2017 21st International Conference on Process Control (PC)
June 6–9, 2017, Štrbské Pleso, Slovakia

Trajectory Planning and Following for UAVs with Nonlinear Dynamics

Filip Janeček, Martin Klaučo, and Michal Kvasnica
Institute of Information Engineering, Automation, and Mathematics
Faculty of Chemical and Food Technology
Slovak University of Technology in Bratislava
Radlinského 9, 812 37 Bratislava, Slovakia
E-mail:{filip.janecek,martin.klauco,michal.kvasnica}@stuba.sk

***Abstract*—In this paper, we introduce a Matlab-based toolbox called OPTIPLAN, which is intended to formulate, solve and simulate problems of obstacle avoidance based on model predictive control (MPC). The main goal of the toolbox is that it allows the users to simply set up even complex control problems without loss in efficiency only in few lines of code. Slow mathematical and technical details are fully automated allowing researchers to focus on problem formulation. It can easily perform MPC based closed-loop simulations followed by fetching visualizations of the results. From the theoretical point of view, non-convex obstacle avoidance constraints are tackled in two ways in OPTIPLAN: either by solving mixed-integer program using binary variables, or using time-varying constraints, which leads to a suboptimal solution, but the problem remains convex.**

I. Introduction

Collision avoidance and control of autonomous vehicles are of a big interest nowadays. The development of suitable control strategies belongs to one of the most important parts of the overall design of autonomous vehicles, which are capable of avoiding obstacles. Numerous control strategies are developed in this area of research. A widely used approach is the model predictive control (MPC) strategy. The MPC allows for easy incorporation of various constraints, prediction of obstacles positions etc. [1], [2]. The MPC control strategy is also used in lower level control tasks, like optimal steering of the autonomous vehicle [3], breaking [4], improvement of passengers' comfort [5], control of racing cars [6] or adaptive cruise control [7].

In the literature, many tools for solving collision avoidance and trajectory planning and following exists, but only a few of them are available to general public, which leads to the necessity of self-implementing of theoretical algorithms, which can cause many problems such as suboptimal formulations etc.

The implementation of the control algorithm often boils down to a design of the MPC, which can be bothersome especially for nonlinear models of the vehicles. Moreover, the formulation of the MPC must be extended by expressions which represent obstacles. None of the aforementioned nor any other scientific works specifically address the issue of reformulating and solving the optimization problem itself. One of the contributions of this paper is to present a toolbox called OPTIPLAN designed specifically to construct and solve optimal control problems related to autonomous vehicles based on user specifications. Furthermore, one of the features of this toolbox is the ability to simulate the behavior of the vehicle under the control authority of the specified controller.

A. Notation

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

II. Problem Statement

We consider an autonomous vehicle (robot, quadcopter, UAV, etc.) which is supposed to follow a reference trajectory while avoiding collisions with obstacles. The vehicle is controlled by an MPC strategy.

OPTIPLAN is designed to easily synthesize, solve and simulate this kind of problems, where general representation of the dynamics of the vehicle (which will be represented by *agent*) is given as discrete-time state-update and output equations of the form

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

where the vector of the agent's states is denoted by $x \in \mathbb{R}^{n_\text{x}}$, the vector of the control inputs is represented by $u \in \mathbb{R}^{n_\text{u}}$, and the vector of the agent's outputs is $y \in \mathbb{R}^{n_\text{y}}$. The outputs represent the agent's position in the n_y-dimensional Euclidian space. The task is to obtain control inputs u by the action of feedback controller in such 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 kept;
2) obstacles $\mathcal{O}_j \subset \mathbb{R}^{n_\text{y}}$ are avoided by the agent, i.e., $y \notin \mathcal{O}_j,\ \forall j \in \mathbb{N}_1^{n_\text{obs}}$;
3) user-specified trajectory y_ref is tracked by the agent as closely as possible.

As the system is controllable, OPTIPLAN handles different types of dynamics in (1). Linear time-invariant dynamics with $f(x,u) := Ax + Bu$ and $g(x,u) := Cx + Du$ are supported, as well as generic nonlinear functions $f : \mathbb{R}^{n_\text{x}} \times \mathbb{R}^{n_\text{u}} \to \mathbb{R}^{n_\text{x}}$

and $g : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \to \mathbb{R}^{n_y}$. All the obstacles $\mathcal{O}_j$ are polytopes represented by half-spaces:

$$\mathcal{O}_j = \{y \mid \alpha_{i,j}^{\intercal} u \le \beta_{i,j},\ i = 1,\dots,m_j\}, \quad \forall j \in \mathbb{N}_1^{n_{\text{obs}}}, \quad (3)$$

where number of half-spaces defining j-th obstacle is m_j. Moreover, the constraints sets in (2) are assumed as polyhedra of the corresponding dimension.

The MPC problem solved by OPTIPLAN based on given data ($x(t)$ as the current value of the agent's state, the dynamics in (1), the constraints in (2), and the obstacles in (3)) looks as follows:

$$\min_{u_0,\dots,u_{N-1}} \sum_{k=0}^{N-1} \left(\|y_k - y_{\text{ref},k}\|_{Q_y}^2 + \|u_k - u_{\text{ref},k}\|_{Q_u}^2 \right) \quad (4a)$$

$$\begin{aligned} \text{s.t.} \quad & x_{k+1} = f(x_k, u_k), && (4b)\\ & y_k = g(x_k, u_k), && (4c)\\ & x_k \in \mathcal{X},\ u_k \in \mathcal{U},\ y_k \in \mathcal{Y}, && (4d)\\ & y_k \notin \mathcal{O}_j,\ \forall j \in \mathbb{N}_1^{n_{\text{obs}}}, && (4e)\\ & x_0 = x_t && (4f) \end{aligned}$$

where $k = 0,\dots,N-1$ for (4b)–(4e). The references (possibly time-varying) for the agent's inputs and outputs are denoted by $u_{\text{ref},k}$ and $y_{\text{ref},k}$ respectively. If references are constants, $u_{\text{ref},k} = u_{\text{ref}}$ and $y_{\text{ref},k} = y_{\text{ref}}$ $\forall k \in \mathbb{N}_0^{N-1}$. Weight matrices for performance tunning are denoted by Q_u and Q_y. Variables to optimize are $u_0,\dots,u_{N-1}$. Since MPC is implemented in the receding horizon fashion, the only optimized input implemented to the system in (1) is the first one u_0^*. The whole algorithm repeats for a new value of the initial condition in (4f) for every subsequent time instant.

The optimization problem (4) becomes an "ordinary" MPC problem by disregarding the obstacle avoidance constraints (4e), so the tools as YALMIP, CVX or ACADO could be used for the formulation as well as many free or commercial solvers such as `quadprog`, `fmincon`, GUROBI, which depends on the type of the dynamics in (4b) and (4c), cf. (1). But in the case of constraints (4e) are in place, the problem is far more difficult because of their non-convexity, even though we assume the obstacles $\mathcal{O}_j$ to be convex. It is possible to handle such a non-convex constraint, but it is not efficient, could cause mistakes and it can significantly raise the complexity of the problem and the runtime of the optimization (if it is done in a non-optimal manner). Automatic formulation and solution of non-convex MPC problems with minimal user intervention are one of the advantages of OPTIPLAN. It consists two different ways how to formulate the obstacle avoidance constraints (4e). In this paper, we present one of these two methods, which is presented in the next section.

III. Tackling Obstacle Avoidance Constraints

As we have mentioned in the previous section, OPTIPLAN is able to formulate obstacle avoidance constraints in (4e) in two different ways. The first method is based on using binary variables, thus the problem becomes mixed-integer optimization problem (non-convex and NP-hard). Since we focus on a nonlinear model of the agent in this paper, this approach is not convenient, because of its complexity.

The second way to tackle the constraints in (4e) is to choose the direction of avoiding the obstacle (from the right or from the left) by using time-varying output constraints. The advantage is that the constraints remain convex, but the price is that trajectory becomes suboptimal.

A. Convex Formulation of Obstacle Avoidance Constraints

This approach consists of usage of time-varying constraints on outputs (position of the agent) as shown in [8]. The decision from which side of the obstacle the agent should go around (from the right or from the left) is made by heuristics. When this decision is done, the output constraints set $\mathcal{Y}$ is modified in the way that the agent avoids collision with the obstacle, i.e., $\mathcal{Y} \cap \mathcal{O}_j = \emptyset \forall j \in \mathbb{N}_1^{n_{\text{obs}}}$. Moreover, different sets of constraints $\mathcal{Y}_k$ are generated for each step of the prediction horizon, what is shown in the Fig. 1.

Two steps for the technical realization of this approach are needed. First, the decision from which side agent should avoid the obstacle is made by heuristic block based on the current positions of the obstacles and the agent. This generates a sequence of constraint sets $\mathcal{Y}_k$ for $k = 0,\dots,\mathbb{N}-1$. We modify the output constraints in (4d) to $y_k \in \mathcal{Y}_k$ and wipe the non-convex collision avoidance constraint (4e) out of the MPC problem. Then, at each sampling step, the time-varying constraints $\mathcal{Y}_k$ are updated.

The advantage of this method is the series of convex constraints replace the non-convex constraints (4e). The result is a lower complexity of the problem, which is the requirement for usage of the nonlinear model. Obviously, the price for this goal is that it is not guaranteed that the obtained trajectory would be optimal. The quality of heuristics mentioned above is responsible for the amount of suboptimality.

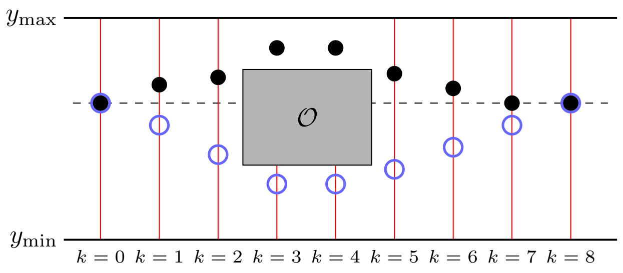

Fig. 1. The idea of obstacle avoidance by using time-varying constraints. Output constraints $\mathcal{Y}_k$ (red lines) are updated at each step of the prediction horizon in such a way to not collide with the obstacle $\mathcal{O}$. Optimal avoidance is denoted by solid circles, possibly suboptimal trajectory obtained via time-varying constraints is denoted by empty blue circles. The reference to be tracked is represented by the dashed line.

IV. OPTIPLAN Mini User's Guide

This section shows OPTIPLAN's user interface and how the tasks as MPC design, simulation and visualization are automated. Thanks to the user-friendly interface, an engineer can focus on controller tuning instead of mathematical and technical details.

Object-oriented programming in Matlab was used to write OPTIPLAN. To formulate the MPC optimization problem (4), it uses YALMIP, and to solve such problems, many popular solvers like GUROBI, CPLEX or MOSEK are supported. If one want to install it, instructions can be found on the project's website at `https://bitbucket.com/kvasnica/optiplan`. Import of the package into Matlab workspace is required to access the tool's interface

```
import optiplan.*
```

OPTIPLAN can be classified as a high-level language to simplify description and solution of the MPC problem, to carry out close-loop simulation and result visualization. OPTIPLAN contains four main classes:

- `Agent` - definition of the dynamics, physical dimensions, and constraints of the agent
- `Obstacle` - specification of the obstacle's properties;
- `Planner` - represents the MPC optimization problem (4) and constraints for obstacle avoidance
- `Simulator` - responsible for closed-loop simulations and visualizations of the results

Here comes a quick summary of available functionality. Even though the examples in this paper are in 2D, control of agents in different dimensions is supported by OPTIPLAN.

A. The `Agent` Class

Definitions of the agent's dynamics (1), constraints (2), physical dimensions, and parameters of the objective function in (4a) are provided by this class.

1) Dynamics: Three types of prediction models in (4b)–(4c) are supported in OPTIPLAN. The linear dynamics (may be also time-varying) of the form $x_{k+1} = A_k x_k + B_k u_k$, $y_k = C_k x_k + D_k u_k$ (here it is important to remind that the output y correspond to the position of the agent). By instantiating the `LinearAgent` subclass we create an agent with linear dynamics:

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

State, input, and output dimensions are provided by the user, accompanied by prediction horizon, which is required in time-varying dynamics and/or constraints. Then specification of matrices A, B, C, D can be done as follows:

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

In the case of time-varying dynamics, one should set

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

The meaning of the parametric setting says the formulation of the MPC problem in (4) will contain a symbolic value of the matrices and we need to provide their real values just at the time of the problem solution.

Generic non-linear state-update and output equations in (1) represent the second type of dynamics. This is achieved by instantiating the `NonlinearAgent` class:

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

The functions $f(\cdot,\cdot)$ and $g(\cdot,\cdot)$ are provided as function handles:

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

It is important to mention that the MPC problem (4) becomes non-convex by using nonlinear dynamics, so difficult to solve to global optimality.

By using the *linearizing agent*, it is possible to decrease this limitation, to some range. It means that OPTIPLAN automatically linearizes the nonlinear dynamics along the trajectory. This results in a time-varying linear system which is updated in every sampling instant. The definition of linearized dynamics looks like:

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

where `agent.StateEq` and `agent.OutputEq` are set in the same way as for the nonlinear agent.

2) Constraints: In OPTIPLAN, it is allowed to construct constraint sets $\mathcal{X}$, $\mathcal{U}$ and $\mathcal{Y}$ representing min/max bounds on corresponding signals as hyperboxes. State constraints are set as follows:

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

Input bounds `agent.U.Min`, `agent.U.Max`, and output bounds `agent.Y.Min`, `agent.Y.Max` are specified similarly. The constraints can be also time-varying by setting the corresponding property to `'parameter'`.

3) Parameters of the objective function (4a): The penalty matrices Q_y, Q_u penalizing the tracking error in (4a), and also the values of the respective reference can be specified by the user as follows:

```
agent.Y.Penalty = Q_y; agent.U.Penalty = Q_u;
agent.Y.Reference = y_ref;
agent.U.Reference = u_ref;
```

In the case of no reference provided, OPTIPLAN assumes a zero vector instead. Also, time-varying reference can be provided by setting the corresponding values to `'parameter'`.

B. The `Obstacle` Class

For now, OPTIPLAN supports rectangular obstacles. The user needs to instantiate the `Obstacle Class` to create n_obs obstacles of the form (3).

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

Each obstacle can be provided by its size:

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

along with its position:

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

OPTIPLAN also allows to create moving obstacles by setting `Position.Value = 'parameter'`.

C. *The* `Planner` *Class*

OPTIPLAN is able to set up the MPC optimization problem in (4) automatically, with the agent and obstacle(s)[1] defined, by instantiating the `Planner` class:

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

In this case, OPTIPLAN is told to use GUROBI solver[2]. The `MixedInteger` flag can acquire values true/false specifying the formulation of the obstacle avoidance constraints in (4e). In this paper, we always use value `false`, which refers to time-varying constraints described in Section III-A.

The MPC problem (4) can be solved for specified value x_0 representing the initial condition, if the planner is created, by calling:

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

where output `u` stands for optimal feedback control action u_0^*. The output `feasible` is a true/false flag which talks about the feasibility of the optimization problem. And last output, `openloop`, hold for the structure with information about the optimal open-loop trajectories of the states (`openloop.X`), inputs (`openloop.U`), and outputs (`openloop.Y`).

Now is the time when all the properties of the agent and/or of the obstacles previously specified as parameters have to be provided with their values, just before the optimization can start. For example, to parametrically defined output reference we associate its value by

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

and then call `planner.optimize()`. The value `yref` can represent a vector (in the case of no preview of the output reference in (4a), i.e., $y_{\text{ref},k} = y_{\text{ref}} \forall k \in \mathbb{N}_0^{N-1}$), or $n_y \times N$ matrix whose k-th column corresponds to $y_{\text{ref},k-1}$. This is an advantage in case that user wants to change the parametric values "on-the-fly", so it is not needed to re-build the optimization problem from the beginning.

D. *The* `Simulator` *Class*

The real power of OPTIPLAN is the ability to simply, still powerfully perform closed-loop simulations under MPC control. First, it needs to be instantiated the `Simulator` class with the planner as the input:

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

To perform the closed-loop simulation over N_{sim} steps, starting from x_0 as the initial condition, user call

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

Key/value pairs are used to specify various options

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

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

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

Knowledge of the future input/output references and obstacle's position by the MPC problem in (4) is controlled by `Preview` option. It is obvious that better control comes with more knowledge.

OPTIPLAN is extended with the following scenario by the `RadarDetector`: agent avoids obstacles only if they are detected by its radar. This function handle should be provided by three inputs: (i) the current position of the agent y_0, (ii) the current obstacle's position, and (iii) the size of the obstacle. The return from the function for each obstacle is an array of true (if the obstacle is inside of the radar's range) or false (if the obstacle is outside of the radar's range) values. To create a simple circular radar, one should call

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

where `R` specifies the range of the radar.

Also, various helpers to synthesize time-varying reference trajectories are provided by the `Simulator` class. For example, the circular trajectory of a known radius can be generated by calling

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

Alternatively, we can obtain a polygonic trajectory passing through given waypoints by

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

where the waypoints are stored column-wise.

The closed-loop profiles are obtained when the simulation is completely done, and they can be plotted over N_{sim} simulation steps calling

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

where various key/value pair options can be provided. See `help Simulator/plot` for details.

V. Examples

In this section we present the results obtained by OPTIPLAN for two different scenarios:

1) multiple static obstacles;
2) multiple moving obstacles with radar.

where in each scenario we compare solutions for non-linear dynamics and automatically linearized dynamics along the trajectory. To illustrate the examples, we used `Simulator.plot()` method directly. The nomenclature used for illustrations is defined in Table I. All example source codes are available at https://bitbucket.com/kvasnica/optiplan/wiki/pc17. For all calculations was used OPTIPLAN 1.1 in Matlab R2016b on a 2.9 GHz machine with 8 GB of RAM.

We consider an agent moving in two-dimensional Euclidian space. Its dynamics are driven independently in each axis by double-integrator dynamics with exponential friction acting

TABLE I
SYMBOLS USED IN SIMULATION PICTURES

symbol		meaning
🟩	*green filled square*	agent
□	*red empty square*	undetected obstacle
🟥	*red filled square*	detected obstacle
○	*red circle*	radar detecting obstacles
···	*thin green dotted line*	reference trajectory
—	*purple line*	actual trajectory
□	*thin black squares*	predicted positions of the agent
▭	*thick black rectangle*	time-varying constraints

against the direction of the motion of the agent. This is the cause of the nonlinear dynamics.

$$\dot{x}_1 = v_1 \tag{5a}$$
$$\dot{x}_2 = v_2 \tag{5b}$$
$$\dot{v}_1 = 1/\mathrm{m}\ (u_1 - \mathrm{k}v_1^2) \tag{5c}$$
$$\dot{v}_2 = 1/\mathrm{m}\ (u_2 - \mathrm{k}v_2^2) \tag{5d}$$

Here, $x_1,\ x_2$ are the positions in 2D Euclidian space, $v_2,\ v_2$ are speeds and $u_1,\ u_2$ are manipulated forces for x- and y-axis respectively, $\mathrm{m} = 1500\,\mathrm{kg}$ is the mass of the vehicle and $\mathrm{k} = 0.0474\,\mathrm{kg\,m^{-1}}$ is the coefficient of friction. Such a dynamics are defined by

```
% continuous state-update equation
f = @(x, u) [x(3);x(4) ...
             (u(1) - k*x(1)^2)/m; ...
             (u(2) - k*x(2)^2)/m];
% output equation
g = @(x, u) x(1:2);
```

and discretized via forward Euler by

```
state_eq = @(x, u, ~) x+Ts*f(x, u);
output_eq = @(x, u, ~) g(x, u);
```

where `Ts = 1` is the sampling time. To construct nonlinear[3] agent with the dynamics described above, user call

```
agent = NonlinearAgent('nx',4,'nu',2,'ny',2,...
        'PredictionHorizon', 15)
agent.StateEq = state_eq;
agent.OutputEq = output_eq;
```

where `agent.StateEq` is discrete state update equation and `agent.OutputEq` is output equation. Sampling time `Ts = 1`. The input (acceleration) constraints were set to $-310 \leq u \leq 310$, speed constraints to $-10 \leq v \leq 10$, and since we used only time-varying constraints for outputs (position) , these were set to `'parameter'`. The state vector is composed of the position and the speed of the agent. Output weight matrix $Q_\mathrm{y} = 1000 \times Q_\mathrm{u} = I_{2\times 2}$, and $u_\mathrm{ref} = 0$ in (4a). The value of the output reference will be provided during the closed-loop simulation, so its value is set as parametric:

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

The settings described above was the same for both scenarios.

[3]In the case of Linearized Agent, user calls `LinearizedAgent` instead of `NonlinearAgent`.

A. Multiple Static Obstacles

In this scenario, we define four rectangular static obstacles by:

```
n_obs = 4;
obs = Obstacle(agent, n_obs);
obs(1).Position.Value = [-15;   0];
obs(2).Position.Value = [  0; -15];
obs(3).Position.Value = [  0;  15];
obs(4).Position.Value = [ 15;   0];
for i=1:4, obs(i).Size.Value = [3; 3]; end
```

When done, construction of planner follows

```
planner = Planner(agent, obs, ...
          'solver', 'fmincon', ...
          'MixedInteger', false)
```

where `fmincon` is used as solver for nonlinear[4] dynamics. Then the closed-loop simulation can be run by

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

where trajectory to be followed needs to be specified. We assume trajectory as a square of the edge of 30, centered at the origin, generated by

```
yref = psim.pointwiseTrajectory(Nsim,...
       [-15 15 15 -15;-15 -15 15 15],...
       'Sampling',true,'Loops',1);
```

with number of simulation steps `Nsim = 250`. Option `'Sampling'` defines whether trajectory should be made only of specified waypoints (`false`), or evenly distributed points between them (`true`) (in the case of Linearized Agent, also the first linearization point needs to be provided, so for the states as for the inputs).

```
psim.UserData.Xlin = Xlin;
psim.UserData.Ulin = Ulin;
```

The initial position of the agent for simulation is at the corner of the trajectory, i.e., `x0 = [-15;-15;0;0]`.

The results of simulation generated directly by `psim.plot()` are presented in Fig. 2. Since difference between trajectory for nonlinear and linearized dynamics are too small to show, Fig. 2 stands for both cases. It is visible from the figure, that during simulation, the time-varying constraints changes in such a way, that vehicle avoids obstacles and follows the reference trajectory. However, there is a difference in computational time. While time required to run the whole simulation (250 steps) for nonlinear dynamics was 39.8 seconds (0.16 seconds per step), time for linearized dynamics was only 5.8 seconds (0.02 seconds per step). The difference between two calculated trajectories was about 1%.

B. Multiple Moving Obstacles With Radar

This scenario differ from the previous one in two points - definition of position of the obstacles and usage of radar. First assume that the agent is defined and we are about to define four moving obstacles. This can be done by

[4]For linearized dynamics we used GUROBI solver.

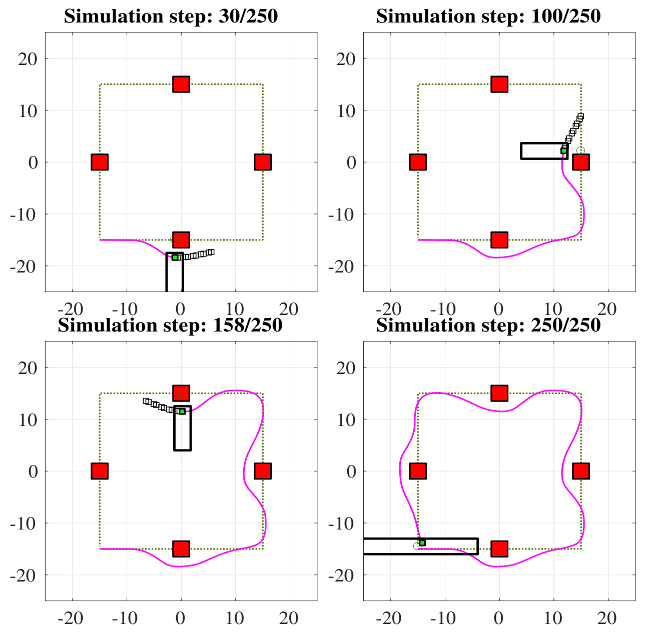

Fig. 2. Collision avoidance with static obstacles. See Table I for the legend.

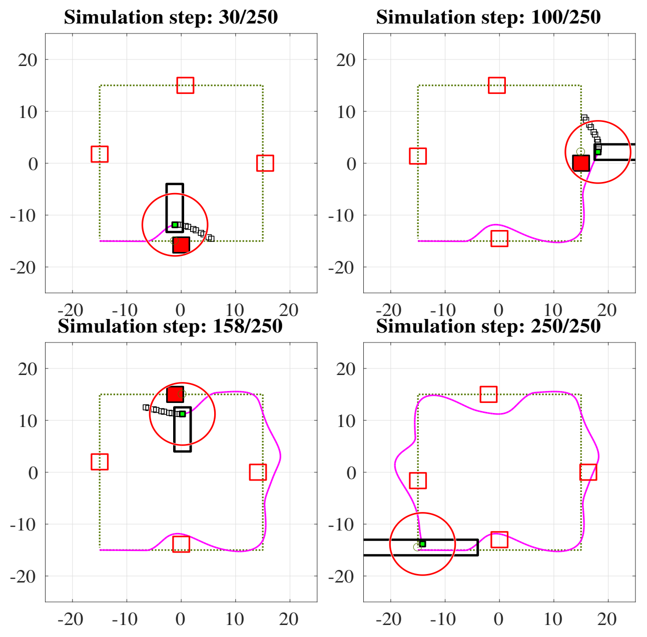

Fig. 3. Collision avoidance with moving obstacles and radar. See Table I for the legend.

```
n_obs = 4;
obs = Obstacle(agent, n_obs);
for i=1:4
  obs(i).Position.Value = 'parameter'
  obs(i).Size.Value = [3; 3];
end
```

Then we construct the planner the same way as for the static obstacles and create the simulator. Before running the simulation, the trajectories for obstacles has to be provided

```
obs{1} = psim.pointwiseTrajectory(Nsim,...
  [-15,-15;-2,2],'Loops',4,'Sampling',true);
obs{2} = psim.pointwiseTrajectory(Nsim,...
  [0,0;-13,-17],'Loops',3,'Sampling',true);
obs{3} = psim.pointwiseTrajectory(Nsim,...
  [17,13;0,0],'Loops',7,'Sampling',true);
obs{4} = psim.pointwiseTrajectory(Nsim,...
  [-2,2;15,15],'Loops',3,'Sampling',true);
for i = 1:4
  psim.Parameters.Obstacles(i).Position.Value =...
      obs{i};
end
```

and the radar needs to be defined

```
R = 6;
radar_detector = @(apos, opos, osize)...
  psim.circularRadar(R,apos,opos,osize);
```

and now the simulation can be run

```
psim.run(x0,Nsim,'RadarDetector',radar_detector)
```

The results of simulation for moving obstacles can be seen in Fig. 3. In this case, the simulation time for nonlinear dynamics was 44.4 seconds for the whole simulation (0.18 seconds per step) and for linearized dynamics 6.9 seconds (0.03 seconds per step). The difference between these two trajectories was about 0.5%.

ACKNOWLEDGMENTS

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 is supported by an internal STU grant for Young Researchers.

REFERENCES

[1] A. Carvalho and S. Lefévre and G. Schildbach and J. Kong and F. Borrelli, "Automated driving: The role of forecasts and uncertaintya control perspective," *European Journal of Control*, vol. 24, pp. 14 – 32, 2015.
[2] A. Carvalho and Y. Gao and A. Gray and H. E. Tseng and F. Borrelli, "Predictive control of an autonomous ground vehicle using an iterative linearization approach," in *16th International IEEE Conference on Intelligent Transportation Systems (ITSC 2013)*, pp. 2335–2340, Oct 2013.
[3] T. Keviczky and P. Falcone and F. Borrelli and J. Asgari and D. Hrovat, "Predictive control approach to autonomous vehicle steering," in *2006 American Control Conference*, pp. 6 pp.–, June 2006.
[4] P. Falcone and H. E. Tseng and J. Asgari and F. Borrelli and D. Hrovat, "Integrated braking and steering model predictive control approach in autonomous vehicles," *5th IFAC Symposium on Advances in Automotive Control*, vol. 40, no. 10, pp. 273 – 278, 2007.
[5] Elbanhawi, M. and Simic, M. and Jazar, R., "The effect of receding horizon pure pursuit control on passenger comfort in autonomous vehicles," in *Intelligent Interactive Multimedia Systems and Services*, pp. 335–345, 2016.
[6] Liniger, A. and Domahidi, A. and Morari, M., "Optimization-based autonomous racing of 1: 43 scale RC cars," *Optimal Control Applications and Methods*, vol. 36, no. 5, pp. 628–647, 2015.
[7] Corona, D. and De Schutter, B., "Adaptive cruise control for a SMART car: A comparison benchmark for MPC-PWA control methods," *IEEE Transactions on Control Systems Technology*, vol. 16, no. 2, pp. 365–372, 2008.
[8] Frasch, J. and Gray, A. and Zanon, M. and Ferreau, H. and Sager, S. and Borrelli, F. and Diehl, M., "An auto-generated nonlinear mpc algorithm for real-time obstacle avoidance of ground vehicles," in *Control Conference (ECC), 2013 European*, pp. 4136–4141, 2013.