Approximate Multiparametric Mixed-integer Convex Programming

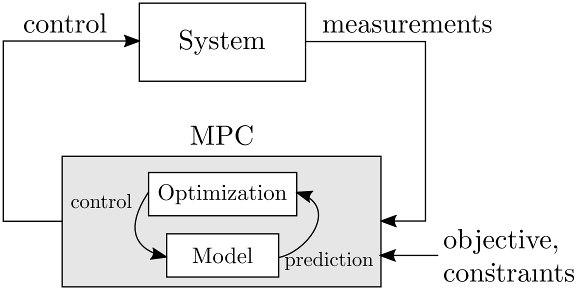
Danylo Malyuta*, Behçet Açıkmeşe

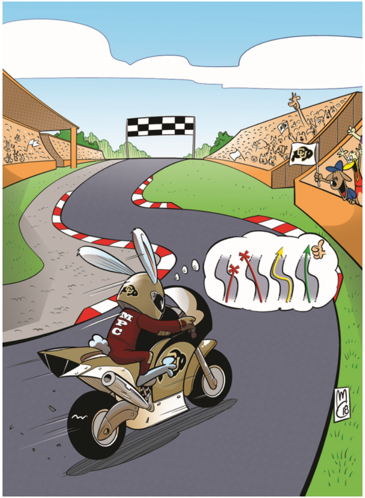
Autonomous Controls Laboratory, University of Washington

* danylo@uw.edu

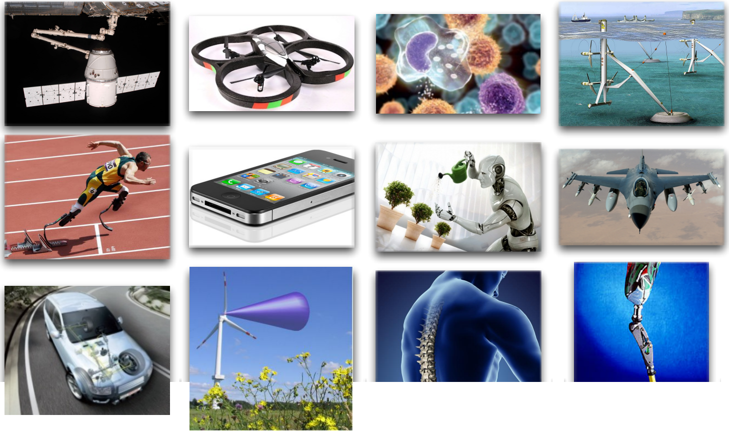
June 21, 2019

Model Predictive Control (MPC)





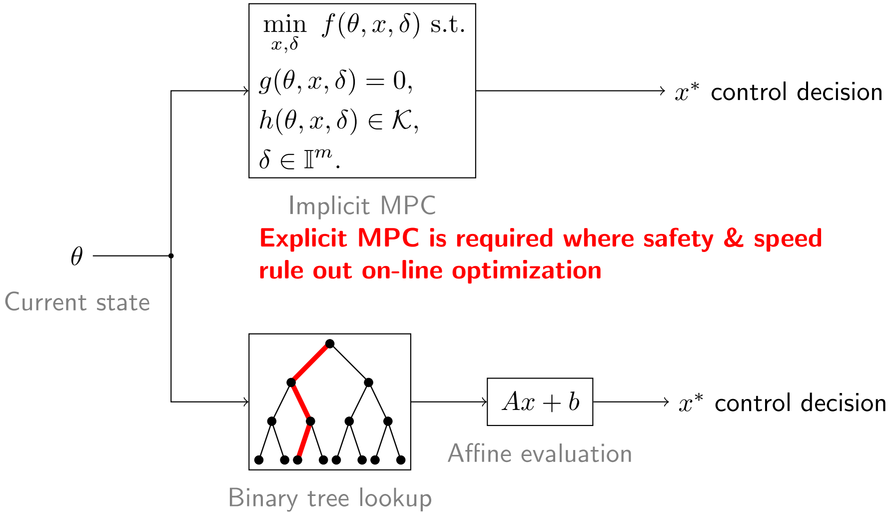
Model Predictive Control (MPC)



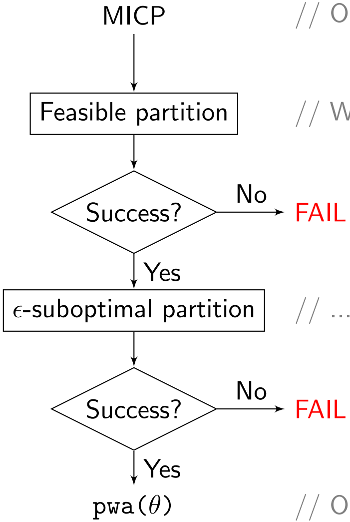
[1]

[1] From Colin Jones' *Control Systems 1* slides at EPFL.

Explicit vs. Implicit MPC



Algorithm Flowchart



// Optimization problem oracle

// We'll talk about this first [1]...

// ...and then about this [2]

// Optimization-free control law

Outline

Partition-based Feasible Integer Solution Pre-computation for Hybrid MPC

Approximate Multiparametric Mixed-integer Convex Programming

Template Optimization Problem

$$V^*(\theta) = \min_{x,\delta} \; f(\theta, x, \delta) \text{ s.t.} \tag{1a}$$

$$g(\theta, x, \delta) = 0, \tag{1b}$$

$$h(\theta, x, \delta) \in \mathcal{K}, \tag{1c}$$

$$\delta \in \mathbb{I}^m. \tag{1d}$$

- Multiparametric mixed-integer conic program (MICP)
- $f(\theta, x, \delta) : \mathbb{R}^p \times \mathbb{R}^n \times \mathbb{I}^m \to \mathbb{R}$ jointly convex in θ and x
- $\{g, h\}(\theta, x, \delta) : \mathbb{R}^p \times \mathbb{R}^n \times \mathbb{I}^m \to \mathbb{R}^{\{n_g, n_h\}}$ affine in θ and x
- $\mathcal{K} = \mathcal{K}_1 \times \cdots \times \mathcal{K}_2 \times \cdots$ convex cone (non-negative orthant, second-order cone, semidefinite cone, etc.)
- Difficult/slow to solve!

Fixed-commutation Version

$$V_\delta^*(\theta) = \min_x \; f(\theta, x, \delta) \text{ s.t.} \quad (2a)$$
$$g(\theta, x, \delta) = 0, \quad (2b)$$
$$h(\theta, x, \delta) \in \mathcal{K}. \quad (2c)$$

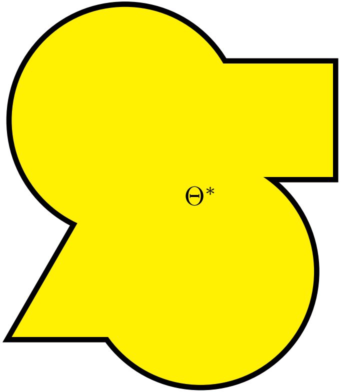
- Multiparametric conic program (CP)
- Commutation $\delta \in \mathbb{I}^m$ is fixed (i.e. chosen)
- Two questions: how to choose δ such that...
  1. ... Problem 2 is feasible?
  2. ... $V_\delta^*(\theta) = V^*(\theta)$ (i.e. the optimal cost is achieved)?

In this presentation we answer these two questions

Definitions

Definition 1

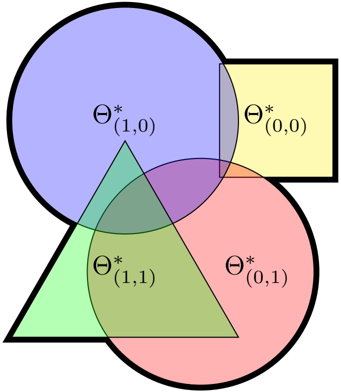
The feasible parameter set $\Theta^* \subset \mathbb{R}^p$ is the set of all θ parameters for which the MICP is feasible.



Definitions

Definition 2

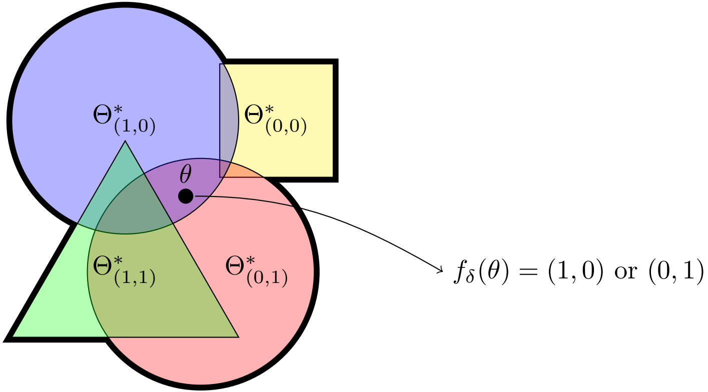
The fixed-commutation feasible parameter set $\Theta^*_\delta \subset \mathbb{R}^p$ is the set of all θ parameters for which the fixed-commutation CP is feasible. Θ^*_δ is convex.



Definitions

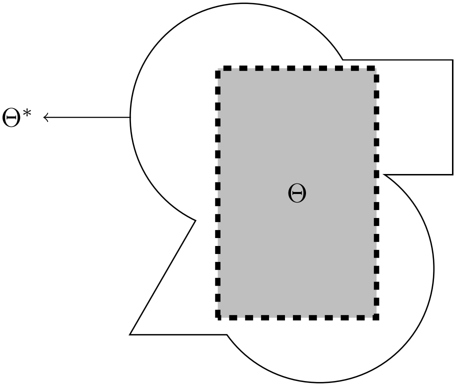
Definition 3

The feasible commutation map $f_\delta : \Theta^* \to \mathbb{I}^m$ maps $\theta \in \Theta^*$ to a commutation δ such that $\theta \in \Theta^*_\delta$ (i.e. the fixed-commutation CP is feasible for this θ).



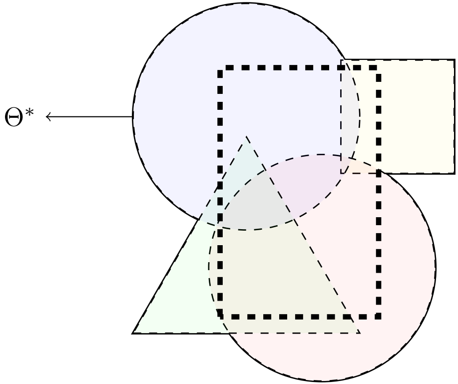
Objective

- Compute f_δ over a subset of its domain $\Theta \subseteq \Theta^*$
- Typically, choose Θ as an invariant set
- f_δ will seed the computation of the explicit control law



General Idea

- Generate a simplicial partition $\mathcal{R} = \{(\mathcal{R}_i, \delta_i)\}_{i=1}^{|\mathcal{R}|}$ such that
  - $\Theta = \bigcup_{i=1}^{|\mathcal{R}|} \mathcal{R}_i$
  - δ_i is feasible everywhere in $\mathcal{R}_i$, i.e $\mathcal{R}_i \subseteq \Theta^*_{\delta_i}$



Brute Force Approach

Algorithm 1 Brute force f_δ computation.

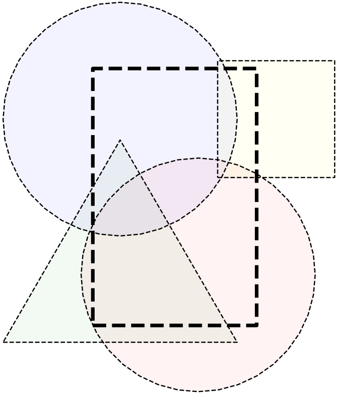
1: $\mathcal{R} \leftarrow \emptyset$, $\bar{\Theta} \leftarrow \Theta$
2: **for** all $\delta \in \mathbb{I}^m$ **do**
3: $\quad \mathcal{R} \leftarrow \{(\mathcal{R}', \delta) : \mathcal{R}' \in \bar{\Theta} \cap \Theta^*_\delta\} \cup \mathcal{R}$
4: $\quad \bar{\Theta} \leftarrow \bar{\Theta} \setminus \Theta^*_\delta$
5: $\quad$ **if** $\bar{\Theta} = \emptyset$ **then**
6: $\quad\quad$ STOP

- Exploit convexity of Θ^*_δ
- Inner-approximation algorithm exists [3]

Brute Force Approach

Algorithm 1 Brute force f_δ computation.

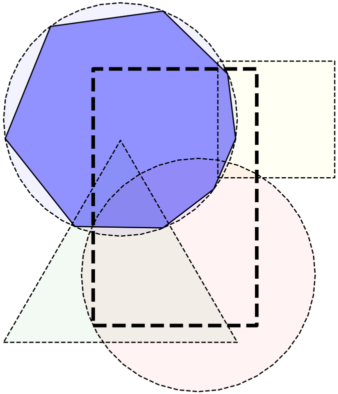
$\mathcal{R} \leftarrow \emptyset$, $\bar{\Theta} \leftarrow \Theta$
for all $\delta \in \mathbb{I}^m$ **do**
    $\mathcal{R} \leftarrow \{(\mathcal{R}', \delta) : \mathcal{R}' \in \bar{\Theta} \cap \Theta^*_\delta\} \cup \mathcal{R}$
    $\bar{\Theta} \leftarrow \bar{\Theta} \setminus \Theta^*_\delta$
    if $\bar{\Theta} = \emptyset$ **then**
        STOP



Brute Force Approach

Algorithm 1 Brute force f_δ computation.

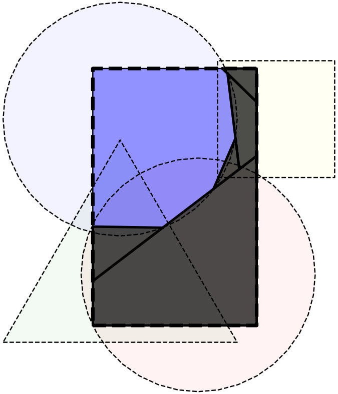
1: $\mathcal{R} \leftarrow \emptyset$, $\bar{\Theta} \leftarrow \Theta$
2: **for** all $\delta \in \mathbb{I}^m$ **do**
3: $\quad \mathcal{R} \leftarrow \{(\mathcal{R}', \delta) : \mathcal{R}' \in \bar{\Theta} \cap \Theta^*_\delta\} \cup \mathcal{R}$
4: $\quad \bar{\Theta} \leftarrow \bar{\Theta} \setminus \Theta^*_\delta$
5: $\quad$ **if** $\bar{\Theta} = \emptyset$ **then**
6: $\quad\quad$ STOP



Brute Force Approach

Algorithm 1 Brute force f_δ computation.

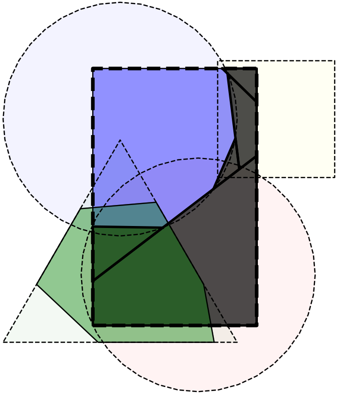
1: $\mathcal{R} \leftarrow \emptyset$, $\bar{\Theta} \leftarrow \Theta$
2: **for** all $\delta \in \mathbb{I}^m$ **do**
3: $\quad \mathcal{R} \leftarrow \{(\mathcal{R}', \delta) : \mathcal{R}' \in \bar{\Theta} \cap \Theta^*_\delta\} \cup \mathcal{R}$
4: $\quad \bar{\Theta} \leftarrow \bar{\Theta} \setminus \Theta^*_\delta$
5: $\quad$ **if** $\bar{\Theta} = \emptyset$ **then**
6: $\quad\quad$ STOP



Brute Force Approach

Algorithm 1 Brute force f_δ computation.

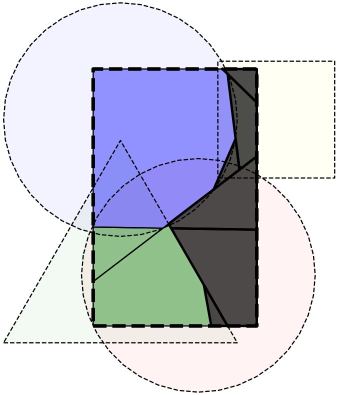
$\mathcal{R} \leftarrow \emptyset$, $\bar{\Theta} \leftarrow \Theta$
for all $\delta \in \mathbb{I}^m$ **do**
    $\mathcal{R} \leftarrow \{(\mathcal{R}', \delta) : \mathcal{R}' \in \bar{\Theta} \cap \Theta^*_\delta\} \cup \mathcal{R}$
    $\bar{\Theta} \leftarrow \bar{\Theta} \setminus \Theta^*_\delta$
    if $\bar{\Theta} = \emptyset$ **then**
        STOP



Brute Force Approach

Algorithm 1 Brute force f_δ computation.

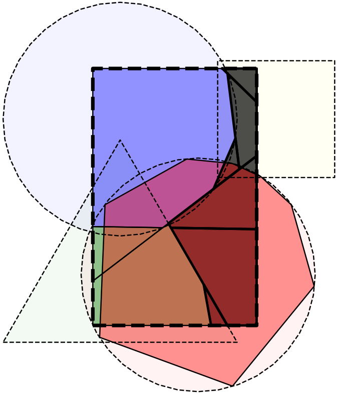
1: $\mathcal{R} \leftarrow \emptyset$, $\bar{\Theta} \leftarrow \Theta$
2: **for** all $\delta \in \mathbb{I}^m$ **do**
3: $\quad \mathcal{R} \leftarrow \{(\mathcal{R}', \delta) : \mathcal{R}' \in \bar{\Theta} \cap \Theta^*_\delta\} \cup \mathcal{R}$
4: $\quad \bar{\Theta} \leftarrow \bar{\Theta} \setminus \Theta^*_\delta$
5: $\quad$ **if** $\bar{\Theta} = \emptyset$ **then**
6: $\quad\quad$ STOP



Brute Force Approach

Algorithm 1 Brute force f_δ computation.

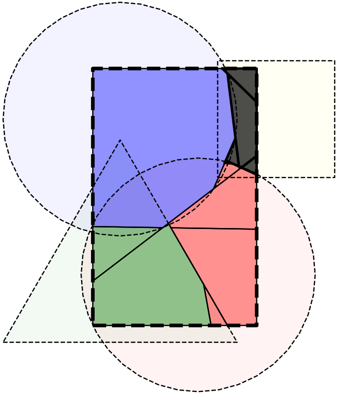
$\mathcal{R} \leftarrow \emptyset$, $\bar{\Theta} \leftarrow \Theta$
for all $\delta \in \mathbb{I}^m$ **do**
    $\mathcal{R} \leftarrow \{(\mathcal{R}', \delta) : \mathcal{R}' \in \bar{\Theta} \cap \Theta^*_\delta\} \cup \mathcal{R}$
    $\bar{\Theta} \leftarrow \bar{\Theta} \setminus \Theta^*_\delta$
    if $\bar{\Theta} = \emptyset$ **then**
        STOP



Brute Force Approach

Algorithm 1 Brute force f_δ computation.

$\mathcal{R} \leftarrow \emptyset$, $\bar{\Theta} \leftarrow \Theta$
for all $\delta \in \mathbb{I}^m$ **do**
    $\mathcal{R} \leftarrow \{(\mathcal{R}', \delta) : \mathcal{R}' \in \bar{\Theta} \cap \Theta^*_\delta\} \cup \mathcal{R}$
    $\bar{\Theta} \leftarrow \bar{\Theta} \setminus \Theta^*_\delta$
    if $\bar{\Theta} = \emptyset$ **then**
        STOP



Desirable Algorithm Properties

Disadvantages of brute force:

- No attempt to go around the combinatorial complexity
- Inner approximation of Θ_δ^* is very slow in high dimensions
- Polytopic set intersection and set difference are numerically poor

A better algorithm:

- Explores all $\delta \in \mathcal{I}^m$ combinations *only in the worst case*
- Minimizes vertex count
- Only uses numerically robust operations

Our algorithm achieves these properties by:

- Solving a MICP to find a feasible δ for a current subset
- Using simplex partition cells
- Doing everything in vertex representation

Proposed Algorithm

Algorithm 2 Proposed computation of f_δ.

```
Create empty tree with open leaf Θ as root
Triangulate Θ (delaunay)
while any non-leaf node exists do
    R ← most recently added node
    if MICP (1) infeasible for θ = c^R then
        STOP, (Θ*)^c ∩ Θ ≠ ∅
    else
        δ̂ ← solve (3)
        if (3) is infeasible then
            Split R in half along longest edge
        else
            Replace with leaf (R, δ̂)
```

- Key idea: checking if $\mathcal{R} \subseteq \Theta^*_\delta$ is a MICP

Lemma 4

$\mathcal{R} \subseteq \Theta^*_\delta \Leftrightarrow$ the fixed-commutation CP is feasible at all vertices of $\mathcal{R}$.

$$\delta(\mathcal{R}) = \text{find } \delta \text{ s.t.} \tag{3a}$$

$$g(\theta, x_\theta, \delta) = 0, \quad \forall \theta \in \mathcal{V}(\mathcal{R}), \tag{3b}$$

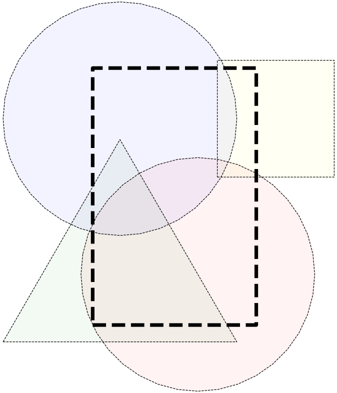
$$h(\theta, x_\theta, \delta) \in \mathcal{K}, \quad \forall \theta \in \mathcal{V}(\mathcal{R}), \tag{3c}$$

$$\delta \in \mathbb{I}^m. \tag{3d}$$

Proposed Algorithm

Algorithm 2 Proposed computation of f_δ.

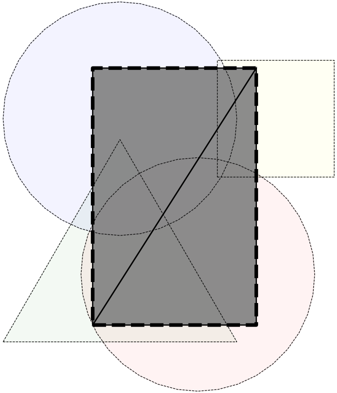
```
Create empty tree with open leaf Θ as root
Triangulate Θ (delaunay)
while any non-leaf node exists do
    R ← most recently added node
    if MICP (1) infeasible for θ = c^R then
        STOP, (Θ*)^c ∩ Θ ≠ ∅
    else
        δ̂ ← solve (3)
        if (3) is infeasible then
            Split R in half along longest edge
        else
            Replace with leaf (R, δ̂)
```



Proposed Algorithm

Algorithm 2 Proposed computation of f_δ.

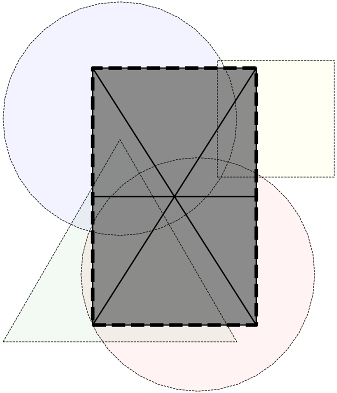
Create empty tree with open leaf Θ as root
Triangulate Θ (`delaunay`)
while any non-leaf node exists **do**
    $\mathcal{R} \leftarrow$ most recently added node
    if MICP (1) infeasible for $\theta = c^{\mathcal{R}}$ **then**
        STOP, $(\Theta^*)^c \cap \Theta \neq \emptyset$
    else
        $\hat{\delta} \leftarrow$ solve (3)
        if (3) is infeasible **then**
            Split $\mathcal{R}$ in half along longest edge
        else
            Replace with leaf $(\mathcal{R}, \hat{\delta})$



Proposed Algorithm

Algorithm 2 Proposed computation of f_δ.

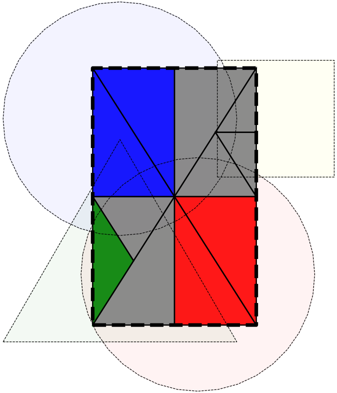
Create empty tree with open leaf Θ as root
Triangulate Θ (`delaunay`)
while any non-leaf node exists **do**
  $\mathcal{R} \leftarrow$ most recently added node
  if MICP (1) infeasible for $\theta = c^{\mathcal{R}}$ **then**
    STOP, $(\Theta^*)^c \cap \Theta \neq \emptyset$
  else
    $\hat{\delta} \leftarrow$ solve (3)
    if (3) is infeasible **then**
      Split $\mathcal{R}$ in half along longest edge
    else
      Replace with leaf $(\mathcal{R}, \hat{\delta})$



Proposed Algorithm

Algorithm 2 Proposed computation of f_δ.

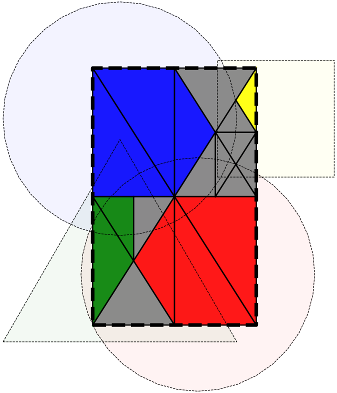
Create empty tree with open leaf Θ as root
Triangulate Θ (`delaunay`)
while any non-leaf node exists **do**
    $\mathcal{R} \leftarrow$ most recently added node
    if MICP (1) infeasible for $\theta = c^{\mathcal{R}}$ **then**
        STOP, $(\Theta^*)^c \cap \Theta \neq \emptyset$
    else
        $\hat{\delta} \leftarrow$ solve (3)
        if (3) is infeasible **then**
            Split $\mathcal{R}$ in half along longest edge
        else
            Replace with leaf $(\mathcal{R}, \hat{\delta})$



Proposed Algorithm

Algorithm 2 Proposed computation of f_δ.

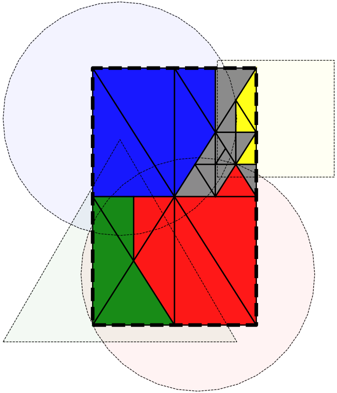
Create empty tree with open leaf Θ as root
Triangulate Θ (`delaunay`)
while any non-leaf node exists **do**
  $\mathcal{R} \leftarrow$ most recently added node
  if MICP (1) infeasible for $\theta = c^{\mathcal{R}}$ **then**
    STOP, $(\Theta^*)^c \cap \Theta \neq \emptyset$
  else
    $\hat{\delta} \leftarrow$ solve (3)
    if (3) is infeasible **then**
      Split $\mathcal{R}$ in half along longest edge
    else
      Replace with leaf $(\mathcal{R}, \hat{\delta})$



Proposed Algorithm

Algorithm 2 Proposed computation of f_δ.

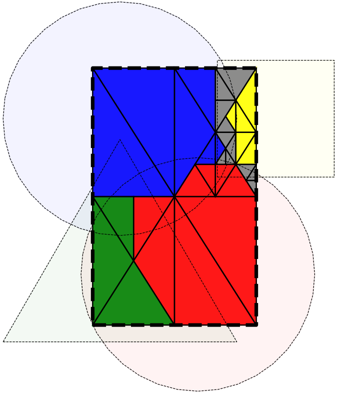
```
Create empty tree with open leaf Θ as root
Triangulate Θ (delaunay)
while any non-leaf node exists do
    R ← most recently added node
    if MICP (1) infeasible for θ = c^R then
        STOP, (Θ*)^c ∩ Θ ≠ ∅
    else
        δ̂ ← solve (3)
        if (3) is infeasible then
            Split R in half along longest edge
        else
            Replace with leaf (R, δ̂)
```



Proposed Algorithm

Algorithm 2 Proposed computation of f_δ.

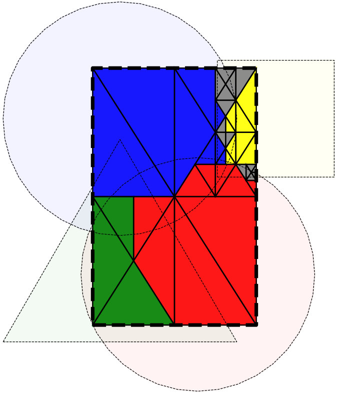
Create empty tree with open leaf Θ as root
Triangulate Θ (`delaunay`)
while any non-leaf node exists **do**
    $\mathcal{R} \leftarrow$ most recently added node
    if MICP (1) infeasible for $\theta = c^{\mathcal{R}}$ **then**
        STOP, $(\Theta^*)^c \cap \Theta \neq \emptyset$
    else
        $\hat{\delta} \leftarrow$ solve (3)
        if (3) is infeasible **then**
            Split $\mathcal{R}$ in half along longest edge
        else
            Replace with leaf $(\mathcal{R}, \hat{\delta})$



Proposed Algorithm

Algorithm 2 Proposed computation of f_δ.

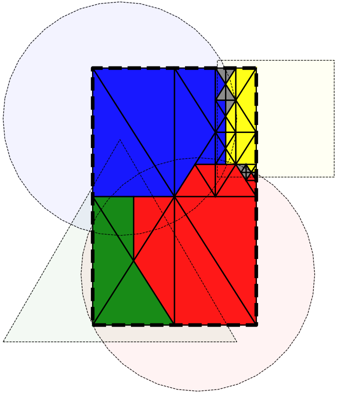
```
Create empty tree with open leaf Θ as root
Triangulate Θ (delaunay)
while any non-leaf node exists do
    R ← most recently added node
    if MICP (1) infeasible for θ = c^R then
        STOP, (Θ*)^c ∩ Θ ≠ ∅
    else
        δ̂ ← solve (3)
        if (3) is infeasible then
            Split R in half along longest edge
        else
            Replace with leaf (R, δ̂)
```



Proposed Algorithm

Algorithm 2 Proposed computation of f_δ.

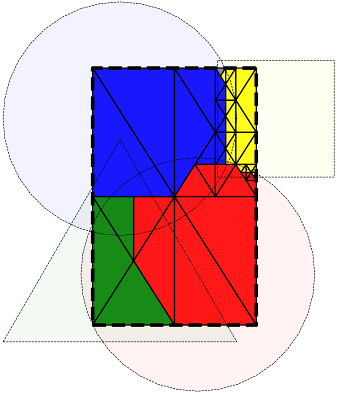
```
Create empty tree with open leaf Θ as root
Triangulate Θ (delaunay)
while any non-leaf node exists do
    R ← most recently added node
    if MICP (1) infeasible for θ = c^R then
        STOP, (Θ*)^c ∩ Θ ≠ ∅
    else
        δ̂ ← solve (3)
        if (3) is infeasible then
            Split R in half along longest edge
        else
            Replace with leaf (R, δ̂)
```



Proposed Algorithm

Algorithm 2 Proposed computation of f_δ.

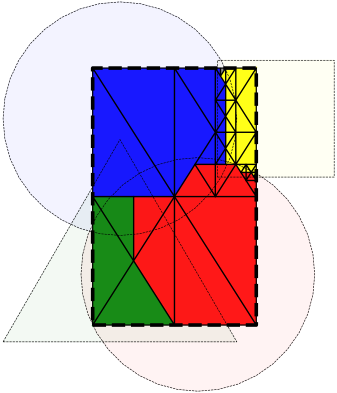
Create empty tree with open leaf Θ as root
Triangulate Θ (`delaunay`)
while any non-leaf node exists **do**
    $\mathcal{R} \leftarrow$ most recently added node
    if MICP (1) infeasible for $\theta = c^{\mathcal{R}}$ **then**
        STOP, $(\Theta^*)^c \cap \Theta \neq \emptyset$
    else
        $\hat{\delta} \leftarrow$ solve (3)
        if (3) is infeasible **then**
            Split $\mathcal{R}$ in half along longest edge
        else
            Replace with leaf $(\mathcal{R}, \hat{\delta})$



Proposed Algorithm

Algorithm 2 Proposed computation of f_δ.

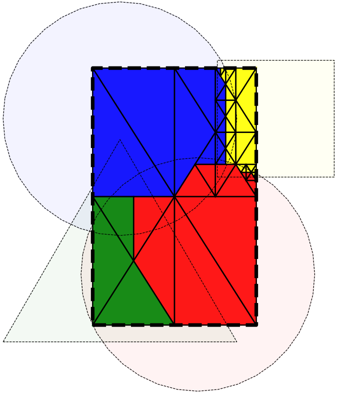
Create empty tree with open leaf Θ as root
Triangulate Θ (`delaunay`)
while any non-leaf node exists **do**
    $\mathcal{R} \leftarrow$ most recently added node
    if MICP (1) infeasible for $\theta = c^{\mathcal{R}}$ **then**
        STOP, $(\Theta^*)^c \cap \Theta \neq \emptyset$
    else
        $\hat{\delta} \leftarrow$ solve (3)
        if (3) is infeasible **then**
            Split $\mathcal{R}$ in half along longest edge
        else
            Replace with leaf $(\mathcal{R}, \hat{\delta})$



Proposed Algorithm

Algorithm 2 Proposed computation of f_δ.

```
Create empty tree with open leaf Θ as root
Triangulate Θ (delaunay)
while any non-leaf node exists do
    R ← most recently added node
    if MICP (1) infeasible for θ = c^R then
        STOP, (Θ*)^c ∩ Θ ≠ ∅
    else
        δ̂ ← solve (3)
        if (3) is infeasible then
            Split R in half along longest edge
        else
            Replace with leaf (R, δ̂)
```



Definition 4

Let $\Delta \triangleq \{\delta \in \mathbb{I}^m : \Theta^*_\delta \cap \Theta \neq \emptyset\}$. The largest value $\kappa \in \mathbb{R}_+$ such that $\forall \theta \in \Theta \; \exists \delta \in \Delta$ such that $(\kappa\mathbb{B} + \theta) \setminus (\Theta^* \cap \Theta)^c \subseteq \Theta^*_\delta$ is called the overlap.

Assumption 1

The overlap is positive, i.e. $\kappa > 0$.

(a) Bad situation: $\kappa = 0$.

(b) Good situation: $\kappa > 0$.

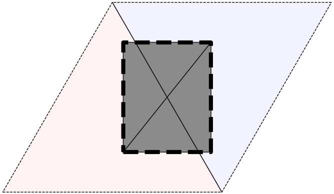
Figure: Illustration of zero (bad) and positive (good) overlap.

What Happens When the Overlap is Zero?

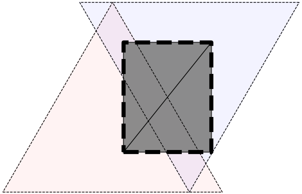
(a) Bad situation: $\kappa = 0$. (b) Good situation: $\kappa > 0$.

Figure: Illustration of zero (bad) and positive (good) overlap. Our algorithm generally (i.e. quasi-always) does not converge if $\kappa = 0$.

What Happens When the Overlap is Zero?



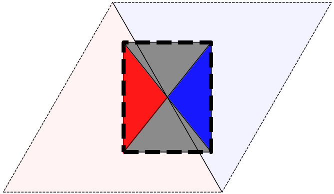
(a) Bad situation: $\kappa = 0$.



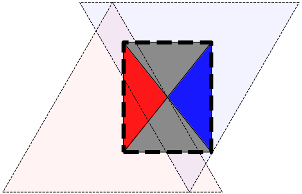
(b) Good situation: $\kappa > 0$.

Figure: Illustration of zero (bad) and positive (good) overlap. Our algorithm generally (i.e. quasi-always) does not converge if $\kappa = 0$.

What Happens When the Overlap is Zero?



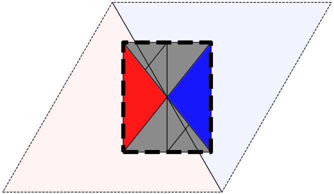
(a) Bad situation: $\kappa = 0$.



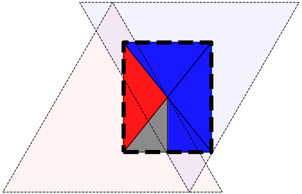
(b) Good situation: $\kappa > 0$.

Figure: Illustration of zero (bad) and positive (good) overlap. Our algorithm generally (i.e. quasi-always) does not converge if $\kappa = 0$.

What Happens When the Overlap is Zero?



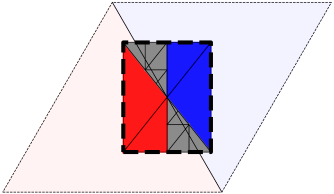
(a) Bad situation: $\kappa = 0$.



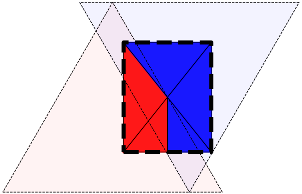
(b) Good situation: $\kappa > 0$.

Figure: Illustration of zero (bad) and positive (good) overlap. Our algorithm generally (i.e. quasi-always) does not converge if $\kappa = 0$.

What Happens When the Overlap is Zero?



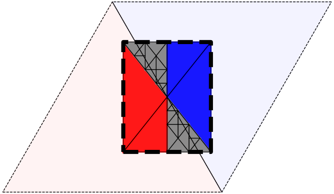
(a) Bad situation: $\kappa = 0$.



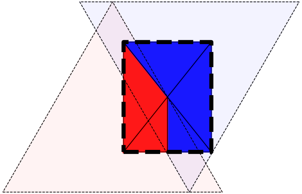
(b) Good situation: $\kappa > 0$.

Figure: Illustration of zero (bad) and positive (good) overlap. Our algorithm generally (i.e. quasi-always) does not converge if $\kappa = 0$.

What Happens When the Overlap is Zero?



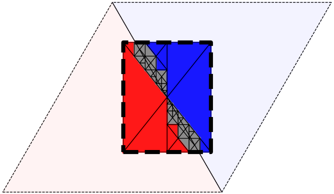
(a) Bad situation: $\kappa = 0$.



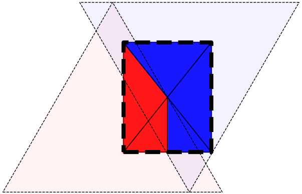
(b) Good situation: $\kappa > 0$.

Figure: Illustration of zero (bad) and positive (good) overlap. Our algorithm generally (i.e. quasi-always) does not converge if $\kappa = 0$.

What Happens When the Overlap is Zero?



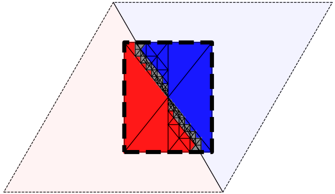
(a) Bad situation: $\kappa = 0$.



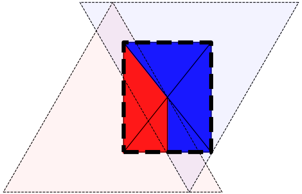
(b) Good situation: $\kappa > 0$.

Figure: Illustration of zero (bad) and positive (good) overlap. Our algorithm generally (i.e. quasi-always) does not converge if $\kappa = 0$.

What Happens When the Overlap is Zero?



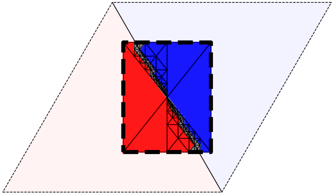
(a) Bad situation: $\kappa = 0$.



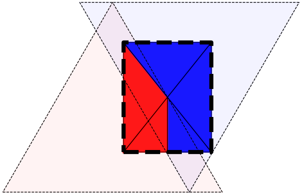
(b) Good situation: $\kappa > 0$.

Figure: Illustration of zero (bad) and positive (good) overlap. Our algorithm generally (i.e. quasi-always) does not converge if $\kappa = 0$.

What Happens When the Overlap is Zero?



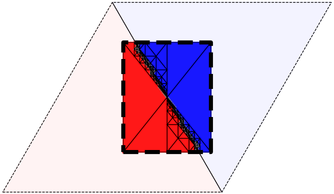
(a) Bad situation: $\kappa = 0$.



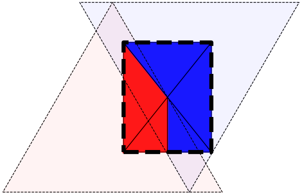
(b) Good situation: $\kappa > 0$.

Figure: Illustration of zero (bad) and positive (good) overlap. Our algorithm generally (i.e. quasi-always) does not converge if $\kappa = 0$.

What Happens When the Overlap is Zero?



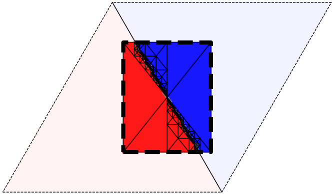
(a) Bad situation: $\kappa = 0$.



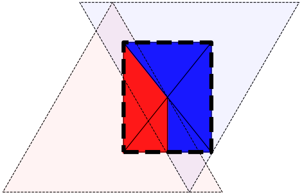
(b) Good situation: $\kappa > 0$.

Figure: Illustration of zero (bad) and positive (good) overlap. Our algorithm generally (i.e. quasi-always) does not converge if $\kappa = 0$.

What Happens When the Overlap is Zero?



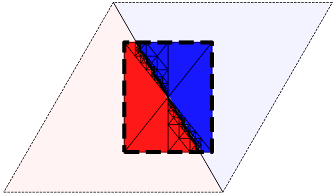
(a) Bad situation: $\kappa = 0$.



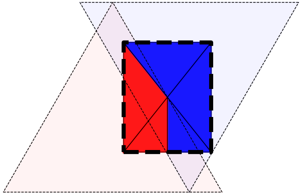
(b) Good situation: $\kappa > 0$.

Figure: Illustration of zero (bad) and positive (good) overlap. Our algorithm generally (i.e. quasi-always) does not converge if $\kappa = 0$.

What Happens When the Overlap is Zero?



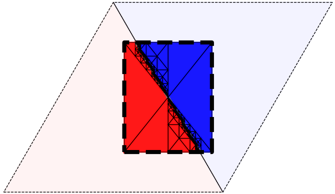
(a) Bad situation: $\kappa = 0$.



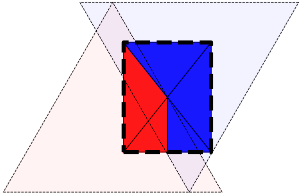
(b) Good situation: $\kappa > 0$.

Figure: Illustration of zero (bad) and positive (good) overlap. Our algorithm generally (i.e. quasi-always) does not converge if $\kappa = 0$.

What Happens When the Overlap is Zero?



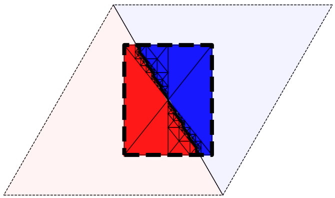
(a) Bad situation: $\kappa = 0$.



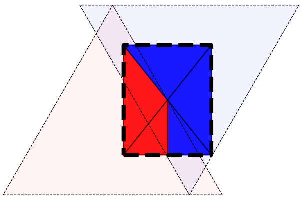
(b) Good situation: $\kappa > 0$.

Figure: Illustration of zero (bad) and positive (good) overlap. Our algorithm generally (i.e. quasi-always) does not converge if $\kappa = 0$.

What Happens When the Overlap is Zero?



(a) Bad situation: $\kappa = 0$.



(b) Good situation: $\kappa > 0$.

Figure: Illustration of zero (bad) and positive (good) overlap. Our algorithm generally (i.e. quasi-always) does not converge if $\kappa = 0$.

Guaranteed Convergence for Non-Zero Overlap

Theorem 5

If the overlap is positive then our algorithm either converges or fails in a finite number of iterations.

Lemma 6

If the overlap is zero, our algorithm generally (read: almost always) does not converge.

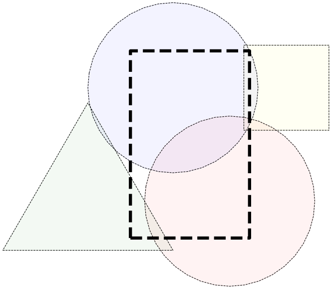


Figure: Illustration of failure in a finite number of iterations.

Guaranteed Convergence for Non-Zero Overlap

Theorem 5

If the overlap is positive then our algorithm either converges or fails in a finite number of iterations.

Lemma 6

If the overlap is zero, our algorithm generally (read: almost always) does not converge.

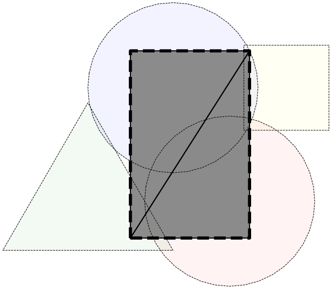


Figure: Illustration of failure in a finite number of iterations.

Guaranteed Convergence for Non-Zero Overlap

Theorem 5

If the overlap is positive then our algorithm either converges or fails in a finite number of iterations.

Lemma 6

If the overlap is zero, our algorithm generally (read: almost always) does not converge.

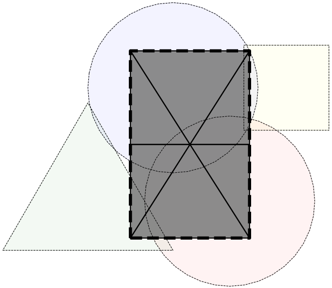


Figure: Illustration of failure in a finite number of iterations.

Guaranteed Convergence for Non-Zero Overlap

Theorem 5
If the overlap is positive then our algorithm either converges or fails in a finite number of iterations.

Lemma 6
If the overlap is zero, our algorithm generally (read: almost always) does not converge.

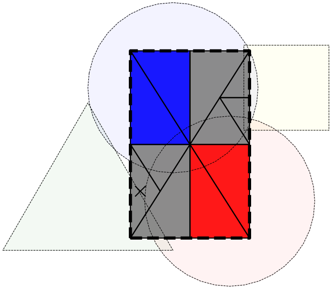


Figure: Illustration of failure in a finite number of iterations.

Partition Complexity

Recall:

- Partition is stored as a binary tree
- Parameter $\theta \in \mathbb{R}^p$

Theorem 7

The worst-case partition tree depth τ is $\mathcal{O}(p^2 \log(\kappa^{-1}))$.

Corollary 8

The worst-case tree leaf count η is $\mathcal{O}(2^{p^2 \log(\kappa^{-1})})$.

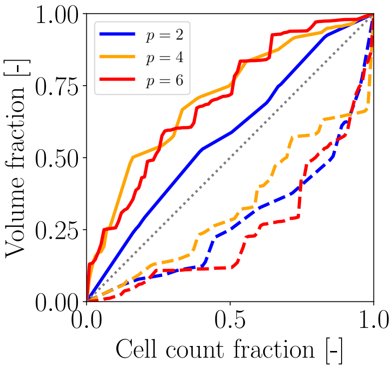
Theorem 9

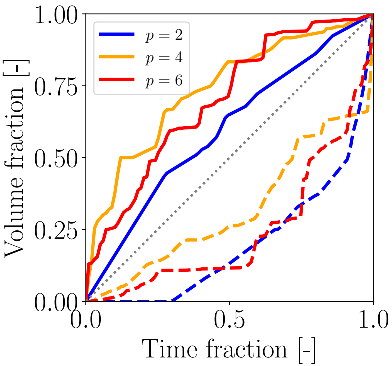
The on-line evaluation complexity of f_δ is $\mathcal{O}(p^4)$, i.e. polynomial time.

Illustrative Example

Multiple DOF controllable oscillator [1]:

$$M\ddot{r} + C\dot{r} + Kr = Lu. \quad (3)$$

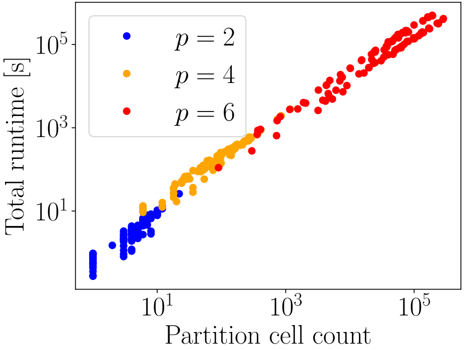


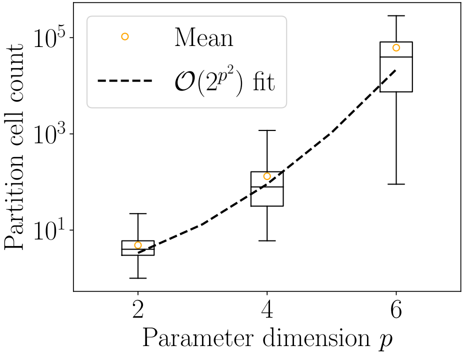


Illustrative Example

Multiple DOF controllable oscillator [1]:

$$M\ddot{r} + C\dot{r} + Kr = Lu. \quad (3)$$





Outline

Partition-based Feasible Integer Solution Pre-computation for Hybrid MPC

Approximate Multiparametric Mixed-integer Convex Programming

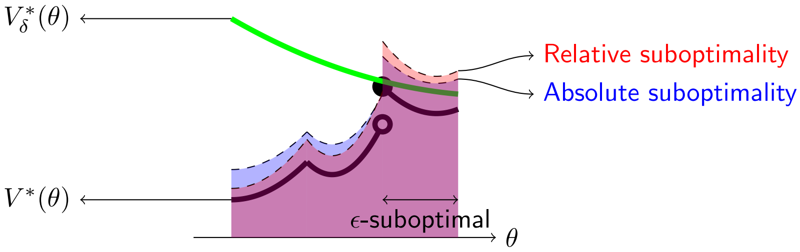
Definitions

Definition 10

The suboptimal commutation map $f_\delta^\epsilon : \Theta^* \to \mathbb{I}^m$ associates $\theta \in \Theta^*$ to an ϵ-suboptimal commutation δ such that

$$V_\delta^*(\theta) - V^*(\theta) < \max\{\epsilon_\mathrm{a}, \epsilon_\mathrm{r} V^*(\theta)\}, \tag{4}$$

where ϵ_a and ϵ_r are the absolute and relative errors.



Checking if δ is ϵ-suboptimal

$$V_\delta^*(\theta) - V^*(\theta) < \max\{\epsilon_{\mathrm{a}}, \epsilon_{\mathrm{r}} V^*(\theta)\}. \tag{5}$$

- If (8) does not hold, then the following is feasible:

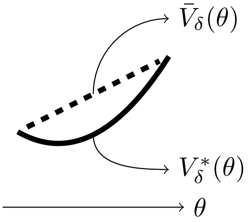
$$\delta^*, \theta^* = \underset{\theta \in \mathcal{R}}{\text{find}} \;\; \delta' \text{ s.t.} \tag{6a}$$

$$V_\delta^*(\theta) - V_{\delta'}^*(\theta) \geq \max\{\epsilon_{\mathrm{a}}, \epsilon_{\mathrm{r}} V_{\delta'}^*(\theta)\}. \tag{6b}$$

- Since (6b) is non-convex, use a convex approximation:

$$\delta^*, \theta^* = \underset{\theta \in \mathcal{R}}{\text{find}} \;\; \delta' \text{ s.t.} \tag{7a}$$

$$\bar{V}_\delta(\theta) - V_{\delta'}^*(\theta) \geq \max\{\epsilon_{\mathrm{a}}, \epsilon_{\mathrm{r}} V_{\delta'}^*(\theta)\}. \tag{7b}$$



Affine over-approximator:

$$\bar{V}_\delta(\theta) \triangleq \sum_{i=1}^{|\mathcal{V}(\mathcal{R})|} \alpha_i V_\delta^*(v_i).$$

Checking if δ is ϵ-suboptimal

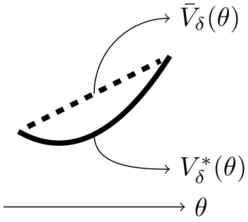
$$V_\delta^*(\theta) - V^*(\theta) < \max\{\epsilon_\text{a}, \epsilon_\text{r} V^*(\theta)\}. \tag{8}$$

- To prevent oscillations, ensure δ^* is feasible over $\mathcal{R}$:

$$\begin{aligned} \delta^*, \theta^* = \underset{\theta\in\mathcal{R}}{\text{find}}\ & \delta' \text{ s.t.} && \text{(9a)} \\ & \bar{V}_\delta(\theta) - V_{\delta'}^*(\theta) \geq \max\{\epsilon_\text{a}, \epsilon_\text{r} V_{\delta'}^*(\theta)\}, && \text{(9b)} \\ & \delta' \in \{\delta'' \in \mathbb{I}^m \setminus \{\delta\} : \mathcal{R} \subseteq \Theta_{\delta''}^*\}. && \text{(9c)} \end{aligned}$$

- To prevent excessive partitioning, only partition when:

$$\max_{\theta\in\mathcal{R}} V_\delta^*(\theta) - \min_{\theta\in\mathcal{R}} V_\delta^*(\theta) < \max\{\epsilon_\text{a}, \epsilon_\text{r} V_{\delta^*}^*(\theta^*)\}, \tag{10}$$



Affine over-approximator:

$$\bar{V}_\delta(\theta) \triangleq \sum_{i=1}^{|\mathcal{V}(\mathcal{R})|} \alpha_i V_\delta^*(v_i).$$

Checking if δ is ϵ-suboptimal

$$V_\delta^*(\theta) - V^*(\theta) < \max\{\epsilon_\mathrm{a}, \epsilon_\mathrm{r} V^*(\theta)\}. \tag{8}$$

- To prevent oscillations, ensure δ^* is feasible over $\mathcal{R}$:

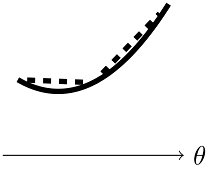
$$\delta^*, \theta^* = \underset{\theta \in \mathcal{R}}{\mathrm{find}} \; \delta' \text{ s.t.} \tag{9a}$$

$$\bar{V}_\delta(\theta) - V_{\delta'}^*(\theta) \geq \max\{\epsilon_\mathrm{a}, \epsilon_\mathrm{r} V_{\delta'}^*(\theta)\}, \tag{9b}$$

$$\delta' \in \{\delta'' \in \mathbb{I}^m \setminus \{\delta\} : \mathcal{R} \subseteq \Theta_{\delta''}^*\}. \tag{9c}$$

- To prevent excessive partitioning, only partition when:

$$\max_{\theta \in \mathcal{R}} V_\delta^*(\theta) - \min_{\theta \in \mathcal{R}} V_\delta^*(\theta) < \max\{\epsilon_\mathrm{a}, \epsilon_\mathrm{r} V_{\delta^*}^*(\theta^*)\}, \tag{10}$$



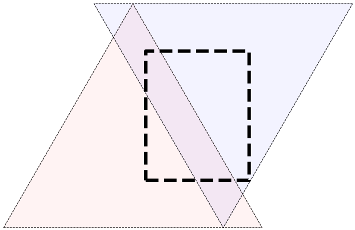
Affine over-approximator:

$$\bar{V}_\delta(\theta) \triangleq \sum_{i=1}^{|\mathcal{V}(\mathcal{R})|} \alpha_i V_\delta^*(v_i).$$

Partitioning Algorithm

Algorithm 3 Computation of f_δ^ϵ.

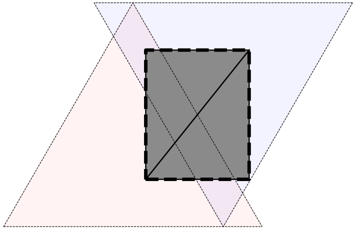
Create feasible partition (Algorithm 2)
while any nodes exist **do**
    $(\mathcal{R}, \delta) \leftarrow$ most recently added node
    if (7) infeasible **then**
        Change node to leaf
    else
        $\delta^*, \theta^* \leftarrow$ solve (9)
        if (9) feasible and (10) holds **then**
            Change node to $(\mathcal{R}, \delta^*)$
        else
            $\delta^* \leftarrow \delta$ if (10) infeasible
            Split $\mathcal{R}$ in half along longest edge
            Add nodes using δ^*



Partitioning Algorithm

Algorithm 3 Computation of f_δ^ϵ.

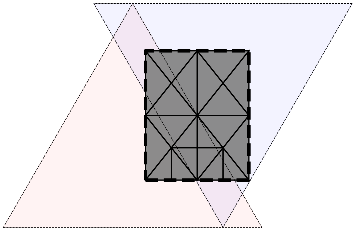
Create feasible partition (Algorithm 2)
while any nodes exist **do**
  $(\mathcal{R}, \delta) \leftarrow$ most recently added node
  if (7) infeasible **then**
    Change node to leaf
  else
    $\delta^*, \theta^* \leftarrow$ solve (9)
    if (9) feasible and (10) holds **then**
      Change node to $(\mathcal{R}, \delta^*)$
    else
      $\delta^* \leftarrow \delta$ if (10) infeasible
      Split $\mathcal{R}$ in half along longest edge
      Add nodes using δ^*



Partitioning Algorithm

Algorithm 3 Computation of f_δ^ϵ.

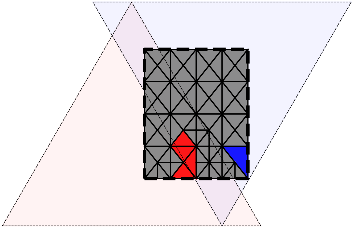
Create feasible partition (Algorithm 2)
while any nodes exist **do**
$(\mathcal{R}, \delta) \leftarrow$ most recently added node
if (7) infeasible **then**
Change node to leaf
else
$\delta^*, \theta^* \leftarrow$ solve (9)
if (9) feasible and (10) holds **then**
Change node to $(\mathcal{R}, \delta^*)$
else
$\delta^* \leftarrow \delta$ if (10) infeasible
Split $\mathcal{R}$ in half along longest edge
Add nodes using δ^*



Partitioning Algorithm

Algorithm 3 Computation of f_δ^ϵ.

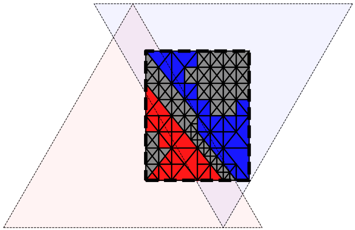
Create feasible partition (Algorithm 2)
while any nodes exist **do**
    $(\mathcal{R}, \delta) \leftarrow$ most recently added node
    if (7) infeasible **then**
        Change node to leaf
    else
        $\delta^*, \theta^* \leftarrow$ solve (9)
        if (9) feasible and (10) holds **then**
            Change node to $(\mathcal{R}, \delta^*)$
        else
            $\delta^* \leftarrow \delta$ if (10) infeasible
            Split $\mathcal{R}$ in half along longest edge
            Add nodes using δ^*



Partitioning Algorithm

Algorithm 3 Computation of f_δ^ϵ.

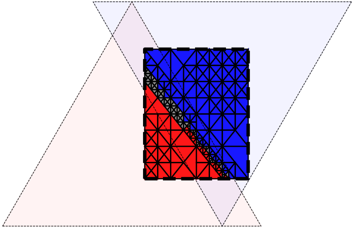
Create feasible partition (Algorithm 2)
while any nodes exist **do**
  $(\mathcal{R}, \delta) \leftarrow$ most recently added node
  if (7) infeasible **then**
    Change node to leaf
  else
    $\delta^*, \theta^* \leftarrow$ solve (9)
    if (9) feasible and (10) holds **then**
      Change node to $(\mathcal{R}, \delta^*)$
    else
      $\delta^* \leftarrow \delta$ if (10) infeasible
      Split $\mathcal{R}$ in half along longest edge
      Add nodes using δ^*



Partitioning Algorithm

Algorithm 3 Computation of f_δ^ϵ.

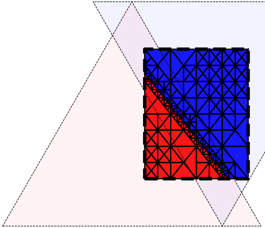
Create feasible partition (Algorithm 2)
while any nodes exist **do**
  $(\mathcal{R}, \delta) \leftarrow$ most recently added node
  if (7) infeasible **then**
    Change node to leaf
  else
    $\delta^*, \theta^* \leftarrow$ solve (9)
    if (9) feasible and (10) holds **then**
      Change node to $(\mathcal{R}, \delta^*)$
    else
      $\delta^* \leftarrow \delta$ if (10) infeasible
      Split $\mathcal{R}$ in half along longest edge
      Add nodes using δ^*



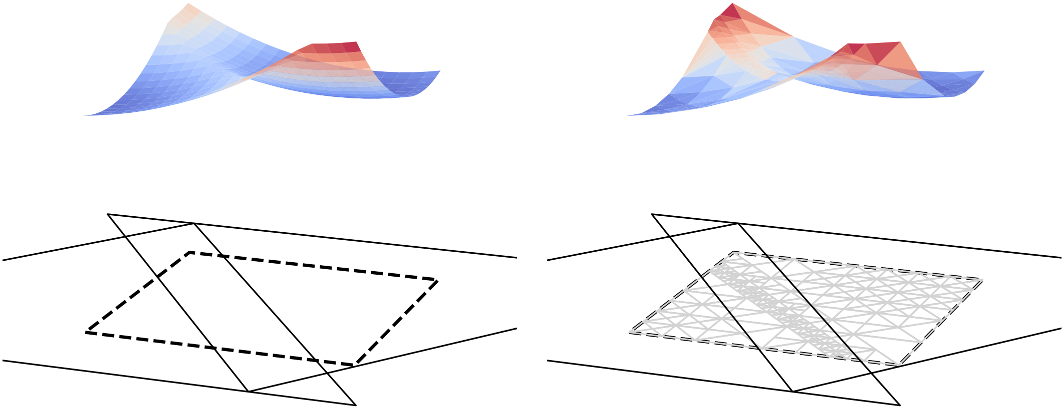
Partitioning Algorithm

Algorithm 3 Computation of f_δ^ϵ.

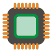
Create feasible partition (Algorithm 2)
while any nodes exist **do**
  $(\mathcal{R}, \delta) \leftarrow$ most recently added node
  if (7) infeasible **then**
    Change node to leaf
  else
    $\delta^*, \theta^* \leftarrow$ solve (9)
    if (9) feasible and (10) holds **then**
      Change node to $(\mathcal{R}, \delta^*)$
    else
      $\delta^* \leftarrow \delta$ if (10) infeasible
      Split $\mathcal{R}$ in half along longest edge
      Add nodes using δ^*



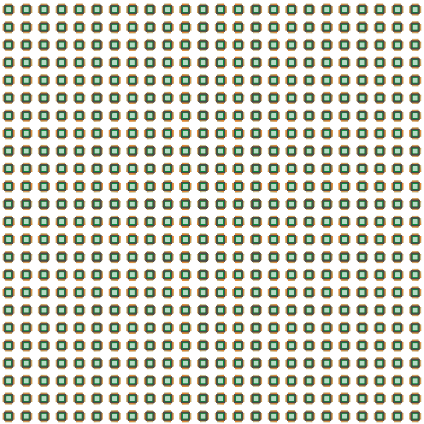
Optimal Cost Approximation as Piecewise-Affine Function



Parallelization opportunity



Master process



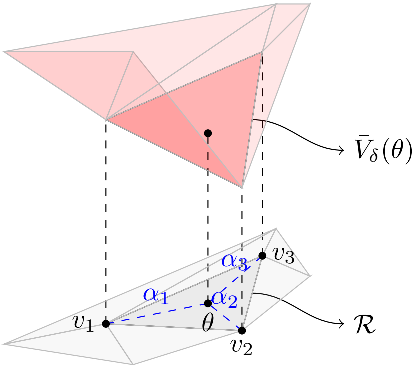
Slave processes

Algorithm 2 Proposed computation of f_δ.

Create empty tree with open leaf Θ as root
Triangulate Θ (`delaunay`)
while any non-leaf node exists **do**
    $\mathcal{R} \leftarrow$ most recently added node
    if MICP (1) infeasible for $\theta = c^{\mathcal{R}}$ **then**
        STOP, $(\Theta^*)^c \cap \Theta \neq \emptyset$
    else
        $\hat{\delta} \leftarrow$ solve (3)
        if (3) is infeasible **then**
            Split $\mathcal{R}$ in half along longest edge
        else
            Replace with leaf $(\mathcal{R}, \hat{\delta})$

Explicit Implementation

$$\hat{x} = \sum_{j=1}^{|\mathcal{V}(\mathcal{R})|} \alpha_j x_j^* \text{ where } \theta = \sum_{j=1}^{|\mathcal{V}(\mathcal{R})|} \alpha_j v_j,\ v_j \in \mathcal{V}(\mathcal{R}).$$



Explicit implementation = binary tree search + $(Ax + b)$ evaluation

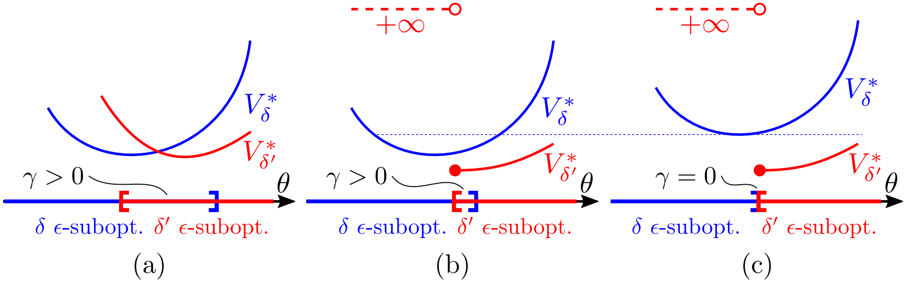
Convergence Properties

Definition 11

The overlap is the largest $\gamma \geq 0$ such that for each $\theta \in \Theta$, $\exists \delta \in \Delta$ which is ϵ-suboptimal in $(\gamma\mathbb{B} + \theta) \setminus \Theta^c$.

Assumption 2

The overlap is positive, i.e. $\gamma > 0$.



Convergence Properties

Theorem 12

Our algorithm converges if and only if the overlap is positive.

Theorem 13

The worst-case partition tree depth τ *is* $\mathcal{O}(p^2 \log(\gamma^{-1}))$.[2]

Theorem 14

The evaluation complexity of f_δ^ϵ *is* $\mathcal{O}(p^4)$, *i.e. polynomial time.*

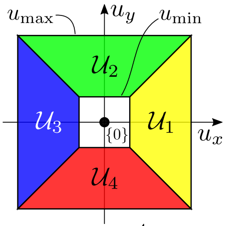
[2] In fact, fixed-commutation cost function gradients also need to be considered, see [2, Definition 6].

Illustrative Example

Clohessy-Wilstshire-Hill dynamics [2]:

$$\texttt{cwh\_xy}: \begin{cases} \ddot{x} = 3\omega_0^2 x + 2\omega_o \dot{y} + u_x + w_x, \\ \ddot{y} = -2\omega_o \dot{x} + u_y + w_u, \end{cases}$$

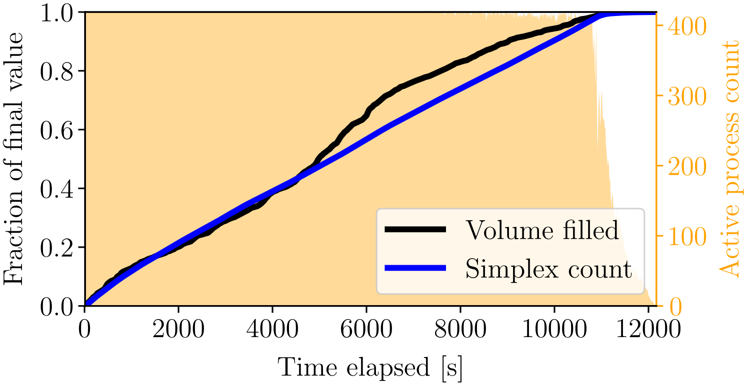
$$\texttt{cwh\_z}: \quad \ddot{z} = -\omega_o^2 z + u_z + w_z,$$



$u = \{0\} \cup \bigcup_{i=1}^4 \mathcal{U}_i$

Example	s_a	ϵ_r	τ	λ	T_{wall} [hr]	T_{cpu} [hr]	M [MB]
cwh_z	0.50	2.00	13	101	0.01	0.09	< 0.01
cwh_z	0.25	1.00	17	978	0.06	0.96	< 0.01
cwh_z	0.10	0.10	20	13500	0.31	7.72	11
cwh_z	0.03	0.05	26	235231	1.91	154.19	202
cwh_z	0.01	0.01	31	3322941	6.37	2516.98	2916
cwh_xy	0.50	2.00	32	30448	0.57	53.44	36
cwh_xy	0.25	1.00	49	884323	3.38	1297.35	1069

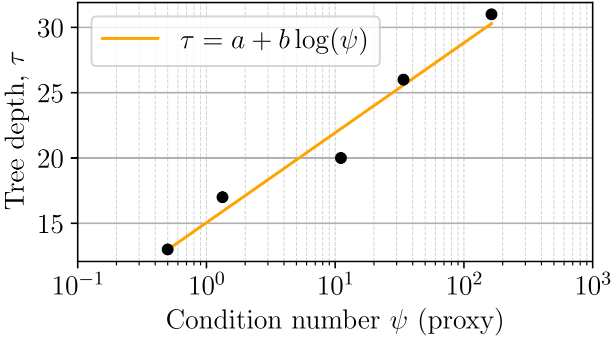
Partitioning Progress Plot



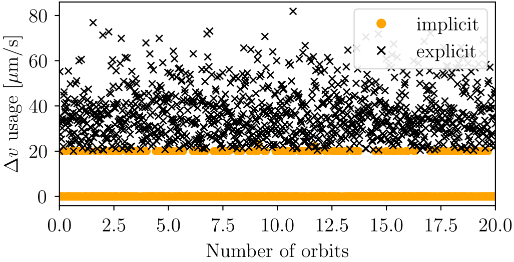
- Python 3.7.2
- CVXPY 1.0.21
- MOSEK 9.0.87
- MPICH 3.2
- CentOS 7
- Up to 1120 processors (20 nodes $\times$ 28 cores/node)
- Cote = 2.4 GHz Intel E5-2680, 20 GB RAM

Platform: **UW Hyak supercomputer**

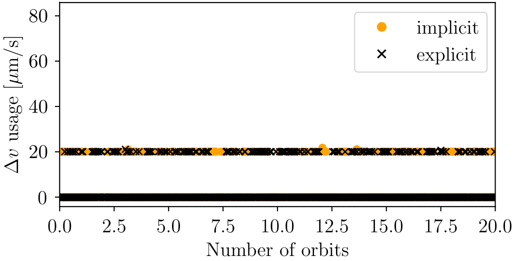
Partition Complexity



Control History



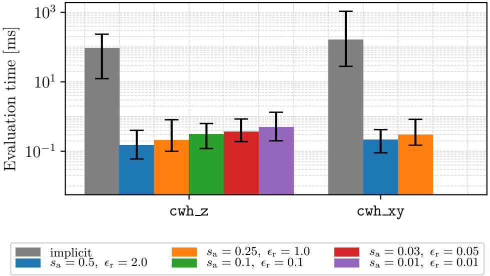
(a) Input 2-norm history for `cwh_z` with $s_{\mathrm{a}} = 0.5$ and $\epsilon_{\mathrm{r}} = 2$.



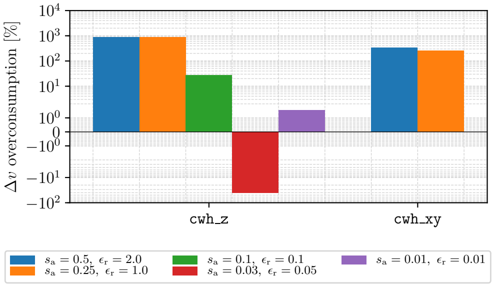
(b) Input 2-norm history for `cwh_z` with $s_{\mathrm{a}} = 0.01$ and $\epsilon_{\mathrm{r}} = 0.01$.

Figure: Comparison of control input histories for a coarse and a refined ϵ-suboptimal partition. By reducing ϵ_{a} and ϵ_{r}, explicit MPC approaches the behavior of implicit MPC.

Control Performance



(a) MPC on-line evaluation time. Bars show the mean while error bars shown the minimum and maximum values.



(b) Overconsumption of fuel with respect to implicit MPC due to ϵ-suboptimality. Implicit MPC uses ≈ 4 mm/s over 20 orbits.

Figure: Comparison of the proposed semi-explicit and explicit implementations to implicit MPC in terms of (a) on-line control input computation time and (b) total fuel consumption over 20 orbits.

Bibliography

- D. Malyuta, B. Açıkmeşe, M. Cacan, and D. S. Bayard, "Partition-based feasible integer solution pre-computation for hybrid model predictive control," in *European Control Conference (accepted)*, p. arXiv:1902.10989, IFAC, jun 2019.
- D. Malyuta and B. Açıkmeşe, "Approximate multiparametric mixed-integer convex programming," *IEEE Control Systems Letters (accepted)*, p. arXiv:1902.10994, 2019.
- D. Dueri, S. V. Raković, and B. Açıkmeşe, "Consistently improving approximations for constrained controllability and reachability," in *2016 European Control Conference (ECC)*, IEEE, jun 2016.