine Models from Input-Output Data

Rajeev Alur, Nimit Singhania
University of Pennsylvania

ABSTRACT

Formal design and analysis of embedded control software relies on mathematical models of dynamical systems, and such models can be hard to obtain. In this paper, we focus on automatic construction of piecewise affine models from input-output data. Given a set of examples, where each example consists of a d-dimensional real-valued input vector mapped to a real-valued output, we want to compute a set of affine functions that covers all the data points up to a specified degree of accuracy, along with a disjoint partitioning of the space of all inputs defined using a Boolean combination of affine inequalities with one region for each of the learnt functions. While traditional machine learning algorithms such as linear regression can be adapted to learn the set of affine functions, we develop new techniques based on automatic construction of interpolants to derive precise guards defining the desired partitioning corresponding to these functions. We report on a prototype tool, Mosaic, implemented in Matlab. We evaluate its performance using some synthetic data, and compare it against known techniques using datasets modeling electronic placement process in pick-and-place machines.

1. INTRODUCTION

Formal design and analysis of embedded control software relies on the construction of mathematical models representing the dynamics of system components [11]. For many real-world systems, constructing a model of the continuous-time dynamics from basic principles is very difficult, but it is possible to obtain input-output data from observed behaviors of the system. Thus algorithms to learn mathematical models of dynamical systems from input-output data can play a critical role in model-based design of embedded software. In this paper, we focus on learning *piecewise affine* models from data. Piecewise affine models are important since such models can approximate more complex behaviors including non-linear dynamics, and at the same time, are more amenable to formal analysis such as symbolic model checking [3].

ESWEEK'14 October 12 - 17 2014, New Delhi, India

ACM 978-1-4503-3052-7/14/10…$15.00.
http://dx.doi.org/10.1145/2656045.2656064

A piecewise affine model is a function from d-dimensional real-valued input vectors to real-valued outputs, and consists of a finite partitioning of the input domain into regions, with an affine function associated with each of these distinct regions. Each region of the partition has to be defined by a guard that is a boolean combination of affine inequalities. An example of a 2-dimensional piecewise affine model is the function given by the conditional expression **if** $(x_1 + 2x_2 \leq 3 \ \vee \ x_1 - 3x_2 \geq 7)$ **then** $2x_1 + x_2$ **else** $3x_2 + 5$. The problem of learning piecewise affine model from given input-output data is to find a set of affine functions that covers all examples and to compute guards that identify the partitioning corresponding to these affine functions.

This problem has been explored previously in research literature [8, 4, 20, 15, 21, 14, 12, 5]. Paoletti et al. present an extensive survey on these existing techniques in [19]. Some of these techniques [20, 15] pose this problem as a quadratic optimization problem. In [12, 5], the piecewise affine model is learnt iteratively by alternating between learning guards and learning affine functions. In [8], machine learning techniques for clustering (K-means) and linear separation (SVM) are used to learn the piecewise affine models. In [4], Bemporad et al. give a greedy heuristic to identify the affine functions and then use two-class or multi-class linear separation techniques to learn guards. All these techniques, however, assume that each region of the partition is *convex*, that is, defined via guards that are conjunctions of affine inequalities. In contrast, our solution is able to associate a *non-convex* region with each affine function. In other words, our method is able to detect when a single affine function can cover inputs in multiple convex sub-regions, and this potentially reduces the size of the representation of the learnt function.

To find a partitioning into non-convex regions, we propose a new technique to learn guards based on automatic construction of *interpolants* [1, 16]. An interpolant is a formula which precisely separates the solution space of two inconsistent predicates. While interpolant generation techniques in the literature operate on predicates, we need to adapt them to learn guards from input-output data points. It should be noted that many problems explored in the verification literature require automatic construction of predicates and functions. For instance, there is a large body of work on automatic construction of loop invariants in programs ([6, 10, 7]), there has been some recent work on synthesizing straight-line programs from logical constraints and examples [9, 2]. Our algorithm to learn piecewise affine models

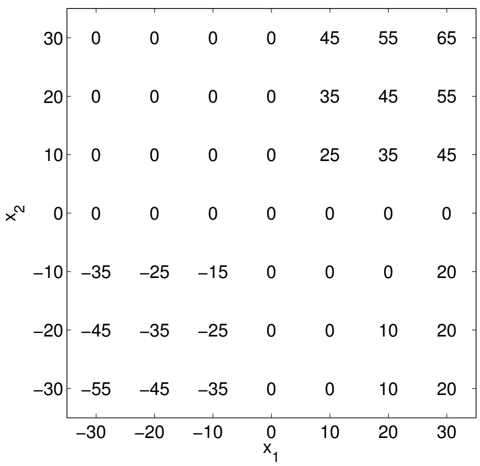


Figure 1: Example data-set. Each input point is labelled by the corresponding output value.

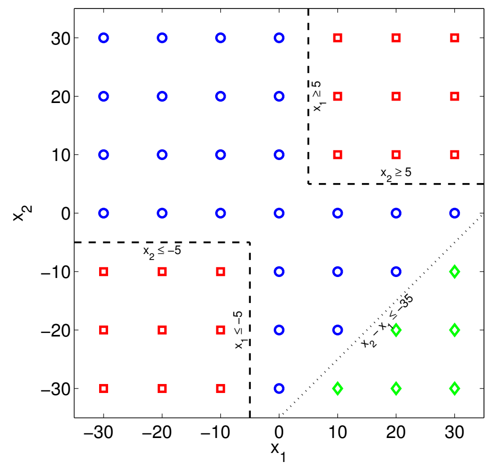


Figure 2: A piecewise affine model for data-set in Figure 1. □ points map to affine function (x_1+x_2+5), ◇ points to (x_1-10) and the ◯ points to 0.

can also be considered as a solution to a specific instance of such synthesis problems.

The first phase of the proposed algorithm learns a set of affine functions that *covers* all data points upto a specified degree of accuracy (given by an error bound δ). To do this, we first find an affine function that covers points in the neighborhood of a given point and then refine it further so as to cover as many points as possible. Next, we remove the points covered by this function and repeat the computation on remaining points until all points are covered. The second phase of the algorithm learns a guard for each affine function that characterizes the region containing points covered by this function. The desired guard is a boolean combination of affine inequalities that separates points covered by the function from remaining points in the data-set. For this purpose, we iteratively create positive and negative groups of points such that each pair of positive and negative group can be separated by a single affine inequality, and combine these inequalities using boolean connectives to get the desired guard. Finally, we use these affine functions and guards to construct the required piecewise affine model. A simpler model is more desirable as it is easier to analyze and by Occam's razor, more likely to generalize. Hence, we try to minimize the number of affine functions and the sizes of guards in the learnt model. However, these problems are computationally hard and thus, we only give a best effort solution for both of them.

We evaluate the performance of our algorithm using a prototype tool, Mosaic. It is implemented in Matlab. We use synthetic data to measure the quality of models learnt by Mosaic and real data from electronic placement process in pick-and-place machines to compare it against existing approaches. On synthetic data, we find that Mosaic performs well on low dimensional input-output data and also learns models of small sizes. For real data, we implement two alternate approaches. In the first approach, we replace our technique to learn guards with a machine learning based technique as used in most existing approaches [8, 4]. In the second one, we also replace our solution to learn affine functions with a clustering-based approach from [8]. We compare these on different data-sets from electronic placement process in pick-and-place machines. We find that Mosaic outperforms the other two on three of the data-sets with little overhead in the size of the model. This shows that non-convex regions can be found in real data which validates the need for our algorithm. This also provides some evidence that our approach can perform better than existing approaches and hence can be useful in practice.

The outline of the paper is as follows. In Section 2, we formalize the computational problem of learning piecewise affine models from input-output data. In Section 3, we describe the algorithms to learn affine functions and the guards defining the regions corresponding to these functions, along with analysis of correctness and complexity. In Section 4, we describe the implementation of our tool, Mosaic and evaluation of its performance. Finally in Section 5, we conclude with some discussion and directions for future work.

2. PROBLEM

Preliminaries. An *affine function* $l : \mathbb{R}^d \to \mathbb{R}$ is a function of the form $l(x_1, x_2, \ldots, x_d) = h_1x_1 + h_2x_2 + \cdots + h_dx_d + c$, where $h_1 \ldots h_d$, and c are real constants and $x_1, \ldots, x_d$ are real variables. Equivalently, it can be written as $l(\mathbf{x}) = \mathbf{h}.\mathbf{x} + c$, where $\mathbf{h}$ and $\mathbf{x}$ are d-dimensional vectors. Similarly, an *affine inequality*, $\phi : \mathbb{R}^d \to \{true, false\}$ is a predicate of the form $\phi(\mathbf{x}) \equiv p(\mathbf{x}) \leq 0$, where p is an affine function.

A *piecewise affine model* $f : \mathbb{R}^d \to \mathbb{R}$ is defined using the following expression.

$$\begin{aligned} f(\mathbf{x}) = &\ \textbf{if } \{\phi_1(\mathbf{x})\} \textbf{ then } l_1(\mathbf{x}) \\ &\ \textbf{else if } \{\phi_2(\mathbf{x})\} \textbf{ then } l_2(\mathbf{x}) \\ &\ \ldots \\ &\ \textbf{else if } \{\phi_{m-1}(\mathbf{x})\} \textbf{ then } l_{m-1}(\mathbf{x}) \\ &\ \textbf{else } l_m(\mathbf{x}) \end{aligned}$$

where $l_i(\mathbf{x})$ is an affine function i.e. $l_i(\mathbf{x}) = \mathbf{h}_i.\mathbf{x} + c_i$ and ϕ_i is a boolean combination of affine inequalities, i.e. $\phi_i(\mathbf{x}) = \bigvee_j \bigwedge_k p_i^{jk}(\mathbf{x}) \leq 0$.

The *size* of a piecewise affine model is defined as the total number of affine functions and affine inequalities used in the

Algorithm 1 genPiecewiseAffineModel

Input: $D = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_N, y_N)\}$ and an error bound δ
Output: Piecewise affine model f, s.t. $\|y_i - f(\mathbf{x}_i)\| \leq \delta$ for all $1 \leq i \leq N$

```
/* Compute a set of affine functions L = {l_1, l_2 ... l_m} that
   covers all points in D. */
X := D
L = {}
while X is non-empty do
    l := genAffineFunction(X)
    X := {(x_i, y_i) ∈ X | ||y_i − l(x_i)|| > δ}
    Add l to L
end while
/* Compute guards {φ_1, φ_2, ... φ_{m−1}} for regions defined
   by the affine functions in L */
for k = 1 to m − 1 do
    Let P_j = {x_i | ||y_i − l_j(x_i)|| ≤ δ, (x_i, y_i) ∈ D, l_j ∈ L}
    Select l_k ∈ L s.t. for all j, |P_k| ≤ |P_j|
    X_+ := P_k \ ⋃_{j≠k} P_j
    X_− := ⋃_{j≠k} P_j \ P_k
    φ_k := genGuard(X_+, X_−)
    L := L \ {l_k}
    D := D \ {(x_i, y_i) ∈ D | φ_k(x_i) = true}
end for
return f(x) = if {φ_1(x)} then l_1(x),
              else if {φ_2(x)} then l_2(x),
              ...
              else l_m(x)
```

Algorithm 2 genAffineFunction

Input: $X = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_N, y_N)\}$ and an error bound δ
Output: An affine function $l(\mathbf{x}) = \mathbf{h}.\mathbf{x} + c$ that covers some points in X

```
Randomly select (x_p, y_p) from X
P := {(x_p, y_p)}
l' := Affine function found by linear regression on P
while ||y_i − l'(x_i)|| ≤ δ, for all (x_i, y_i) ∈ P do
    l := l'
    v := argmin_j {|x_p − x_j| | (x_j, y_j) ∉ P }
    Add (x_v, y_v) to P
    l' := Affine function found by linear regression on P
end while
l' := l
P' := {(x_i, y_i) ∈ X | ||y_i − l'(x_i)|| ≤ δ}
repeat
    l := l'
    P := P'
    l' := Affine function found by linear regression on P
    P' := {(x_i, y_i) ∈ X | ||y_i − l'(x_i)|| ≤ δ}
until |P'| ≤ |P|
return l
```

expression for the model.

Problem. Given a set of input-output points, $D : \mathbb{R}^d \times \mathbb{R}$ and an error bound δ, the problem is to learn a *piecewise affine model* $f : \mathbb{R}^d \to \mathbb{R}$ such that,

$$\|f(\mathbf{x}_i) - y_i\| \leq \delta, \text{ for all } (\mathbf{x}_i, y_i) \in D$$

A function g *covers* a point $(\mathbf{x}, y)$ if $\|g(\mathbf{x}) - y\| \leq \delta$. Thus, we need to learn a model that covers all points in D.

Example. Consider a set of input-output points in $\mathbb{R}^2 \times \mathbb{R}$ as shown in Figure 1. Here, each input point is labelled by the corresponding output value. For example, a point $(10, 10)$ is mapped to the value 25 as can be seen in Figure 1. Let the error bound δ be 0.01. A possible piecewise affine model that covers all points in this data is as follows.

$f(x_1, x_2) =$
if $x_2 - x_1 \leq -35$ **then**
  $x_1 - 10$
else if $(x_1 \leq -5 \wedge x_2 \leq -5) \vee (5 \leq x_1 \wedge 5 \leq x_2)$ **then**
  $x_1 + x_2 + 5$
else
  0

This model is also shown in Figure 2. It has 5 inequalities and 3 affine functions and thus, a total size 8. This is the smallest possible model for the given data.

3. SOLUTION

The problem, as described in Section 2, is to compute a piecewise affine model f that covers all points in the dataset D. We divide this problem into two subproblems. The first subproblem is to find a set of affine functions L such that every point in D is covered by at least one affine function in L. These form the required affine functions l_i in f. The second subproblem is to learn a guard predicate for each affine function $l \in L$ such that it separates the points covered by l from the remaining points in D. These guard predicates form the guards ϕ_i in f. Finally, we use the affine functions and guard predicates to construct the required piecewise affine model f which covers all points in D. We give strategies to solve the subproblems in Section 3.1 and Section 3.2 respectively.

3.1 Learning Affine Functions

We describe here, our strategy to learn a set of affine functions L that covers all points in D. An affine function l *covers* a point $(\mathbf{x}, y)$ if $\|l(\mathbf{x}) - y\| \leq \delta$, where δ is the error bound. A set of affine functions L *covers* $(\mathbf{x}, y)$ if there exists at least one affine function $l \in L$ such that l covers $(\mathbf{x}, y)$. We would want to learn the smallest set L which covers all points in D so as to minimize the size of the model. However, this problem is computationally hard. We state this formally in Theorem 1. Meggido et al. prove this for $d = 2$ in [18]. This can be extended further to all constants $d > 2$, by showing a polytime reduction from the problem in d dimensions to that in $d + 1$ dimensions.

Theorem 1. *Given N input-output points in d dimensions i.e. $\mathbb{R}^d \times \mathbb{R}$, the problem of finding r affine functions such that they cover all N points is NP-Hard for all constants $d \geq 2$.*

Hence, we give a heuristic that learns a small set L in practice. We learn this set iteratively. We first find an affine function l that covers some points in D. Then, we remove points covered by l and repeat this process on the remaining points until all points in D are covered. We also describe this in the first part of Algorithm 1.

We now explain the algorithm to learn an affine function that covers some points in a set X (Algorithm 2). First, we select a random point $p : (\mathbf{x}_p, y_p)$ in X and learn an affine function that covers neighboring points of p. We start with a set P containing only p and repeatedly add nearest neighbors of p until all of them can not be covered by the learnt affine function. We use linear regression to learn the affine function that covers points in P. *Linear regression* is a technique that computes the affine function which minimizes the error on the given set of points. This is a classic technique in machine learning and we use it as a black box to learn the optimum affine function. Let l be the affine function that covers the maximum number of neighboring points of p. Now, we find all points in X which are covered by l. We apply *linear regression* on these points to learn a new affine function l' and compute a new set of points covered by l'. We repeat this step until the number of covered points stops increasing and then return the affine function that covers the maximum number of points. The assumption here is that an affine function which covers a point p, must also cover points in the neighborhood of p. Hence, we first learn an affine function that covers the neighborhood of a point in X and then use it as seed to learn the final affine function.

We illustrate this further with the example described in Section 2. Suppose, we select $((-20, -10), -25)$ as our random point p. We use its neighboring point $((-10, -10), -15)$ to learn an affine function $l_1(\mathbf{x}) = x_1 - 5$. Both points lie within the given error bound 0.01 from l_1 and hence, are covered by l_1. Next, we add another neighboring point $((-20, -20), -35)$ and learn the function $l_2(\mathbf{x}) = x_1 + x_2 + 5$. Again, all 3 points are covered by l_2. Next, we add the point $((-20, 0), 0)$ and learn the function $l_3(\mathbf{x}) = 0.5x_1 + 1.75x_2 + 7.5$. Now, none of the points are covered by l_3. Hence we stop and l_2 is the function that covers most points in the neighborhood of p. Next, we find all points in D which are covered by l_2. All $\square$ points, as shown in Figure 2, lie within the error bound 0.01 from l_2. Thus, we use these points to learn a new affine function l'_2 which is same as l_2. Since, the number of covered points remains same as earlier, we stop and return l'_2 as the required affine function. Now, we remove all $\square$ points and repeat the search for affine functions on the remaining points.

3.2 Learning Guard Predicates

After we have learnt a set of affine functions L, the next task is to learn a guard predicate ϕ_j for each affine function $l_j \in L$. ϕ_j identifies the region where f is defined by the affine function l_j. Therefore, given an input point $\mathbf{x} \in \mathbb{R}^d$, when $\phi_j(\mathbf{x})$ is *true*, $f(\mathbf{x}) = l_j(\mathbf{x})$ and so, ϕ_j must be *true* on points in D which are covered by l_j. Also, it must be *false* on the points $\mathbf{x}_i \in D$ which are not covered by l_j so that $f(\mathbf{x}_i) \neq l_j(\mathbf{x}_i)$. Thus, given an affine function l_j, the problem here is to find a guard predicate ϕ_j such that ϕ_j is *true* on points that are covered by l_j and *false* on the remaining points in D.

3.2.1 Overall Strategy

We explain here the overall strategy to learn guards for affine functions in L. This is also described in the second part of Algorithm 1. First, we select the affine function l_1 that covers the *smallest* set of points in D. This is a heuristic to eliminate spurious affine functions, which cover small sets of points, early in the algorithm and learn simpler guards for the relevant affine functions. Next, we learn a guard predicate ϕ_1 that separates points covered by l_1 from the remaining points in D. Then, we remove l_1 from L

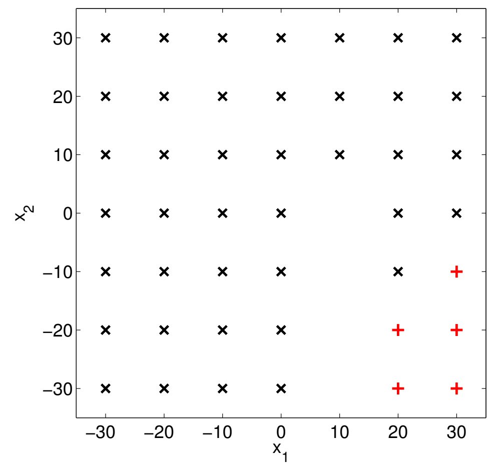


Figure 3: Positive points are marked by '+'. Negative points are marked by '×'.

and points in D for which ϕ_1 is *true*. This is because, the guards are checked in a sequential order in the piecewise affine model f and ϕ_1 is the first guard to be checked. Hence, while learning the subsequent guards, we can ignore points on which ϕ_1 is *true*. Now, we repeat the above process and learn guards for the remaining affine functions in L.

To learn a guard predicate ϕ_j that separates points covered by l_j from the remaining points, we label points in D that are covered only by l_j and no other affine function in L as *positive* and the points that are not covered by l_j as *negative*. Points that are covered by l_j and also some other affine function in L are ignored as they can be labelled either way. Now, ϕ_j is a predicate such that it is *true* on the positive points and *false* on the negative points. This is a standard problem of learning a binary classifier and many techniques in machine learning can be used for this. These techniques work well when the points can be separated by an affine inequality or a conjunction of affine inequalities. However, such a classifier does not always suffice and we may need a boolean combination of inequalities, for example, when positive points occur in disconnected groups in between negative points (Figure 4). Hence, we develop a new technique that learns precise classifiers, based on the work on learning interpolants by Albarghouthi et al. in [1]. We present this in Section 3.2.2.

We explain the overall strategy further using the example in Section 2. Suppose that the set of affine functions learnt for this data is $L = \{l_1 = x_1 - 10, l_2 = x_1 + x_2 + 5, l_3 = 0\}$. l_1 covers the least number of points and hence, we select it first and learn a guard ϕ_1 that separates points covered by l_1 from the remaining points. As we can see in Figure 3, we label points that are covered only by l_1 as positive and those not covered by l_1 as negative. Note that, the points $(10, 0), (10, -10), (10, -20), (10, -30)$ are covered by both l_1 and l_3. Hence, these points are ignored. Suppose, we learn ϕ_1 as $x_2 - x_1 \leq -35$. We remove l_1 from L and points in D where ϕ_1 is *true*. Next, we repeat the above process and select l_2 which covers the least number of remaining points in D. We learn a predicate ϕ_2 that separates points covered by l_2 from those covered by l_3. The corresponding positive

Algorithm 3 genGuard

Input: A set of points X_+ and X_-, s.t. $X_+ \cap X_- = \{\}$
Output: A guard ϕ, s.t. for all $\mathbf{x}_i \in X_+,\ \phi(\mathbf{x}_i) = \mathit{true}$ and for all $\mathbf{x}_i \in X_-,\ \phi(\mathbf{x}_i) = \mathit{false}$

```
Randomly select x+ ∈ X+ and x− ∈ X−
S+ := {{x+}}
S− := {{x−}}
while true do
    φ := genPred(S+, S−)                                   ▷ [E]
    Y := {xi ∈ X+ | ¬φ(xi)} ∪ {xi ∈ X− | φ(xi)}
    if Y is empty then
        return φ
    Randomly select xce in Y                               ▷ [CE]
    case xce ∈ X+
        for all g− ∈ S− do
            if genPred({{xce}}, {g−}) = NULL then
                /*xce conflicts with group g− */           ▷ [S]
                /*g− needs to be split */
                Find an affine function l such that
                    l(xce) = 0 and l(w) ≠ 0, for all w ∈ g−.
                g> := {w ∈ g− | l(w) > 0}
                g< := {w ∈ g− | l(w) < 0}
                Remove g− from S−
                Add g> and g< to S−.
        end for
        for all g+ ∈ S+ do
            /* Try merging xce in the group g+ */          ▷ [M]
            if genPred({g+ ∪ {xce}}, S−) ≠ NULL then
                g+ := g+ ∪ {xce}
                break
        end for
        if xce is not merged in any group g+ then
            /* Create a new group {xce} in S+ */           ▷ [N]
            S+ := S+ ∪ {{xce}}
    case xce ∈ X−
        ...
end while
```

Algorithm 4 genPred

Input: A set of positive and negative groups, S_+ and S_-.
Output: A boolean combination of affine inequalities ψ such that for all $\mathbf{x}$ in positive groups, $\psi(\mathbf{x}) = \mathit{true}$ and for all $\mathbf{x}$ in negative groups, $\psi(\mathbf{x}) = \mathit{false}$.
Returns $NULL$ if some positive and negative group are not separable.

```
ψ := false
for all g+ ∈ S+ do
    t := true
    for all g− ∈ S− do
        Find an affine function l such that
            for all x+ ∈ g+, l(x+) ≤ 0 and
            for all x− ∈ g−, l(x−) > 0
        if no such l exists then
            return NULL
        t := t ∧ (l(x) ≤ 0)
    end for
    ψ := ψ ∨ t
end for
```

and negative points are shown in Figure 4. Note that here, the positive points can not be separated from the negative points using only conjunction of inequalities and thus ϕ_2 can not be learnt using traditional machine learning techniques. Finally, we are left with points only from l_3 and we need not learn a predicate for this function.

3.2.2 Learning Precise Classifiers

Here we describe our approach to learn a guard predicate ϕ that separates a set of positive points, X_+ from a set of negative points X_- i.e. $\phi(\mathbf{x}) = \mathit{true}$ for all x in X_+ and $\phi(\mathbf{x}) = \mathit{false}$ for all x in X_-. It may not be possible to separate all points in X_+ and X_- by a single affine inequality. Therefore, in this algorithm, we create some positive and negative *groups* of points, such that each positive group can be separated from every negative group by an affine inequality and then, combine these inequalities using boolean connectives to learn a predicate ψ which separates all positive groups from the negative groups. If ψ also separates all points in X_+ from those in X_-, then we have the required guard ϕ. Otherwise, we update our groups to learn a new predicate ψ. The procedure to compute ψ from sets of positive and negative groups is described in Algorithm 4. We create a disjunct for each positive group by conjoining the affine inequalities which separate the positive group from the negative groups and then disjoin these disjuncts to get the required predicate. To compute an affine inequality that separates a positive group from a negative group, we use a *linear constraint solver* that takes in a set of constraints and returns a feasible solution, if it exists. Note that, Algorithm 4 returns $NULL$ if a positive group can not be separated from a negative group by an affine inequality.

Now, we describe our algorithm to create positive and negative groups and learn the required guard predicate ϕ. The pseudocode is given in Algorithm 3. Let the set of positive groups be S_+ and that of negative groups be S_-. We start with a single positive point and negative point as positive and negative group respectively. We learn a predicate ϕ from these groups using Algorithm 4 (step **E** in Algorithm 3) and then compute the set of misclassified points or counterexamples, Y. If Y is empty and thus, all points in X_- and X_+ are classified correctly by ϕ, we return ϕ. Otherwise, we randomly pick a counterexample point $\mathbf{x}_{ce}$ in Y (step **CE**). We update our groups such that $\mathbf{x}_{ce}$ is classified correctly in future. Let us consider the case that $\mathbf{x}_{ce}$ is a positive counterexample i.e. a positive point on which ϕ is *false*. We check if $\mathbf{x}_{ce}$ can be added to some positive group, $g_+ \in S_+$ (step **M**). Note that, g_+ must remain separable from all negative groups by affine inequalities even after adding $\mathbf{x}_{ce}$. If this is the case, we add $\mathbf{x}_{ce}$ to g_+. Otherwise, we create a new positive group that consists only of $\mathbf{x}_{ce}$ and add it to S_+ (step **N**).

We might have a situation where $\mathbf{x}_{ce}$ can not be separated from a negative group g_- by an affine inequality even when it is not added to any of the positive groups. This can happen when $\mathbf{x}_{ce}$ lies in between the points in g_-. In this case, we split g_- into new groups $g_>$ and $g_<$ such that they can be separated from $\mathbf{x}_{ce}$ by affine inequalities (step **S**). We do this by finding an affine function l such that $l(\mathbf{x}_{ce}) = 0$ and $l(\mathbf{x}) \neq 0$ for all $\mathbf{x} \in g_-$. We again use the linear constraint solver to find l. Now, group $g_>$ is the set of points in g_- where $l(\mathbf{x}) > 0$. Group $g_<$ is similarly defined. We add these new groups to S_- and remove g_- from S_-.

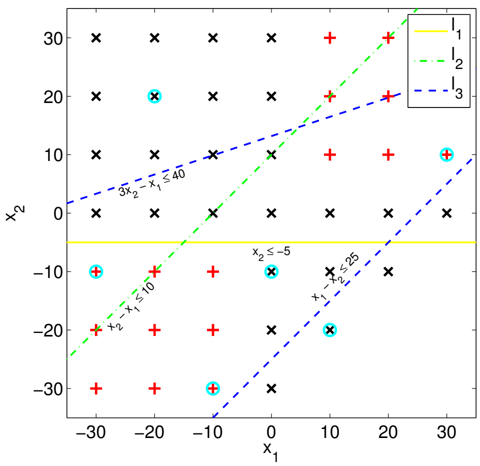


Figure 4: Positive points are marked by '+'. Negative points are marked by '×'.

We follow a similar procedure when $\mathbf{x}_{ce}$ is a negative counterexample. After the groups are updated, all points in groups along with the counterexample $\mathbf{x}_{ce}$ are classified correctly by the new predicate ϕ (step **E**). We repeat this process until all points in X_+ and X_- are classified correctly.

We explain this further using the example in Section 2. Let us try to separate positive points from negative points in Figure 4. Suppose we start with groups $g_+^1 = \{(-10, -30)\}$ and $g_-^1 = \{(-20, 20)\}$ and learn the predicate separating positive and negative groups as $\phi_1 : x_2 \leq -5$. ϕ_1 is *true* in the region below line l_1 in Figure 4. Let us pick a positive counterexample $ce_1 = (30, 10)$. ce_1 can be merged with g_+^1 to produce $g_+^1 = \{(-10, -30), (30, 10)\}$. Now, the predicate separating groups is $\phi_2 : x_2 - x_1 \leq 10$ which is *true* in the region below l_2. Next, we pick another positive counterexample $ce_2 = (-30, -10)$ and add it to g_+^1 in a similar way.

Now suppose we pick a negative counterexample $ce_3 = (10, -20)$. Clearly, it can not be merged with g_-^1 as $(g_-^1 \cup ce_3)$ can not be separated from g_+^1 by an affine inequality. Hence, we create a new negative group $g_-^2 = \{(10, -20)\}$. Now, the predicate separating groups becomes $\phi_3 : (3x_2 - x_1 \leq 40 \wedge x_1 - x_2 \leq 25)$. ϕ_3 is *true* in the region between two dashed lines marked as l_3. Next, suppose we pick another negative counterexample $ce_4 = (0, -10)$. ce_4 lies in between the elements of g_+^1 and so, it can not be separated from g_+^1 by any affine inequality. Thus, we split $g_+^1 : \{(-10, -30), (30, 10), (-30, -10)\}$ into 2 new groups $g_+^2 = \{(-30, -10), (-10, -30)\}$ and $g_+^3 = \{(30, 10)\}$. g_+^2 and g_+^3 contain points with $x_1 < 0$ and $x_1 > 0$ respectively. ce_4 can be separated from the positive groups g_+^2 and g_+^3 by affine inequalities. Further, we can add ce_4 to g_-^2 and learn a new predicate separating groups. We repeat this process until no new counterexamples can be found.

Note that, this algorithm may not produce a predicate with smallest number of affine inequalities. However, doing so is computationally hard as stated in Theorem 2. Meggido et al. prove this in [17].

Theorem 2 ([17]). *The problem of checking if 2 sets of points can be separated by a predicate with k affine inequalities is NP-Hard.*

3.3 Correctness and Runtime Analysis

We analyze here the correctness and the running time of our algorithm. First we show the correctness of Algorithm 3. To show this, we prove that groups in S_+ and S_- can always be separated by an affine inequality.

Lemma 3. *In Algorithm 3, for all pairs $(g_+, g_-), g_+ \in S_+, g_- \in S_-$, g_+ and g_- can be separated by an affine inequality.*

Proof. Initially, groups in S_+ and S_- are separable by an affine inequality. We show that the theorem holds also at steps **S**, **M** and **N** where S_+ and S_- are modified. First, at step **S**, both $g_>$ and $g_<$ are subsets of g_- and thus, they can be separated from every $g_+ \in S_+$ by affine inequalities. Hence, the theorem holds at step **S**. Also note that after the split, $\mathbf{x}_{ce}$ can be separated from both $g_>$ and $g_<$ by affine inequalities. Thus, after all conflicting negative groups are split, $\mathbf{x}_{ce}$ can be separated from all groups in S_- by affine inequalities. At step **M**, $\mathbf{x}_{ce}$ is added to g_+ only if it remains separable from all groups in S_- by affine inequalities. Therefore, S_+ and S_- still remain valid. Finally, at step **N**, we know that $\mathbf{x}_{ce}$ is separable from all groups in S_- by affine inequalities and so, $\{\mathbf{x}_{ce}\}$ can be added to S_+ while maintaining validity. □

Correctness of Algorithm 2 can be shown trivially. Correctness of Algorithm 1 follows from the correctness of Algorithm 2 and Algorithm 3. We state it as a lemma here.

Lemma 4. *In Algorithm 1, piecewise affine model f covers all points in D i.e. $\|y_i - f(\mathbf{x}_i)\| \leq \delta, \ \forall(\mathbf{x}_i, y_i) \in D$.*

Next, we analyze the running time of Algorithm 3, which is the most expensive component of the complete algorithm. We quantify this by the number of times we search for an affine function during the algorithm. This is because, searching an affine function that satisfies the given constraints is costly and dominates the running time of Algorithm 3. This search happens at step **S** in Algorithm 3 and within the loops in Algorithm 4 and is done by making queries to a linear constraint solver. We now state the following lemmas.

Lemma 5. *Algorithm 4 makes $O(|S_+| \times |S_-|)$ queries to a linear constraint solver.*

Lemma 6. *Algorithm 3 makes $O(N^3)$ queries to a linear constraint solver, where N is the number of points in D.*

Proof. First, the number of iterations of the outer while loop in Algorithm 3 is $O(N)$. This is because, in each iteration, the groups are updated such that the counterexample $\mathbf{x}_{ce}$ is classified correctly in future and can not be a counterexample again. Thus each iteration eliminates at least one point from being a counterexample which implies that there are at most $O(N)$ iterations. Now, $|S_+| < N$ and $|S_-| < N$. Thus, at step **E**, the call to Algorithm 4 makes $O(N^2)$ queries to the linear constraint solver. Step **S** runs at most $O(N)$ times and makes a constant number of queries to the solver each time. Similarly, step **M** runs at most $O(N)$ times in an iteration and each call to Algorithm 4 makes $O(N)$ queries to the solver. Therefore, there are $O(N^2)$ queries to the linear constraint solver in each iteration of the outer while loop and thus, the total number of queries is $O(N^3)$. □

4. EVALUATION

In this section, we evaluate the performance of our algorithm using a prototype tool, Mosaic. In Section 4.1, we describe its implementation and some synthetic and real data-sets used for its evaluation. In Section 4.2, we use synthetic data to measure the quality of the piecewise affine models learnt by the tool. In Section 4.3, we use real data from electronic placement process in pick and place machines to compare its performance against existing techniques.

4.1 Experimental Setup

We have implemented our tool, Mosaic, in Matlab. We implement the linear constraint solver in Mosaic via the linear program solver within Matlab and some simple heuristics. To compute an affine function that separates a positive group from a negative group in Algorithm 4, we encode the constraints as a linear program and use the linear program solver to solve them. To compute the affine function l at step **S** in Algorithm 3, we generate an affine function with random coefficients and update it such that $l(\mathbf{x}_{ce}) = 0$. Then, we check if there is a point $\mathbf{x}$ in g_-, such that $l(\mathbf{x}) = 0$. We repeat the above steps until no point $\mathbf{x}$ in g_- has $l(\mathbf{x}) = 0$, which gives the required affine function l. While some off-the-shelf linear constraint solver could also be used, we found this to work well in practice. We implement linear regression using an inbuilt function of Matlab. We use a machine with 2.6 GHz i5 processor and 8GB RAM to conduct the experiments and evaluate the performance of our tool.

We have created some synthetic functions and use data sampled from these to learn piecewise affine models, so as to compare the performance and the size of the models learnt by Mosaic against the true models. We have manually constructed four piecewise affine functions, f_1, f_2, f_3 and g and three smooth non-linear functions, p, q, r. f_1, f_2, f_3 are functions in $\mathbb{R}^5 \to \mathbb{R}$, while g is a function in $\mathbb{R}^{10} \to \mathbb{R}$. The sizes of these functions are $10, 10, 11$ and 19 respectively. Functions p, q and r have input dimensions $3, 4$ and 3 respectively. We sample points from these functions by selecting a value for each input variable randomly in range $[-1000, 1000]$.

We use data from an experimental study conducted by Juloski et al. in [13] to compare our algorithm against existing approaches. This study tries to model the electronic component placement process in pick and place machines. A pick and place machine has a mounting head that carries an electronic component. The head is pushed down by the machine until the electronic component is inserted inside the circuit board placed below. Then, the head is retracted and another electronic component is picked up to be pushed inside the board. The data consists of the input voltage of the motor that drives the head down and the position of the head, both of which are collected at regular intervals of time. We want to learn a piecewise affine model that represents the head position at any time instant k as a function of head positions at time $k-1, k-2, \ldots, k-a$ and input voltages at time $k, k-1, k-2, \ldots k-b$, also known as PWARX model. We set a to 2 and b to 1 for our experiments. Data-sets are collected in 4 different settings of the machine for 15 seconds each. We sample each data-set at 150Hz which gives us 2250 data points for each setting.

The synthetic functions, the datasets from the experimental study and the code for Mosaic are available at `http://seas.upenn.edu/~nimits/mosaic`.

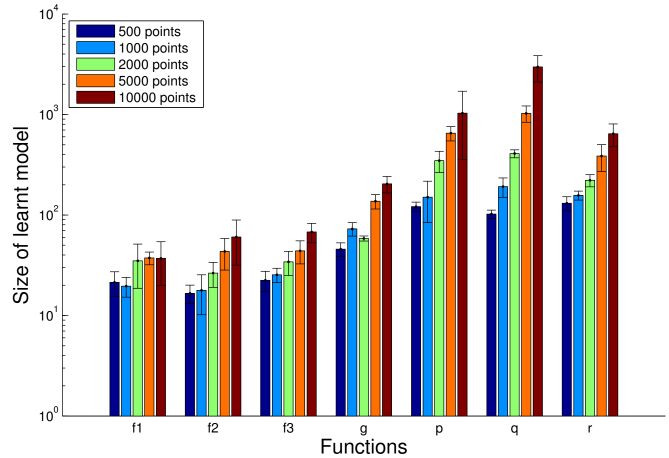


Figure 5: Size of the learnt model

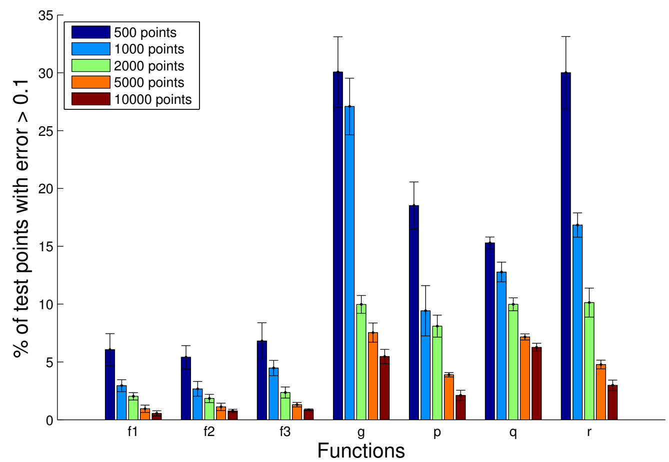


Figure 6: % of test points with error > 0.1

4.2 Quality of the Learnt Models

We use synthetic data, as described in Section 4.1, to measure the quality of the model learnt by Mosaic. To do this, we first sample some training points from the functions. The number of points vary from 500 to 10000. We also add a Gaussian noise with zero mean and standard deviation 0.001 to these points. Next, we use our tool to learn piecewise affine models from these data points with an error bound 0.1. Finally, we evaluate the learnt piecewise affine models on 20000 test points that are sampled separately from the original functions.

We measure the quality of the learnt models using two parameters, the size of the learnt model and its performance on test data. Figure 5 shows the size of learnt models for varying number of training points. Figure 6 shows the percentage of test points for which the error is greater than the error bound 0.1. We found that our algorithm is able to reconstruct the affine functions in f_1, f_2, f_3 and g quite accurately. Hence, we wanted to measure the fraction of test points which are assigned wrong regions by the learnt models. A test point assigned a wrong region is likely to have an error greater than the given error bound 0.1 and thus, we use percentage of test points with error greater than > 0.1 as the measure of performance on test set. While this is not necessarily true for the smooth functions p, q, r, this measure is proportional to the root mean square error and hence, we use it as a measure of performance for all functions.

We can observe that as the number of training points increases, the size of the learnt model increases and the performance on the test set improves. The learnt models seem to perform very well on the test data for f_1, f_2 and f_3 with less than 5% error for models learnt with 1000 points or more. The error on g, p, q, and r is much larger in comparison and thus here, the learnt models do not seem to generalize to the test data. For f_1, f_2 and f_3, the size of the learnt model is fairly close to the actual size even when we have 10000 training points. However, for g, the size of the learnt model grows significantly with increase in the number of training points. As we mentioned in the previous paragraph, the affine functions are reconstructed quite accurately in the learnt models. Hence, the increase in size is mainly because of increase in the size of the guards. We investigated this further and found that while learning the guard predicates, large number of positive and negative groups were created when only few groups would suffice. This was probably because many irrelevant groups were formed containing points from disconnected regions. Such groups could be formed if there were no points of opposite label that lie in between to restrict this formation. For example, in Figure 4, positive points $(-10, -10)$ and $(10, 20)$ could form a group even though they lie in disconnected positive regions, because there are no negative points in between to prevent this. This problem becomes more severe when the input points are in 10 dimensions as in case of g. We will try to address this problem in future. For functions p, q, and r, sizes of the learnt models are quite large, but there is no baseline to compare them. Finally, these functions will be available online and can be used as benchmarks for future work.

4.3 Comparison with Existing Techniques

Now we use the real data from the electronic placement process in pick and place machines to compare our algorithm against existing techniques.

The first question that we want to answer via experiments is whether it is useful to learn guards using the technique presented in Section 3.2.2 and how does it compare with traditional learning techniques like Support Vector Machines (SVMs). SVM is a machine learning technique that learns an affine inequality such that the separation between given positive and negative points is maximized. We can observe in Figure 4, that a single affine inequality or even a conjunction of affine inequalities can not separate positive and negative points and so, SVMs would not perform well in this example. However, the question is whether such scenarios are found often in real data. To answer this, we have implemented an alternate approach that uses SVM for learning guards and compare it with Mosaic on the pick and place machine data. In this approach, we use SVM to learn a single affine inequality that separates positive points from negative points. While in Mosaic, we learn a single guard predicate that separates points covered by an affine function l_i from points covered by remaining affine functions, here we learn multiple predicates where each predicate is an affine inequality separating points covered by l_i from points covered by one of the remaining affine functions, and return a conjunction of these predicates as the required guard. Let us call this implementation Linear.

The next question that we want to answer is whether Linear would perform better if we split points covered by an affine function into clusters and then learn guards for these clusters using SVMs. For example, in Figure 4, we could split positive points into 2 clusters, each with 9 points forming a square and then, each cluster could be separated from negative points by a conjunction of inequalities. In fact, Ferrari-Trecate et al. give such an algorithm in [8]. We have implemented a version of this algorithm and compare it with Mosaic and Linear. In this algorithm, first, for each training point p, an affine function l_p that covers its c nearest neighbors and the mean point m_p of these neighbors is computed. Then, the points are assigned a weight w_p that measures how well l_p fits the c neighboring points of p with lower weight for poorer fit. We faced some numerical issues like infinite or not-a-number values while computing these weights and hence weighed all points equally. Next, the training points are segregated into K clusters using a weighted version of K-means, such that two points lying in the same cluster have similar mean point m_p and are covered by similar affine functions l_p. This is done so that the points in a cluster are covered by the same affine function and also are close to each other, whereby each cluster could be separated from other clusters by affine inequalities. Next, an affine function l_k is assigned to each cluster C_k such that it minimizes error on points in C_k and then, a predicate that separates C_k from other clusters is learnt using the approach described in Linear. Let us call this implementation Cluster.

We compare these implementations via experiments on the pick and place machine data as described in Section 4.1. We use first 1500 points in each data-set as training points and remaining 750 points as test points. Note that each point is given by $((y_{k-1}, y_{k-2}, v_k, v_{k-1}), y_k)$ where y_k is the position of the head at time k and v_k is the input voltage of the motor at time k. Figure 7 shows the results for the experiments on 4 data-sets. For Mosaic and Linear, we learn models by varying the error bound δ. We repeat this 15 times for each value of δ and report train (dashed line) and test (solid line) root mean square error and the size of the learnt model in Figure 7. For Cluster, we learn models by varying the number of neighboring points c from 10 to 70 and the number of clusters K from 2 to 12. We report train and test root mean square error for different values of c and K by the black $\diamond$ and $\circ$ points respectively in Figure 7. Note that, the size of the learnt model for Cluster is given by about $K(K+1)/2$, where K is the number of clusters and thus varies from 3 to 78.

As we can observe in Figure 7, the train root mean square error of Mosaic is smaller than both Linear and Cluster and reduces as the error bound is decreased. Further, Mosaic has generally lower test error than Linear and Cluster for data-sets 1, 2 and 3 and similar error for data-set 4. This gives some evidence that non-convex regions (regions that can not be identified by conjunction of affine inequalities) for affine functions can be found in real data and thus, our technique to learn guards can perform better than machine learning techniques like SVMs. Further, we can observe that, as δ is reduced, train error reduces for Mosaic but, test error increases generally. Also, the size of the learnt model increases rapidly as we reduce δ. This is a standard observation in machine learning and is known as overfitting i.e. when δ is reduced significantly, the learnt model overfits the training data and thus, does not generalize to the test data. Further, CLUSTER performs worse than MOSAIC on data-set 1 and 2 and 3. This is probably because in CLUSTER, while the regions for the affine functions are better separated by SVM, the affine functions do not fit the data very well. To conclude, we have some evidence that non-convex regions for affine functions can be found in real data and therefore, our algorithm to learn piecewise affine models can be useful in practice.

5. CONCLUSION AND FUTURE WORK

In this paper, we have presented a new technique to learn piecewise affine models, with a novel application of interpolant generation to learn guards which can identify non-convex regions for affine functions. Further, we have evaluated this algorithm on some synthetic and real data and observe that it performs well with a little overhead in the size of the learnt models. In future, we would like to solve the problem discussed in Section 4.2 where many irrelevant groups are formed while learning guard predicates. Further, currently while learning guard predicates, we pick a *random* counterexample and use it to update the positive and negative groups. We would like to explore other strategies to pick the counterexample and see how they affect the performance of this algorithm. We would also like to explore if our technique to learn guard predicates can also be used as a machine learning technique to learn classifiers. Lastly, we would like to explore other applications for our algorithm to learn piecewise affine models.

6. ACKNOWLEDGEMENTS

This research was partially supported by NSF Expeditions in Computing award CCF 1138996. We thank Aleksandar Lj. Juloski and Giancarlo Ferrari-Trecate for providing the datasets used for experiments in Section 4.3. We also thank Sepehr Assadi for help with the proof to Theorem 1 and the unknown reviewers for their valuable feedback.

7. REFERENCES

[1] A. Albarghouthi and K. L. McMillan. Beautiful interpolants. In *Proceedings of the 25th International Conference on Computer Aided Verification*, CAV'13, pages 313–329, 2013.

[2] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD*, pages 1–17, 2013.

[3] R. Alur, T. Henzinger, G. Lafferriere, and G. Pappas. Discrete abstractions of hybrid systems. *Proceedings of the IEEE*, 88(7):971–984, 2000.

[4] A. Bemporad, A. Garulli, S. Paoletti, and A. Vicino. A bounded-error approach to piecewise affine system identification. *Automatic Control, IEEE Transactions on*, 50(10):1567–1580, Oct 2005.

[5] L. Breiman. Hinging hyperplanes for regression, classification, and function approximation. *Information Theory, IEEE Transactions on*, 39(3):999–1013, May 1993.

[6] M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In W. Hunt and F. Somenzi, editors, *Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 420–432. Springer Berlin Heidelberg, 2003.

[7] I. Dillig, T. Dillig, B. Li, and K. McMillan. Inductive invariant generation via abductive inference. *SIGPLAN Not.*, 48(10):443–456, Oct. 2013.

[8] G. Ferrari-Trecate, M. Muselli, D. Liberati, and M. Morari. A clustering technique for the identification of piecewise affine systems. *Automatica*, 39(2):205 – 217, 2003.

[9] S. Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 13–24, 2010.

[10] A. Gupta and A. Rybalchenko. Invgen: An efficient invariant generator. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 634–640, Berlin, Heidelberg, 2009. Springer-Verlag.

[11] T. A. Henzinger and J. Sifakis. The embedded systems design challenge. In *FM 2006: 14th International Symposium on Formal Methods*, LNCS 4085, pages 1–15, 2006.

[12] M. I. Jordan and R. A. Jacobs. Hierarchical mixtures of experts and the em algorithm. *Neural Comput.*, 6(2):181–214, Mar. 1994.

[13] A. Juloski, W. Heemels, and G. Ferrari-Trecate. Data-based hybrid modelling of the component placement process in pick-and-place machines. *Control Engineering Practice*, 12(10):1241 – 1252, 2004. Analysis and Design of Hybrid Systems.

[14] A. Juloski, S. Weiland, and W. P. M. H. Heemels. A bayesian approach to identification of hybrid systems. *Automatic Control, IEEE Transactions on*, 50(10):1520–1533, Oct 2005.

[15] A. Magnani and S. Boyd. Convex piecewise-linear fitting. *Optimization and Engineering*, 10(1):1–17, 2009.

[16] K. L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, 345(1):101–121, Nov. 2005.

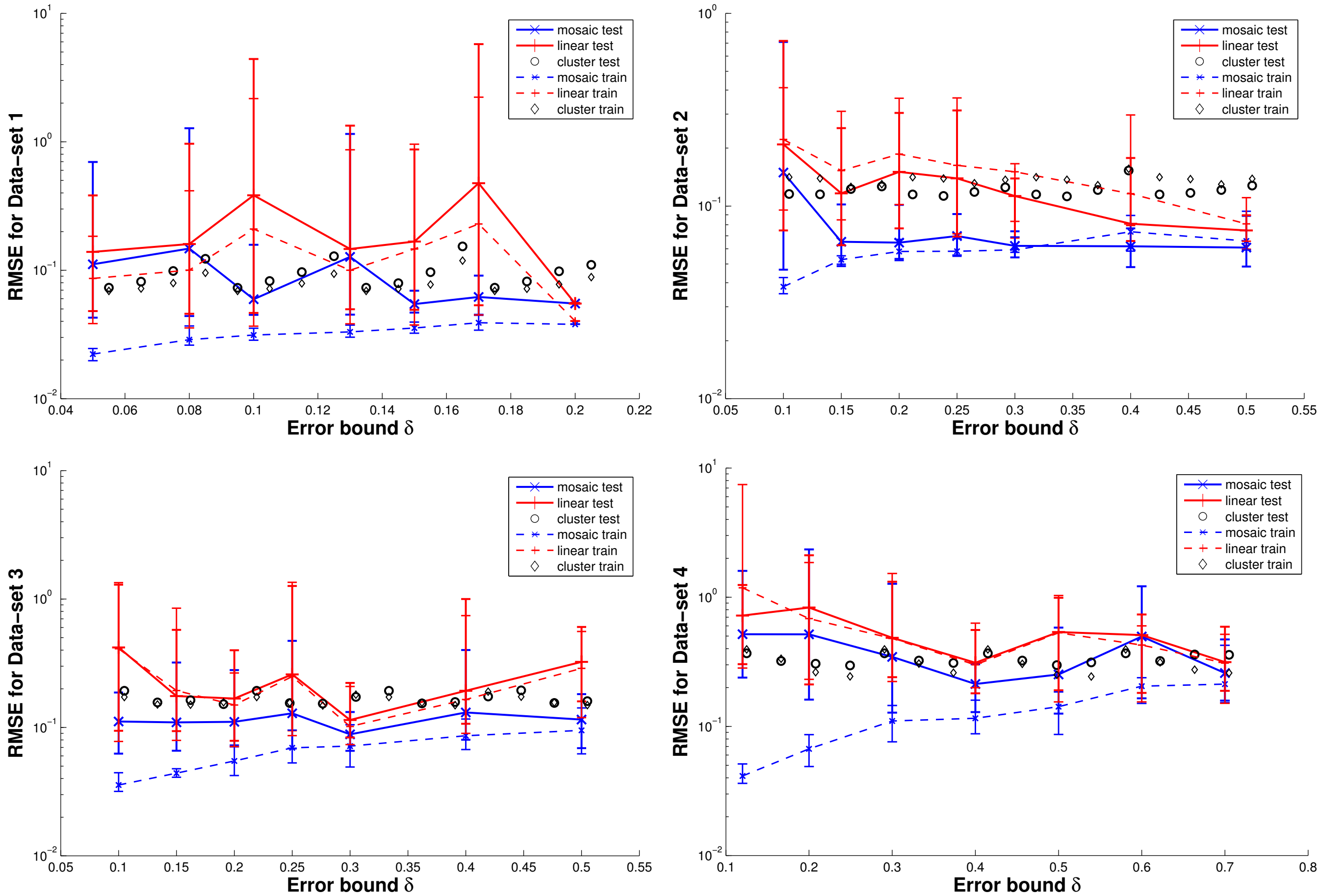
[17] N. Megiddo. On the complexity of polyhedral separability. *Discrete Comput Geom*, 3:325–337, 1988.

[18] N. Megiddo and A. Tamir. On the complexity of locating linear facilities in the plane. *Oper. Res. Lett.*, 1(5):194–197, Nov. 1982.

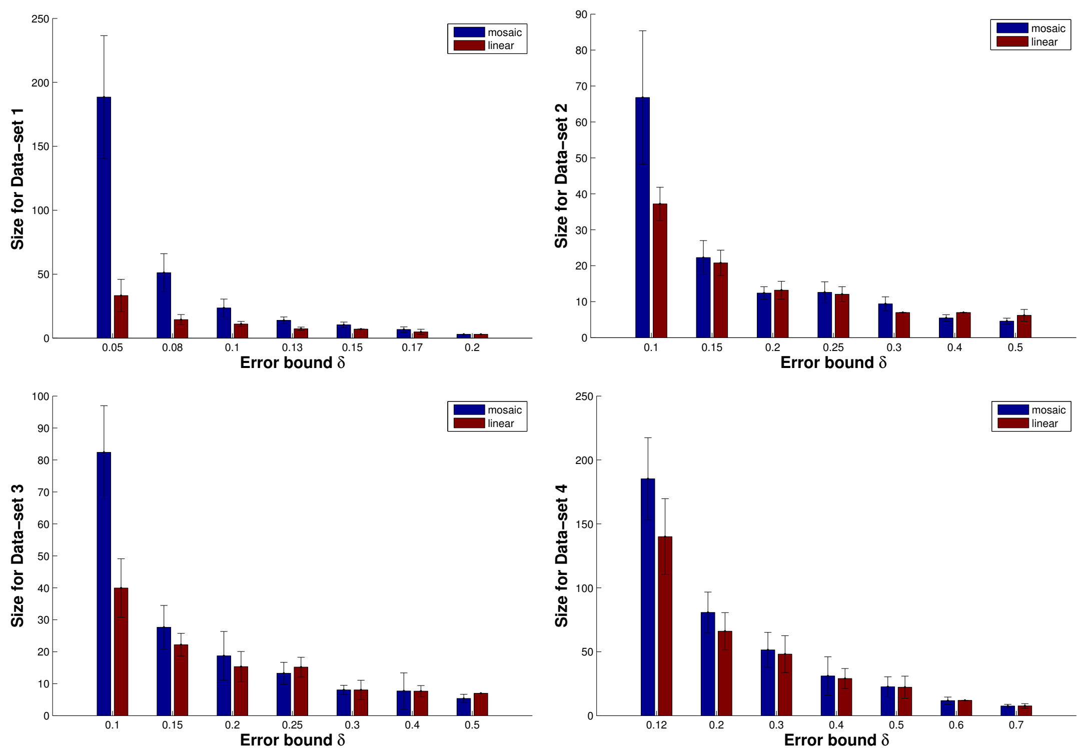
[19] S. Paoletti, A. L. Juloski, G. Ferrari-Trecate, and R. Vidal. Identification of hybrid systems a tutorial. *European Journal of Control*, 13(2-3):242 – 260, 2007.

[20] J. Roll, A. Bemporad, and L. Ljung. Identification of piecewise affine systems via mixed-integer programming. *Automatica*, 40(1):37 – 50, 2004.

[21] R. Vidal, S. Soatto, Y. Ma, and S. Sastry. An algebraic geometric approach to the identification of a class of linear hybrid systems. In *Decision and Control, 2003. Proceedings. 42nd IEEE Conference on*, volume 1, pages 167–172 Vol.1, Dec 2003.



Root Mean Square Error for Data-set 1, 2, 3 and 4



Sizes of models for Data-sets 1, 2, 3, 4

Figure 7: RMSE and Size of models learnt by MOSAIC, LINEAR and CLUSTER on pick and place machine data