OPTIPLAN: A Matlab Toolbox for Model Predictive Control with Obstacle Avoidance

Filip Janeček * **Martin Klaučo** * **Martin Kalúz** * **Michal Kvasnica** *

* *Slovak University of Technology in Bratislava, Slovakia (e-mail: {filip.janecek, martin.klauco, martin.kaluz, michal.kvasnica}@stuba.sk).*

Abstract: This paper introduces OPTIPLAN - a Matlab-based toolbox for formulating, solving, and simulating model predictive controllers (MPC) with embedded obstacle avoidance functionality. The toolbox offers a simple, yet powerful user interface that allows control researchers and practitioners to set up even complex control problems with just a few lines of code. OPTIPLAN fully automates tedious mathematical and technical details and let the user concentrate on the problem formulation. It features a rich set of tools to perform closed-loop simulations with MPC controllers and to visualize the results in an appealing way. From a theoretical point of view, OPTIPLAN tackles non-convex obstacle avoidance constraints in two ways: either by using binary variables or by resorting to suboptimal, but convex, time-varying constraints.

Keywords: collision avoidance, autonomous vehicles, model predictive control

1. INTRODUCTION

Collision-free control of autonomous vehicles is of a big interest nowadays and plays a major role in ensuring safety of the vehicles and the surrounding environment alike. Many control strategies ensuring collision-free operation of autonomous vehicles have been proposed in the literature with model prediction control (MPC) being the predominant technique, see, e.g. Yoon et al. (2009) and references therein. The popularity of MPC is mainly due to the fact that it can naturally incorporate constraints (including obstacle avoidance constraints) directly into the decision making process. Moreover, since MPC employs predictions of the future behavior of the system and its environment, it is straightforward to include prediction of moving obstacles into the problem, see, e.g., (Carvalho et al., 2015). Finally, MPC is an optimization-based control strategy and the computed control inputs are thus optimal with respect to some performance measure. Due to these advantages MPC has found its way into applications involving steering of autonomous vehicles (Keviczky et al., 2006), autonomous braking (Falcone et al., 2007), improvement of passangers' comfort (Elbanhawi et al., 2016), adaptive cruise control (Corona and De Schutter, 2008), and control of racing cars (Liniger et al., 2015) to name just a few.

Although a plethora of MPC-based control algorithms for collision-free trajectory planning and following exists in the literature, the respective authors rarely provide their software implementation to general public (let alone under free/open-source terms). This leaves interested control researchers and/or practitioners with the difficult task of implementing theoretical algorithms on their own using general-purpose optimization modeling tools, such as YALMIP (Löfberg, 2004), CVX (Grant and Boyd, 2014), or ACADO (Houska et al., 2011). Going the manual way, however, opens doors to erroneous and/or suboptimal formulations.

The objective of OPTIPLAN is to provide a free, open-source tool for MPC-based control of autonomous vehicles with embedded obstacle avoidance capabilities. The toolbox features a simple, yet versatile user interface that allows even non-experts to set up MPC problems using just few lines of code. The toolbox then automatically takes care of tedious mathematical and technical details as to provide an efficient formulation of the underlying optimization problem. In particular, OPTIPLAN allows for two different formulations of obstacle avoidance constraints. The first one, discussed in Section 3.1, employs binary variables to avoid obstacles in an optimal fashion. The downside is that it yields a non-convex mixed-integer problem that can be difficult to solve. As an alternative, the toolbox allows to avoid obstacles in a suboptimal way by using time varying constraints, cf. Section 3.2. The advantage here is that the underlying constraints are convex and allow for a simpler optimization problem to be solved as every sampling instant.

Although all theoretical concepts employed in this paper are well known (see, e.g. Williams (1993) and Frasch et al. (2013)), we believe the presented tool is of interest both to the research community, as well as to control practitioners willing to use MPC in their applications. OPTIPLAN exposes a powerful functionality using a user-friendly interface and hides (and fully automates) technical tasks. This allows its users to concentrate on problem formulation rather than on cumbersome math and/or Matlab programming.

1.1 Notation

We denote by $\mathbb{R}^n$ and $\mathbb{R}^{n\times m}$ the set of real-valued n-dimensional vectors and $n \times m$ matrices, respectively. $\mathbb{N}_a^b$ denotes the set of consecutive integers from a to b, i.e., $\mathbb{N}_a^b = \{a, a+1, \ldots, b\}$ for $a \leq b$. $\|z\|_Q^2 := z^\intercal Q z$ with $z \in \mathbb{R}^n$ and $Q \in \mathbb{R}^{n\times n}$, $Q \succeq 0$ is the weighted squared 2-norm of the vector z.

2. PROBLEM SETUP

OPTIPLAN allows to easily synthesize, solve, and simulate MPC-based feedback laws for autonomous vehicles (robots, quadcopters, UAVs, etc.) that are required to avoid obstacles. The dynamics of the controlled vehicle (which will be referred to as an *agent*), is governed by discrete-time state-update and output equations of the form

$$x_{k+1} = f(x_k, u_k),\ y_k = g(x_k, u_k), \tag{1}$$

where $x \in \mathbb{R}^{n_\mathrm{x}}$ is the agent's state vector, $u \in \mathbb{R}^{n_\mathrm{u}}$ is the vector of control inputs, and $y \in \mathbb{R}^{n_\mathrm{y}}$ is the vector of agent's outputs. Without loss of generality we will assume that the output vector corresponds with the agent's position in the n_y-dimensional Euclidian space.

The task is to devise a feedback controller that manipulates the control inputs u in such a way that:

(1) state, input, and output constraints of the form
$$x \in \mathcal{X},\ u \in \mathcal{U},\ y \in \mathcal{Y}, \tag{2}$$
are enforced;
(2) the agent avoids obstacles $\mathcal{O}_j \subset \mathbb{R}^{n_\mathrm{y}}$, i.e., that $y \notin \mathcal{O}_j$ $\forall j \in \mathbb{N}_1^{n_\mathrm{obs}}$;
(3) the agent tracks a user-specified trajectory y_ref as closely as possible.

OPTIPLAN allows for different types of the dynamics in (1) as long as the system is controllable. In particular, it supports linear time invariant dynamics with $f(x, u) := Ax + Bu$ and $g(x, u) := Cx + Du$, as well as generic nonlinear functions $f : \mathbb{R}^{n_\mathrm{x}} \times \mathbb{R}^{n_\mathrm{u}} \to \mathbb{R}^{n_\mathrm{x}}$ and $g : \mathbb{R}^{n_\mathrm{x}} \times \mathbb{R}^{n_\mathrm{u}} \to \mathbb{R}^{n_\mathrm{y}}$. Moreover, we assume that the constraint sets in (2) are polyhedra of corresponding dimension, and the obstacles $\mathcal{O}_j$ are all polytopes (i.e., bounded polyhedra) in the half-space representation:

$$\mathcal{O}_j = \{y \mid \alpha_{i,j}^\intercal u \leq \beta_{i,j},\ i = 1, \ldots, m_j\}, \quad \forall j \in \mathbb{N}_1^{n_\mathrm{obs}}. \tag{3}$$

Here, m_j is the number of half-spaces that define the j-th obstacle.

Given the input data (the current value of the agent's state $x(t)$, the dynamics in (1), the constraints in (2), and the obstacles in (3)), the MPC problem that OPTIPLAN solves is given by

$$\min_{u_0,\ldots,u_{N-1}} \quad \sum_{k=0}^{N-1} \Big(||y_k - y_{\mathrm{ref},k}||_{Q_\mathrm{y}}^2 + ||u_k - u_{\mathrm{ref},k}||_{Q_\mathrm{u}}^2 \Big) \tag{4a}$$
$$\text{s.t.} \quad x_{k+1} = f(x_k, u_k), \tag{4b}$$
$$y_k = g(x_k, u_k), \tag{4c}$$
$$x_k \in \mathcal{X},\ u_k \in \mathcal{U},\ y_k \in \mathcal{Y}, \tag{4d}$$
$$y_k \notin \mathcal{O}_j,\ \forall j \in \mathbb{N}_1^{n_\mathrm{obs}}, \tag{4e}$$
$$x_0 = x(t), \tag{4f}$$

with (4b)−(4e) imposed for $k = 0, \ldots, N-1$. In (4a), $y_{\mathrm{ref},k}$ and $u_{\mathrm{ref},k}$ are the (possibly time-varying) references for the agent's outputs and inputs to be followed, respectively. In case of no preview of future references being available, $y_{\mathrm{ref},k} = y_\mathrm{ref}$ and $u_{\mathrm{ref},k} = u_\mathrm{ref}$ $\forall k \in \mathbb{N}_0^{N-1}$ is assumed. Moreover, Q_y and Q_u are weighting matrices used to tune the performance. The optimization is performed with respect to $u_0, \ldots, u_{N-1}$. Then, in the spirit of a receding horizon implementation, only the first optimized input, i.e., $u_0^\star$ is implemented to the system in (1) and the whole procedure is repeated at a subsequent time instant for a new value of the initial condition in (4f).

If the obstacle avoidance constraints (4e) are disregarded, the optimization problem (4) becomes a "standard" MPC problem that can be readily formulated with off-the-shelf tools such as YALMIP, CVX, or ACADO, and solved using a plethora of free or commercial solvers such as GUROBI, CPLEX, `quadprog`, `fmincon`, depending on the type of the dynamics in (4b) and (4c), cf. (1).

However, with the constraint (4e) in place, the task becomes more challenging because, even though the obstacles $\mathcal{O}_j$ are assumed to be convex, the constraint $y \notin \mathcal{O}_j$ is non-convex. Although a manual handling of such a non-convex constraint is possible, it is cumbersome, error-prone and, if not done in an optimal fashion, can negatively impact the complexity and thus the runtime of the optimization. The underlying objective of OPTIPLAN is therefore to automate the task of formulating and solving non-convex MPC problems as in (4) with a minimal intervention from the user. The mathematical fundaments of two different ways of formulating the obstacle avoidance constraints (4e) are presented in the next section.

3. TACKLING OBSTACLE AVOIDANCE CONSTRAINTS

In this section, we review to methods of formulating obstacle avoidance constraints in (4e). The first method is based on using binary variables and thus results in (4) being a mixed-integer optimization problem. Although such problems are non-convex and NP-hard, efficient off-the-shelf solvers can tackle them for at least a modest number of obstacles. On the other hand, the mixed-integer formulation provides an optimal way of avoiding obstacles.

The second method avoids binary variables by heuristically choosing the direction from which to avoid the obstacle (either from the left or from the right), followed by employing time-varying output constraints. Although this leads to just a suboptimal trajectory, the advantage is that the constraints remain convex.

3.1 Optimal Obstacle Avoidance

Consider one polytopic obstacle $\mathcal{O}$ as in (3) and introduce binary variables $\delta_i \in \{0,\ 1\}$ for $i = 1, \ldots, m$ (here, m denotes the number of defining half-spaces of the obstacle). Introduce the implication

$$[\delta_i = 1] \implies [\alpha_i^\intercal y \geq \beta_i + \epsilon], \tag{5}$$

i.e., that if $\delta_i = 1$, then the i-th constraint of the obstacle in (3) is violated by at least ϵ. Then $y \notin \mathcal{O}$ is clearly equivalent to

$$\left(\sum_{i=1}^{m} \delta_i\right) \geq 1, \tag{6}$$

i.e., that at least one constraint is violated (and thus the agent does not collide with the obstacle).

The implication in (5) can be recast as a set of linear inequalities by using the following result, due to Williams (1993):

Lemma 3.1. Consider a function $\ell : \mathcal{Z} \subset \mathbb{R}^n \to \mathbb{R}$ and a binary variable $\delta \in \{0,\ 1\}$. Then $[\delta = 1] \implies [\ell(z) \leq 0]$ if and only $\ell(z) \leq M(1-\delta)$ where $M := \max_{z\in\mathcal{Z}} \ell(z)$. □

Applying Lemma 3.1 with $\ell := \beta_i - \alpha_i^\intercal y + \epsilon$ thus transforms (5) into

$$\beta_i - \alpha_i^\intercal y + \epsilon \leq M_i(1-\delta_i), \tag{7}$$

which is a linear constraint in y and δ_i. Since $\mathcal{O}$ is assumed to be a polytope (therefore bounded), M_i can be taken as the maximum of the function $\beta_i - \alpha_i^\intercal y + \epsilon$ over the vertices of $\mathcal{O}$. To guarantee that $y \notin \mathcal{O}$, one therefore needs to impose (7) for $i = 1, \ldots, m$, together with (6).

The original obstacle avoidance constraint (4e) can therefore be replaced by

$$\beta_{i,j} - \alpha_{i,j}^\intercal y_k + \epsilon \leq M_{i,j}(1-\delta_{i,j,k}),\ \forall i \in \mathbb{N}_1^{m_j}, \tag{8a}$$

$$\left(\sum_{i=1}^{m_j} \delta_{i,j,k}\right) \geq 1, \tag{8b}$$

which needs to be imposed for $j = 1, \ldots, n_{\text{obs}}$ (i.e., for each obstacle), and $k = 0, \ldots, N-1$ (i.e., for each step of the prediction horizon). Note that (8) involves a total of $N(\sum_{j=1}^{n_{\text{obs}}} m_j)$ binary variables. OPTIPLAN automatically formulates the constraints in (8) and frees the user from doing so manually, which can be a tedious task if n_{obs} and/or N are large, especially in the view of the following remark:

If the dynamics in (4b)−(4c) is linear, the MPC problem (4) with (4e) replaced by (8) will be a mixed-integer quadratic program (MIQP), which can be solved e.g. by GUROBI or CPLEX. The most suitable solver is automatically detected by OPTIPLAN without user's intervention.

3.2 Suboptimal Obstacle Avoidance

The second way of tackling the non-convex collision avoidance constraints in (4e) is to use time-varying output constraints as proposed in Frasch et al. (2013) coupled with a heuristics that decides from which side of the obstacle the agent should go around (either from the left or from the right). Once the get-around direction is fixed, the output (i.e., position) constraint set $\mathcal{Y}$ is modified as not to colide with the obstacle, i.e., $\mathcal{Y} \cap \mathcal{O}_j = \emptyset\ \forall j \in \mathbb{N}_1^{n_{\text{obs}}}$. Moreover, different constraint sets $\mathcal{Y}_k$ are used at different steps of the prediction horizon, cf. Fig. 1.

The technical realization of this approach is then done in two steps. First, based on the current position of the agent and the position of the obstacles, a heuristic block decides the get-around direction and generates a sequence of constraint sets $\mathcal{Y}_k$ for $k = 0, \ldots, N-1$. Subsequently, the output constraints in (4d) are modified to $y_k \in \mathcal{Y}_k$ and the non-convex collision avoidance constraint (4e) is dropped from the MPC problem. The time-varying constraints $\mathcal{Y}_k$ are then updated at each sampling step.

The advantage of such an approach is that the non-convex constraints (4e) are replaced by a series of convex constraints. This results in shorter runtime of the solution to (4), cf. Section 5. The obvious disadvantage is that the obtained trajectory needs not be optimal and the amount of suboptimality depends on the quality of the heuristically determined get-around directions.

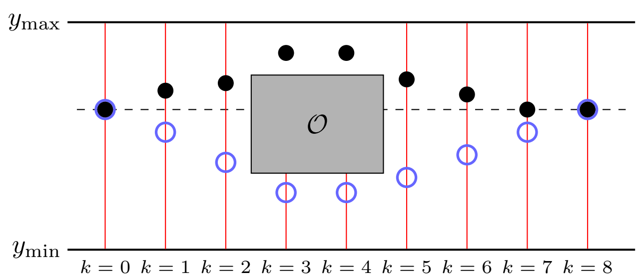

Fig. 1. The idea of obstacle avoidance by using time-varying output constraints. At each step of the prediction horizon the convex output constraints $\mathcal{Y}_k$ (red lines) are updated as not to collide with the obstacle $\mathcal{O}$. Solid circles denote the optimal avoidance based on Section 3.1, empty blue circle denote a (possibly suboptimal) trajectory obtained via time-varying constraints. The dashed line represents the reference to be tracked.

4. OPTIPLAN MINI USER'S GUIDE

In this section we introduce OPTIPLAN's user interface and illustrate how it automates various tasks related to the MPC design, simulation and visualization. By providing a simple, user-friendly interface, the tools allows the engineer to focus on controller tuning rather than on tedious mathematical and technical details.

From a technical point of view, OPTIPLAN is written in Matlab using object-oriented programming. Under the hood, it uses YALMIP to formulate the MPC optimization problem (4) and interfaces with many popular solvers (e.g. GUROBI, CPLEX, MOSEK) to solve such problems. Installation instructions can be found on the project's website at `https://bitbucket.com/kvasnica/optiplan`. To access the tool's interface, the user imports the package into his/her Matlab workspace by

```
import optiplan.*
```

In a way, OPTIPLAN can be viewed at as a high-level language that allows to describe various components of the MPC problem in an easy fashion, to solve the optimization problem, to perform closed-loop simulations, and to visualize its results. From the user point of view, OPTIPLAN exposes four main classes:

- `Agent` - defines the agent's dynamics, physical dimensions, and constraints;
- `Obstacle` - specifies properties of the obstacle(s);
- `Planner` - represents the MPC optimization problem (4) with obstacle avoidance constraints modeled per Section 3;
- `Simulator` - carries on closed-loop simulations and visualizes results.

In what follows, we provide a brief overview of available functionality. Although all examples listed here are in $2D$, OPTIPLAN supports control of agents in arbitrary dimensions.

4.1 The `Agent` *Class*

Instances of this class define the agent's dynamics (cf. (1)), constraints (cf. (2)), physical dimensions, and parameters of the cost function in (4a).

Dynamics: OPTIPLAN supports three types of prediction models in (4b)−(4c). The first one is a linear dynamics (possibly time-varying) of the form $x_{k+1} = A_k x_k + B_k u_k$, $y_k = C_k x_k + D_k u_k$. An agent with linear dynamics can be created by instantiating the `LinearAgent` subclass:

```
agent = LinearAgent('nx', nx, 'nu', nu, ...
           'ny', ny, 'PredictionHorizon', N);
```

Here, the user provides state, input, and output dimensions, respectively, along with the prediction horizon (required if the dynamics and/or constraints are time-varying). The values for the A, B, C, D matrices can then be specified as follows:

```
agent.A.Value = A; agent.B.Value = B;
agent.C.Value = C; agent.D.Value = D;
```

If the dynamics should be time-varying, one sets

```
agent.A.Value = 'parameter'; % also for B,C,D
```

The parametric setting means that the MPC problem in (4) will be formulated for a symbolic value of the matrices and their value only needs to be provided at the time the problem will be solved (see Sections 4.3 and 4.4).

The second type of dynamics is represented by generic nonlinear state-update and output equations in (1). This is achieved by instantiating the `NonlinearAgent` class:

```
agent = NonlinearAgent('nx', nx, 'nu', nu, ...
           'ny', ny, 'PredictionHorizon', N);
```

followed by providing the $f(\cdot,\cdot)$ and $g(\cdot,\cdot)$ functions as function handles, e.g.:

```
agent.StateEq = @(x,u,~) x+(x^2+u);
agent.OutputEq = @(x,u,~) x*u;
```

Note that by using nonlinear dynamics, the MPC problem (4) becomes non-convex and thus difficult to solve to global optimality.

This limitation is abolished, to some extent, by using a *linearizing agent*. Here, the nonlinear dynamics is automatically linearized by OPTIPLAN along the trajectory, resulting in a time-varying linear system that is updated at every sampling instant. Linearized dynamics is defined by

```
agent = LinearizedAgent('nx', nx, 'nu', nu, ...
           'ny', ny, 'PredictionHorizon', N);
```

followed by setting `agent.StateEq` and `agent.OutputEq` as in the case of a nonlinear agent.

Constraints: OPTIPLAN allows to create the constraint sets $\mathcal{X}, \mathcal{U}$, and $\mathcal{Y}$ as hyperboxes by specifying the min/max bounds on corresponding signals. For instance, to set state constraints, the user provides

```
agent.X.Min = x_min;
agent.X.Max = x_max;
```

Similarly, `agent.U.Min`, `agent.U.Max` set input bounds, and `agent.Y.Min`, `agent.Y.Max` specify output bounds. Constraints, too, can be time-varying. In such a case the user sets the corresponding property to `'parameter'`.

Parameters of the cost function (4a)*:* Here, OPTIPLAN lets the user to specify the penalty matrices Q_{y}, Q_{u} that penalize the tracking error in (4a), along with respective reference values. The former is defined by

```
agent.Y.Penalty = Q_y; agent.U.Penalty = Q_u;
```

and the latter by

```
agent.Y.Reference = y_ref;
agent.U.Reference = u_ref;
```

If a reference is not provided, a zero vector is assumed instead. Time-varying reference signals can be specified by setting corresponding values to `'parameter'`.

4.2 The `Obstacle` *Class*

In the initial version of OPTIPLAN described here, the tool supports rectangular obstacles. To create a total of n_{obs} obstacles of the form (3), the user instantiates the `Obstacle` class:

```
obstacles = Obstacle(agent, n_obs);
```

Then, for each obstacle the user can set its size:

```
obstacles(i).Size.Value = [width_i; height_i];
```

along with its position:

```
obstacles(i).Position.Value = [xpos_i; ypos_i];
```

Alternatively, OPTIPLAN allows for moving obstacles by setting `Position.Value = 'parameter'`.

4.3 The `Planner` *Class*

With the agent and the obstacle(s) [1] defined, OPTIPLAN can automatically set up the MPC optimization problem in (4) by instantiating the `Planner` class as follows:

[1] In case of no obstacles, set `obstacles=[]`.

```
planner = Planner(agent, obstacles,...
      'solver', 'gurobi','MixedInteger', flag);
```

Here, we tell OPTIPLAN to use the GUROBI solver [2]. Finally, the `MixedInteger` true/false flag specifies how to formulate the obstacle avoidance constraints in (4e). If set to `true`, then the procedure of Section 3.1 will be used, otherwise the obstacles will be avoided using time-varying constraints as described in Section 3.2.

With the planner in hand, the MPC problem (4) can be solved for a given value of the initial condition x_0 by calling

```
[u, feasible, openloop] = planner.optimize(x0);
```

The first output, `u`, is the optimal feedback control action $u_0^\star$. The second output is a true/false flag indicating whether the optimization problem was feasible. Finally, `openloop` is a structure that contains information about the optimal open-loop trajectories of the states (in `openloop.X`), inputs (in `openloop.U`), and outputs (in `openloop.Y`).

If any properties of the agent and/or of the obstacles were previously declared as parameters, one needs to specify their values before the optimization can commence. For instance, value of the parametric output reference is specified by

```
planner.Parameters.Agent.Y.Rerefence = yref;
```

followed by calling `planner.optimize()`. Here, `yref` can either be a vector (which corresponds to no preview of the output reference in (4a), i.e., $y_{\mathrm{ref},k} = y_{\mathrm{ref}}\ \forall k \in \mathbb{N}_0^{N-1}$), or a $n_{\mathrm{y}} \times N$ matrix whos k-th column specifies $y_{\mathrm{ref},k-1}$. Such an approach allows the user to easily change the parametric values "on-the-fly" without having to re-construct the optimization problem from scratch.

4.4 The `Simulator` *Class*

OPTIPLAN excels at providing a simple, yet powerful way of performing closed-loop simulations under MPC control. To use the simulator, the user first instantiates the `Simulator` class and provides a planner as the input:

```
psim = Simulator(planner);
```

The closed-loop simulation over N_{sim} steps, starting from a given initial condition x_0 is performed by

```
psim.run(x0, Nsim);
```

Various options can be specified as key/value pairs, e.g.

```
psim.run(x0, Nsim, ...
         'Preview', true/false, ...
         'RadarDetector', detector);
```

Here, the `Preview` option controls whether the future input/output references and position of obstacles will be known to the MPC problem in (4). Clearly, the more knowledge does the controller have, the better the control performance gets.

The `RadarDetector` extends OPTIPLAN to the following scenario: obstacles are only avoided if they are detected by the agent's radar. The input to this option is a function handle that takes three inputs: (i) the current agent's position y_0, (ii) the current position of the obstacle(s), and (iii) the obstacle size (in width/height pairs). The function must return an array of true/false values for each obstacle (true if the obstacle is inside of the radar's range, false otherwise). A simple circular radar detector can be created by

```
rad = @(ap,op,os) psim.circularRadar(R,ap,op,os);
```

with `R` specifying the radar's range.

Finally, the `Simulator` class provides various helpers to synthesize time-varying reference trajectories. For example, to generate a circular trajectory of a known radius, call

```
trajectory = psim.circularTrajectory(Nsim, ...
                 'Radius', R, 'Loops', nloops);
```

Alternatively, a polygonic reference passing through given waypoints can be obtained by

```
trajectory = psim.pointwiseTrajectory(Nsim, ...
             waypoints);
```

where the waypoints are stored column-wise.

Once the simulation was ran, the obtained closed-loop profiles over N_{sim} simulation steps can be plotted by calling

```
psim.plot('option1', v1, 'option2', v2, ...);
```

with key/value pairs specifying individual options. See `help Simulator/plot` for details.

5. EXAMPLE

The source code of all examples considered here is available at `https://bitbucket.com/kvasnica/optiplan/wiki/ifac17`. The page also links youtube videos that show individual examples in motion. All computations were run using OPTIPLAN 1.0 in Matlab R2015b on a 2.9 GHz machine with 8 GB of RAM.

Consider an agent that moves in a two-dimensional Euclidian space whose dynamics in each axis is driven, independently, by a frictionless double-integrator dynamics with the x- and y-axis accelerations as the inputs, and x-axis, y-axis positions as the outputs. To quickly generate such an agent, the user calls

```
agent = ...
 LinearAgent.demo2D('PredictionHorizon', N, ...
                    'SamplingTime', Ts);
```

and specifies the prediction horizon (`N=30` was used here) and the sampling time (`Ts=0.25` in our examples). The `demo2D` command automatically sets $-2 \le u \le 2$ as the input constraints, $-20 \le y \le 20$ as the output constraints, $-2 \le v \le 2$ as the constraints on the agent's speed (the state vector is composed of the speed and the position in respective axes), along with $Q_{\mathrm{y}} = Q_{\mathrm{u}} = I_{2\times 2}$ and $u_{\mathrm{ref}} = 0$ in (4a). The output reference is declared

[2] See `https://yalmip.github.io/allsolvers/` for the complete list of supported QP/MIQP/nonlinear solvers.

as parametric and its value will be provided during the closed-loop simulation:

```
agent.Y.Reference = 'parameter';
```

Subsequently, four rectangular obstacles are defined by

```
n_obs = 4;
obs = Obstacle(agent, n_obs);
obs(1).Position.Value = [ 10;   0];
obs(2).Position.Value = [-10;   0];
obs(3).Position.Value = [  0;  10];
obs(4).Position.Value = [  0; -10];
for i=1:4, obs(i).Size.Value = [3; 2]; end
```

Finally, the planner is constructed by

```
planner = Planner(agent, obs, ...
          'solver', 'gurobi', ...
          'MixedInteger', flag)
```

where `flag` will be true/false depending on the type of obstacle avoidance formulation (cf. Sections 3 and 4.3). Finally, a closed-loop simulation is executed by

```
psim = Simulator(planner);
psim.Parameters.Agent.Y.Reference = yref;
psim.run(x0);
```

where at this point we will assume a circular trajectory centered at the origin with a radius of 10, generated by

```
yref = psim.circularTrajectory(Nsim, ...
        'Radius', 10, 'Loops', 2);
```

with `Nsim=350` as the number of simulation steps. At the beginning of the simulation the agent is positioned at the origin, i.e., `x0 = zeros(4, 1)`.

The simulation results as generated by the `psim.plot()` command are depicted in Fig. 2 for the mixed-integer formulation of obstacle avoidance constraints of Section 3.1 (this corresponds to `flag=true` in the construction of the planner). The green square represents agent, red squares are obstacles and empty small black squares are predicted positions of the agent. As the visual representation of time-varying constraints formulation (enabled by `flag=false`) is very similar to mixed-integer approach, it is not presented in this paper. These two approaches differ in the induced computational load. The non-convex mixed-integer formulation of Section 3.1 required a total of 31.9 seconds to run the whole simulation with 350 steps (i.e., 0.091 seconds per step), while the convex formulation using time-varying constraints only took 4.9 seconds (or 0.014 seconds per step). The price to be paid is a deteriorated tracking quality, which is about 6 % worse compared to the mixed-integer formulation.

ACKNOWLEDGEMENTS

Authors gratefully acknowledge the contribution of the Scientific Grant Agency of the Slovak Republic under grant 1/0403/15 and the contribution of the Slovak Research and Development Agency under the project APVV-15-0007. Martin Klaučo also aknowleges the support of an internal STU grant for Young Researchers.

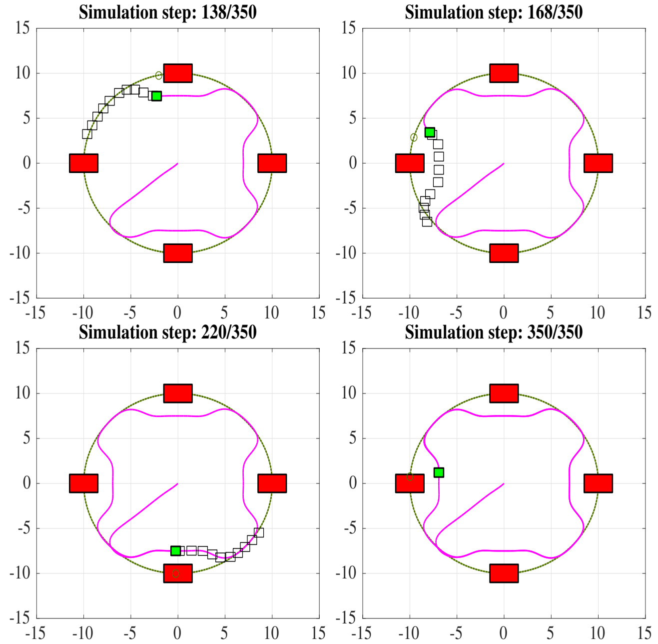

Fig. 2. Collision avoidance with static obstacles using the mixed-integer formulation. Avoiding the obstacles from the left is the optimal decision.

REFERENCES

Carvalho, A., Lefvre, S., Schildbach, G., Kong, J., and Borrelli, F. (2015). Automated driving: The role of forecasts and uncertaintya control perspective. *European Journal of Control*, 24, 14 – 32.

Corona, D. and De Schutter, B. (2008). Adaptive cruise control for a SMART car: A comparison benchmark for MPC-PWA control methods. *IEEE Transactions on Control Systems Technology*, 16(2), 365–372.

Elbanhawi, M., Simic, M., and Jazar, R. (2016). The effect of receding horizon pure pursuit control on passenger comfort in autonomous vehicles. In *Intelligent Interactive Multimedia Systems and Services*, 335–345.

Falcone, P., Tseng, H.E., Asgari, J., Borrelli, F., and Hrovat, D. (2007). Integrated braking and steering model predictive control approach in autonomous vehicles. *5th IFAC Symposium on Advances in Automotive Control*, 40(10), 273 – 278.

Frasch, J., Gray, A., Zanon, M., Ferreau, H., Sager, S., Borrelli, F., and Diehl, M. (2013). An auto-generated nonlinear mpc algorithm for real-time obstacle avoidance of ground vehicles. In *Control Conference (ECC), 2013 European*, 4136–4141.

Grant, M. and Boyd, S. (2014). CVX: Matlab software for disciplined convex programming, version 2.1. `http://cvxr.com/cvx`.

Houska, B., Ferreau, H., and Diehl, M. (2011). Acado toolkitan open-source framework for automatic control and dynamic optimization. *Optimal Control Applications and Methods*, 32(3), 298–312.

Keviczky, T., Falcone, P., Borrelli, F., Asgari, J., and Hrovat, D. (2006). Predictive control approach to autonomous vehicle steering. In *2006 American Control Conference*, 6 pp.–.

Liniger, A., Domahidi, A., and Morari, M. (2015). Optimization-based autonomous racing of 1: 43 scale RC cars. *Optimal Control Applications and Methods*, 36(5), 628–647.

Löfberg, J. (2004). YALMIP. Available from `http://users.isy.liu.se/johanl/yalmip/`.

Williams, H. (1993). *Model Building in Mathematical Programming*. John Wiley & Sons, Third Edition.

Yoon, Y., Shin, J., Kim, J., Park, Y., and Sastry, S. (2009). Model-predictive active steering and obstacle avoidance for autonomous ground vehicles. *Control Engineering Practice*, 17(7), 741–750.