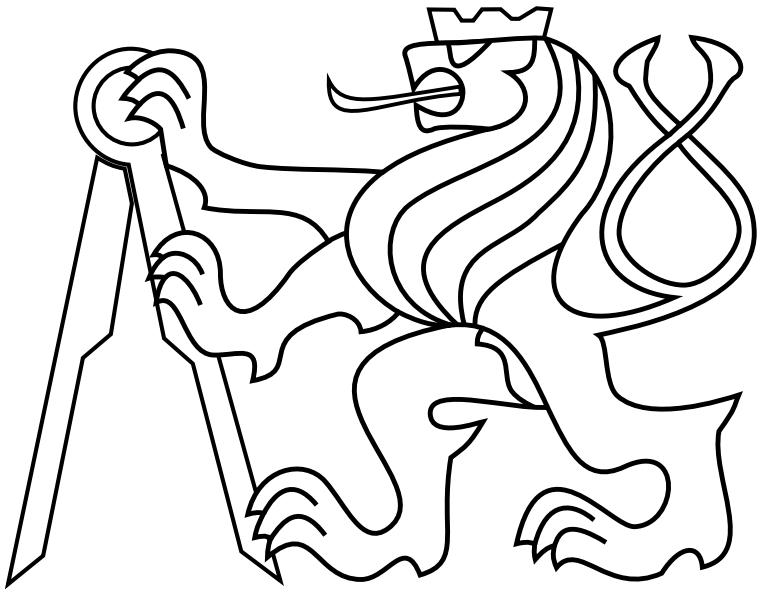
CZECH TECHNICAL UNIVERSITY IN PRAGUE

Faculty of Electrical Engineering

BACHELOR'S THESIS



Tomáš Staruch

Motion Planning for Grasping and Delivering Bricks by Unmanned Aerial Vehicles

Department of Cybernetics

Thesis supervisor: **Ing. Martin Saska, Dr. rer. nat.**

January, 2020

BACHELOR'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Staruch Tomáš** Personal ID number: **465829**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Cybernetics and Robotics**

II. Bachelor's thesis details

Bachelor's thesis title in English:

Motion Planning for Grasping and Delivering Bricks by Unmanned Aerial Vehicles

Bachelor's thesis title in Czech:

Plánování pohybu v úloze sběru a pokládání cihel autonomní helikoptérou

Guidelines:

The goal of the thesis is to design, implement and experimentally verify a motion planning approach for construction of a wall by an unmanned aerial vehicle (UAV). The task is motivated by the second challenge of the MBZIRC 2020 competition, https://www.mbzirc.com/challenge/2020.
1. Design and implement a realistic environment with bricks and wall in a Gazebo simulator under ROS for experimental verification.
2. Design, implement and verify in Gazebo a middle-level motion planning method for grasping of colour bricks. Designing the end-effector, low-level control, and the perception algorithm are not part of the thesis.
3. Design, implement and verify in Gazebo a middle-level motion planning method for placement of the bricks to build the wall. The algorithm will respect actual state of the wall (position and state of neighbouring bricks that are already placed in desired positions).
4. Evaluate reliability of the delivering procedure for different states of the wall and bricks of different size. Provide this information for a high-level planning method, which is solved in parallel to this thesis.

Bibliography / sources:

[1] V Spurny, T Baca, M Saska, R Penicka, T Krajnik, J Thomas, D Thakur, G Loianno and V Kumar. Cooperative Autonomous Search, Grasping and Delivering in a Treasure Hunt Scenario by a Team of UAVs. Accepted in Journal of Field Robotics, 2018.
[2] LaValle, S. M.. Planning Algorithms. Cambridge University Press, Cambridge, U.K., 2006.

Name and workplace of bachelor's thesis supervisor:

Ing. Martin Saska, Dr. rer. nat., Multi-robot Systems, FEE

Name and workplace of second bachelor's thesis supervisor or consultant:

Date of bachelor's thesis assignment: **12.02.2019** Deadline for bachelor thesis submission: **07.01.2020**

Assignment valid until: **20.09.2020**

Ing. Martin Saska, Dr. rer. nat.	doc. Ing. Tomáš Svoboda, Ph.D.	prof. Ing. Pavel Ripka, CSc.
Supervisor's signature	Head of department's signature	Dean's signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

. | Date of assignment receipt

Student's signature

Author statement for undergraduate thesis:

I declare that the presented work was developed independently and that I have listed all source of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague day..... .....

Acknowledgements

I would like to thank Dr. Martin Saska and Ing. Tomáš Báča for their great support during the project. Furthermore, I would like to thank Vít Krátký and other members of Multi-robot System group for valuable advice during the project and especially during the realization of the experiments.

Abstract

The aim of this work is to design, implement and verify methods for motion planning for grasping and placing color bricks by UAV to build a wall. For grasping the bricks is used information from an onboard camera, which is able to detect bricks. For placing the bricks are created curved trajectories by using three different interpolations (B-spline, Catmull-Rom and Hermite). The functionality and precision of methods is verified in GAZEBO simulator, which allows simulating the real conditions of the flight. Results of simulations are compared and analyzed further.

Keywords

UAV, motion planing, building construction

Abstrakt

Cílem této práce je navrhnout,implementovat a ověřit metody plánování pohybu pro sběr a pokládání barevných cihel pomocí autonomních helikoptér s cílem stavby zdi. Ke sbírání cihel jsou použity informace z kamery připevněné na helikoptéru, pomocí které dochází k detekci cihel. Pro pokládání cihel jsou vytvářeny zakřivené trajektorie pomocí třech různých interpolací (B-spline, Catmull-Rom a Hermite). Funkčnost a přesnost metod je ověřena v GAZEBO simulátoru, který umožňuje simulaci reálných podmínek letu. Výsledky simulací jsou porovnány a dále analyzovány.

Klíčová slova

Bezpilotní letoun, plánování pohybu, stavba budov

Contents

List of Figures

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Contents

Bachelor's thesis originated from the Mohamed Bin Zayed International Robotics Challenge (MBZIRC) competition. "MBZIRC provides an ambitious and technologically demanding set of challenges. Robotics is poised to have a transformative impact in a variety of new markets and on various human social aspects. These include robot applications in disaster response, healthcare, domestic tasks, transport, space, manufacturing, and construction."[1] The competition takes place in Abu Dhabi every three years. The competition consists of the accomplish of a few different challenges every year. Every challenge contains the newest problems that should be solved by new original solutions. In the latest competition, challenges involve capturing intruder UAVs inside the arena, constructing large structures by autonomous robots and using UAVs for help in firefighting.

In the past, it was needed months of work with a lot of people to build a building. Construction of buildings is much more frequent these days, therefore it was needed to develop a faster and cheaper way to do that. The latest approach to this problem is the 3D printing of buildings. This approach has many advantages like a reduction of the costs and time and minimizing the pollution of the environment, but this technology still has many limitations, more information in [2]. One of these limitations is the size of the 3D printer which strictly limits the size of the final building. 3D printing system that employs multiple mobile robots printing concurrently a large, single-piece, structure is proposed in [3]. Another modern approach is building structures by autonomous robots. This approach will be more described in the section 1.1.

Challenge two is motivated by the automatic construction of large structures with the use of autonomous robots. Specifically, three unmanned aerial vehicles (UAVs) and

one unmanned ground vehicle (UGV) are available for this task. The goal of the task is to use these vehicles to autonomously collaborate to locate, pick, and assemble a set of brick shaped objects, to construct a digitally pre-specified structure. The task contains four types of bricks, each has a different size and color (red, green, blue and orange). Some bricks (e.g. Blue) may need to be assembled only by the UAVs. Two or more UAVs have to collaborate to be able to carry heavier bricks (e.g. Orange bricks) or UGV can carry these bricks.

This work deal with creating an environment in the Gazebo simulator that would faithfully imitate the real environment for challenge two. This simulator will be used for testing implemented methods that are needed for the challenge (e.g. locating, picking). It also deals with implementing a middle-level motion planning method for grasping bricks and with a middle-level motion planning method for placement bricks. These methods could be used for solving challenge two. Grasping, delivering and dropping objects contained the previous MBZIRC competition too. This article [4] is dealing with the previous competition. We have verified the presented methods by several simulations in Gazebo simulator under Robot Operating System (ROS).

1.1 State of the art

Construction of buildings by autonomous robots is one of the most modern approaches in the construction industry, that's why only a few approaches existing which is used in practice. Research is still largely in progress in this area. Testing of new technologies in simulators or closed testing areas is also in progress. Automation in construction is considerably behind automation in manufacturing, comparison of these industries is discussed in [5].

These papers provide an overview of information about mobile robots. This article ([6]) addresses and classifies the relevant studies in terms of applications, materials, and robotic systems. Similarly, this chapter in the book ([7]) introduces various construction automation concepts that have been developed over the past few decades and presents examples of construction robots that are in current use (as of 2006) and/or in various stages of research and development. In the last article [8] is fabricated a brick wall semi-autonomously in a laboratory environment. Based on this experiment, generic functionalities of the mobile robot and its developed software are presented.

One of the approaches is to use a cable-driven robot for automated construction this approach is described in [9]. This paper [10] presents a simulation-based approach to analyze the technical and economic feasibility of wire robots. `SPIDERobot` is a cable-robot system developed to perform assembly operations and using of this robot in construction is discussed in [11].

Multiple mobile robots are used to fold 2D laser-cut stock into 3D curved structures in [12]. This approach is similar to our task where we use 3 UAV and 1 UGV which is

working in parallel. Articles aren't found describing specifically UAVs which is building the wall or structure. Probably, UAVs used to build the structure weren't used that's why the competition was created. Nowadays UAVs are mostly used to monitoring, e.g. constructions, buildings or bridges which are described in [13].

1.2 Task specification

The goal of the competition is to autonomously build a structure by three UAVs and one UGV. The position of the structure to be built is known before starting the task. This structure is created by four different types of bricks (red, blue, green, orange) and the position of these bricks in structure is pre-specified before starting the task. These bricks have the mass evenly distributed and are located in four different piles, where each pile contains the same type of bricks. Positions of the piles are completely unknown beforehand, therefore, it will be needed to execute mapping of the area to create the map[14].

Only multi-rotor helicopters (marked as UAVs in this work) were used in this work, that's why we focus on them. It is assumed that positions of the UAVs are accurately known, e.g. using GPS or for more accuracy using RTK GPS. Moreover, it is assumed that UAVs are online, which means the UAVs are capable to communicate for sharing their actual planned trajectories and the information about their actual positions with each other. For the purpose of the autonomous solution of the task, it is further assumed that the software for high-level motion planning is existing[15],[16]. This software coordinates three UAVs to construct a structure in obstacle free environment by using the map, positions of individual UAVs and their communication. The software has to plan trajectories for UAVs with a collision avoidance approach and the software uses different software methods to accomplish partial tasks, for example, to pick the brick or to place the brick.

Our aim is to implement a method to grasp the brick and method to place the brick. For the purpose of grasping the bricks, it is assumed that UAV is equipped with a mechanism that can grasp the brick and carry it, e.g. by using electromagnets. This mechanism is attached to UAV so that the mechanism nearly doesn't negatively influence the properties of UAV during the flight. This property must be satisfied while the mechanism is carrying the brick. The other assumption is that the UAVs are equipped with a camera and with software that recognizes the bricks[17]. This software can determine the precise location of the brick and return the coordinates of this brick.

For the purpose of placing the bricks, it is assumed that the precise position of UAV is known. It isn't possible to place the brick on the required location without knowing the current position of UAV. The other assumption is that the mechanism currently carries the brick. This brick is attached to the UAV in its center (on the longer side of the brick in the middle) and the sides of the brick are parallel with sides of UAV (brick is not rotated in yaw axis).

Some of these assumptions aren't necessary, e.g. the attaching of the brick, but their

breach causes the solution of the task more difficult. It will be impossible to build the structure in which be the bricks correctly and precisely placed.

Chapter 2

Realistic environment

The first task of work is to create an environment that would be very similar to the specified environment for challenge two. Arena in challenge two will be approximately 50mx60m and it will contain four randomly located piles of bricks, with each pile consisting of similar-sized bricks. The start position will be specified in the area, where will be placed UAVs and UGV. At last, the size and color of four types of bricks is defined as:

Color	approximate size of bricks	weight of bricks
Red	0.30x0.20x0.20 m	$\leq$ 1kg
Green	0.60x0.20x0.20 m	$\leq$ 1kg
Blue	1.20x0.20x0.20 m	$\leq$ 1.5kg
Orange	1.80x0.20x0.20 m	$\leq$ 2.0kg

Table 2.1: Table of specified dimensions and weights to date 9.11.2019

Existed environment `grass_plane.world` was used, which contains a grass plane having a size of 250x250 m and it has real physics. At first, the size of the grass plane was changed to 50x60 m according to the specifications and the new simulation environment `mbzirc_construct_wall.world` was created. In the next step, four asphalt planes were created and they were located near each corner of the plane. These planes could be located randomly according to the specifications, but we place them to the static locations. The mapping procedure will be necessary at the beginning of the task that is the reason why static locations of planes aren't fault. At last, two asphalt lanes were created in shape of letter L and placed in the middle of the grass plane. This lane indicated the foundations of the wall. For creating each plane was added `visual` section of code to model of the ground plane. This code is described below.

Algorithm 1 Description of code for adding one plane

visual name ▷ name of the plane (wall_place_1)
pose frame = x y z roll pitch yaw **pose** ▷ position of plane center
cast shadows false **cast shadows**
geometry
plane
normal x y z **normal** ▷ normal to plane (0 0 1 is plane in xy)
size x y **size** ▷ value of coordinates is equally divided to both sides from the pose
plane
geometry
material
script
uri location of a material script **uri**
uri location of a texture for material **uri**
name name of material **name** ▷ in our case - vrc/asphalt
script
material
visual

Five bricks of each type were created, where each brick had to be created like a single object in code. It was important to set the same collision and visual size of objects. If these sizes weren't the same, then the brick could penetrate other brick or on the other side, brick could lean on air. In the next step, we had to compute inertia for each brick. If the inertia was set incorrectly, then usually UAV couldn't grasp bricks. Inertia for brick is computed as inertia for block :

$$
\begin{aligned}
i_{xy} &= 0, i_{xz} = 0, i_{yz} = 0 \\
i_{xx} &= 0.083 \cdot mass \cdot (y \cdot y + z \cdot z) \\
i_{yy} &= 0.083 \cdot mass \cdot (x \cdot x + z \cdot z) \\
i_{zz} &= 0.083 \cdot mass \cdot (x \cdot x + y \cdot y)
\end{aligned}
$$

Bricks of the same color were placed on individual asphalt planes after creating objects. Some of these bricks were rotated in coordinates x,y for better testing of grasping.

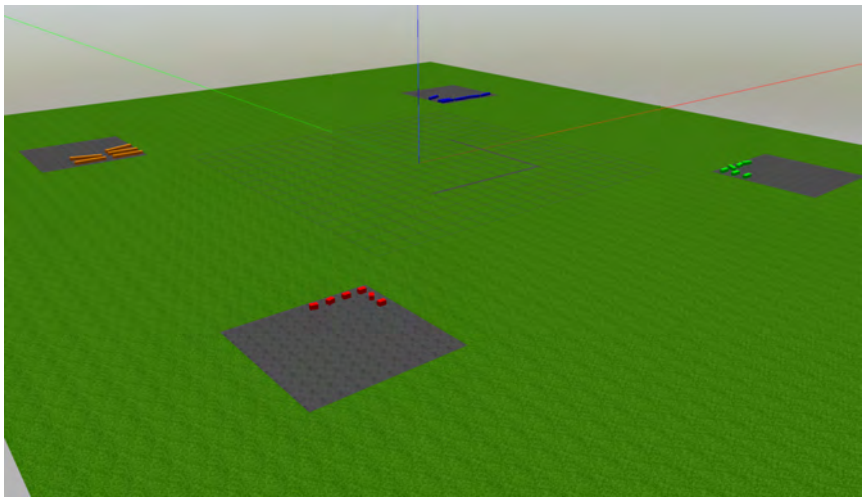


Figure 2.1: Picture of first created world for simulation

This world was uploaded to git for mbzirc competition. The world had to be updated a few times because of changes in the assignment of challenge two. The size and weight of bricks were changed a few times, according to these changes inertia had to be recomputed. The next important change originated from the detection of green bricks. It was difficult to recognize green brick on grass for this reason grass plane was replaced by an asphalt plane. The surface isn't defined in the challenge, but mostly the surface during competitions is similar to asphalt. The last changes included adding more bricks into the existed zones with bricks.

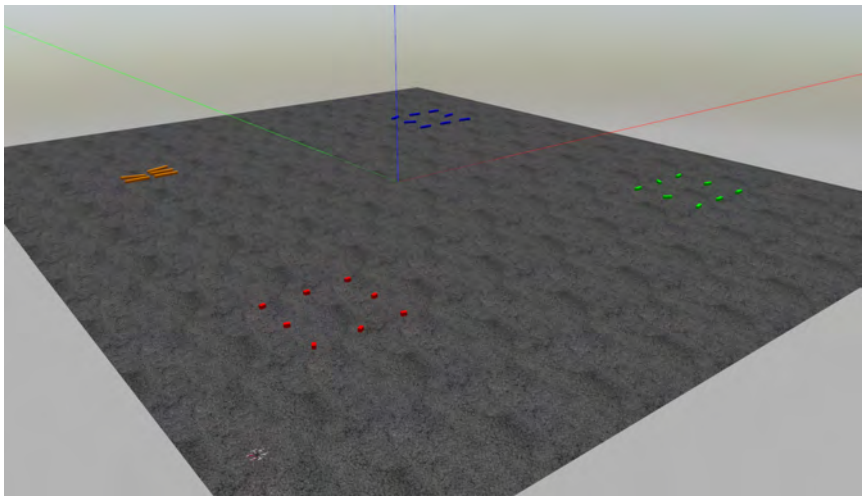


Figure 2.2: Picture of the last created world for simulation, 9.11.2019

Chapter 3

Method for grasping

Contents

The second task of work is to design, implement and experimentally verify a motion planning method for grasping color bricks. A similar problem was solved in the task of the previous mbzirc competition. In this task, It was needed to grasp a magnetic circular object and transport it to the box. The solution of this task is described in [18].

The method for grasping these objects was created in node `visual_servoing`. This node contained two different codes, `landing_object_estimator` and `landing_object_controller`. The `landing_object_estimator` must identify the state of a UAV, by using information from sensors, and the location of the object to be grasped, by information from the camera. The `landing_object_controller` contained state machine which ensured grasping and dropping of the objects. These two codes communicated together constantly for exchanging of information during using of this method.

Robot Operating System (ROS) was updated to a new version since the mentioned node `visual_servoing` was created. For this reason, it was needed to understand working of the node and rewrite it so that this node could be used in new version of ROS. Also, it was needed to modify the node for the detection of bricks (blocks) instead of circle objectives.

3.1 Landing_object_estimator

UAV is equipped with several different sensors, e.g. camera, GPS, RTK GPS, and high sensor. Information from these sensors is fused in terms of accuracy and speed by `Landing_object_estimator` to obtain a reliable position of UAV. Extended Kalman Filter is used for fusing the sensors. However, for this task (grasping the bricks) is required an extremely precise position of UAV. For this purpose is used differential RTK GPS and measurements from this sensor are fused using a Linear Kalman Filter to correct this position.

`Landing_object_estimator` has to get information from the `object_detection` node for correct functionality. This node recognizes objects which the camera has seen and then publishes coordinates of all these objects. Also, this node is able to provide an estimate of the relative distance when flying above an object [19]. `Estimator` creating a map by using this information. This map is changing during the task by these rules: ”

- Objects which have not been seen for more than 5 seconds are deactivated.
- Objects which are deactivated for more than 3 seconds are deleted from the map.
- Measurements from the object detector are paired with objects in the map using a min-distance bipartite graph matching, constrained by the color of the objects.
- Objects located outside of the working area are deleted from the map, and new measurements in such areas are discarded.

”[18]

If UAV fails in grasping the object then the object and 4m radius around the object will be banned on the map for some time. This time could be specified in the config file.

Algorithm 2 Description of function (objectCallback) which process message from camera (creating the map)

```
for over all object do
    load the object
    if Object in Dropout zone then
        discard this object
    end if
    if Object is grasped by drone then      ▷ compared position of all UAVs and object
        discard this object
    end if
    if Object in banned area then                                  ▷ grasping failed
        discard this object
    end if
    function find closest object in map
    if closest object not found then                    ▷ create new object in map
        create new object in map
        create new object for Kalman
        add object to the map
    else                                                ▷ fuse this object with map
        update altitude of object
        set the covariance based on altitude
        if Kalman correction succeeds then
            update position of object
            update time                          ▷ when the object was last seen
        else
            discard this object
        end if
    end if
end for
```

This code describes only the main idea of function. Some parts of the function are not included, e.g. check the content of the received message.

`Estimator` publishes information, which is needed for grasping the object, such as the precise position of UAV or position of the object to be grasped. This information is used by `Landing_object_controller` which can partly manage `Estimator`. For example, the `Estimator` can switch between sensors, from which gets information, depending on the current state of the `Controller`.

3.2 Landing_object_controller

`Landing_object_controller` ensures the execution of the method for grasping the bricks. This code can be managed by 3 different services (start, stop and drop) and on the other hand, this code uses several external services, e.g. controlling the magnet, ban area in the map or switching sensors. For grasping procedure is used service `start 1`, which starts grasping the nearest static object in the map. This procedure is supervised by the state machine described on 3.1.

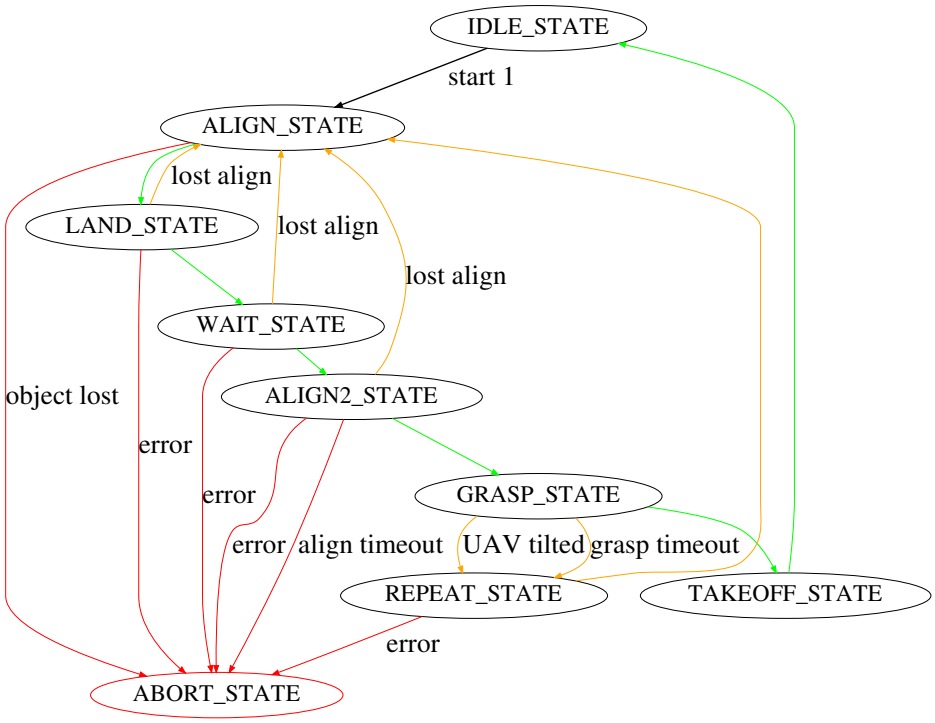


Figure 3.1: State machine for grasping procedure

- `IDLE_STATE` - this state is inactive. The controller is waiting for the service call. For example, after calling service `start 1` the controller find the nearest static object in the map.
- `ALIGN_STATE` - this state aligns a UAV above object. Alignment is eventually ensured by creating a trajectory to align the UAV and the object. The UAV is aligned with the object if the distance between the center of the object and UAV in coordinates x,y is smaller than 1.5 meters. If the object is lost in the map then grasping procedure is abort.
- `LAND_STATE` - in this state, the UAV is slowly descending to height 1.5 meters above the object.

- `WAIT_STATE` - this state checks alignment to the object and occurring of error.
- `ALIGN2_STATE` - this state checks alignment to the object, occurring of error and timeout for aligning and descending.
- `GRASP_STATE` - this state gently descending until the magnet is not connected with the object. The state is checking timeout for grasping and tilt of the UAV. After attaching the object is computed the new weight of UAV and object and some sensors are switched off because they can't work correctly with the attached object. In this article is described the setting of the regulator for precise landing on the object according to wind and changing of the weight. [20]
- `TAKEOFF_STATE` - this state is rising back to the original height.
- `REPEAT_STATE` - this state rises to the height 3.5 meters where the state is switched to aligning. If aligning was achieved 3 times then occurs error and grasping procedure is aborted. In this case, the object is banned for some time.
- `error` - if the grasping object is lost in the map or the UAV lost aligning for 5 times then the error is created.

Parameters such as height for repeat state or number of alignments before the procedure is aborted can be changed in `params.yaml`.

For dropping procedure is used service `drop`. When the service is called, UAV descends to dropping altitude which can be set in `params` and then drops the object. UAV rises to takeoff altitude after releasing the object. This service could be called only in the dropping zone otherwise this service doesn't be executed.

3.3 Upgrade visual_servoing node

Node `visual_servoing` was written in an older version of ROS as mentioned above. This version of ROS was used in the previous mbzirc competition. Since that, ROS released a new version, which contained new libraries with new functions. Simultaneously, some libraries were canceled and replaced by new ones or combined with others. With the new version of ROS, it was remade `uav core node` which is used by the multi-robot system department and contains ordinarily used functions. For this reason, it was needed to replace old libraries by the new libraries including functions which are from these libraries. It was used the ROS forum [21] for finding the correct replacement of an old function (in which library is newly located).

After rewriting libraries, it had to be rewritten `package.xml` which contained dependencies on other nodes or libraries. `package.xml` is closely related to `CMakeLists.txt` which is used to compile the node. Next for the successful compilation of the node, it

was created a new node `mrs_msgs_mbzirc` that contained the same types of messages as `object_detection`. The `object_detection` node didn't exist for the actual competition and the old node couldn't be used because the node detected only circle objects.

During editing codes, it was deleted some parts, which aren't needed for the actual competition. In `Estimator` were deleted parts that solved only moving objects and parts which contained some areas as the dropping zone. In `Controller` were deleted similar parts as in `Estimator`, e.g. it was deleted condition for dropping object only in dropping zone.

Lastly, it were redirected `Subscribers` and `Publishers` in the launch file for correct communication with other nodes. This step could be executed when the compilation was successfully done. Because for precise redirecting was used running ROS and listening all publish and information about them.

3.4 Testing Visual_servoing node

`Visual_servoing` node is first tested in the simulator. It is needed to create a method, which will allow connecting bricks and UAVs in the simulation. Thus, It will simulate a mechanism for attaching the bricks. The method will be replaced by a signal to mechanism in the real environment. The same method was created in the previous competition mbzirc and it was included in a node named `object_movement`.

`Object_movement` is simulating the mechanism as an electromagnet. The node creating a joint between each UAV and object in simulation. Names of these objects and UAVs have to be defined in `params.yaml` and names must faithfully correspond with names in simulation. This code check distance between objects with some periodicity, which can be set in `params`. If the first object is close enough to the center of the second object (again it is a parameter in `params`), then it is possible to connect these objects. However, these objects must be UAV and in our case brick and at the same time the UAV must have set on the magnet.

The node `object_movement` is updated by similar modifications as in the node `visual_servoing`. These changes contain update libraries, modifying dependencies in `package.xml` and `CMakeLists.txt` and launch file. The code included parts, which were used for moving objects. These parts were removed because they weren't needed. It is possible to attach a brick to UAV in simulation after these modifications.

Unfortunately, the testing still can't start running because the node doesn't get information from `object_detection` which is not created for new competition yet. For this reason, it is created the new node named `fake_object_detector` which hands over information about objects to the node without using a camera. By using information from the camera (or fake camera) is UAV guided to the precise position of the brick [22],[23].

Algorithm 3 Description of function (objectCallback) which substitute information from the camera by information from simulator (main function of `fake_object_detector`)

```
function objectCallback(gazebo message with models)
    if not inicialization of simulator then
        return
    end if
    delete old objects in array
    for all objects in simulator do
        if name is "groundplane" then
            continue
        end if
        if name is uav then
            continue
        end if
        create new object
        set type of static object
        copy position from simulation
        push object to array
    end for
end function
```

Created objects in function have same type as objects from `object_detection`. The `object_detection` can be substituted by publishing created array with objects. The array is published with a defined time period (0.05 s) because if the period is much shorter then the `Estimator` crash. The Estimator isn't able to process information from "camera" so fast, because a real camera can't publish information with a shorter period.

After starting the simulation, It is possible to grasp brick, carry it and drop it from the defined altitude at this time. It is created a simple node `basic_trajectory` for easier testing of the overall function of nodes to build the wall and cooperating between each node. This node controls one UAV by a sequence of basic actions specifically the actions are:

- `go_to` for moving UAV in simulator, service of the same name is called. This service is more described 4.3
- `Grasping` for calling service to grasp a brick,
- `Droping` for calling service to drop brick.

This sequence is performing like a queue and the next action starts immediately after the previous action is finished. The queue is represented by the switch statement where each case in the switch is one action. Because of this structure is very easy to extend the queue by copying the case and for `go_to` by copying and changing the position.

Continuity of actions is ensured by subscribing information from other nodes:

- Finishing of `go_to` is detected by information from `odometry/odom_main`, where is actual position of the UAV (GPS). UAV finished the movement if is the difference between required and actual position smaller than 0.1 meters.
- Finishing of `Grasping` is detected by information from state machine in the `Controller`. The state machine is changing `IDLE_STATE` to a different state after calling service, and after finishing this service It will return to `IDLE_STATE`.
- Finishing of `Droping` is detected the same as the finishing of grasping.

This node was used for testing grasping and dropping the bricks by UAV and for simulation of building the wall. If UAV was above the brick then the UAV successfully finished grasping of brick every time. The brick was attached relatively in the center that was needed for the correct dropping of the brick and building of a wall. Unfortunately, relatively small grip error sufficed to generate a much worse error in the placement of brick in the wall. The brick was laying on the other brick which was next to it (therefore the brick laid obliquely). Because of this error, it is needed to design an alternative solution for placing the bricks.

Chapter 4

Method for placement

Contents

The third task of work is to design, implement and experimentally verify a motion planning method for placing of color bricks to the structure. An original idea to solve this problem assumes that a node that will solve this problem get message from a high-level motion planning method that will contain actual state of the wall, the position where to place the brick and position where to move after placing the brick. Trajectory will be planned so that the brick will be placed as accurately as possible according to the actual state of the wall.

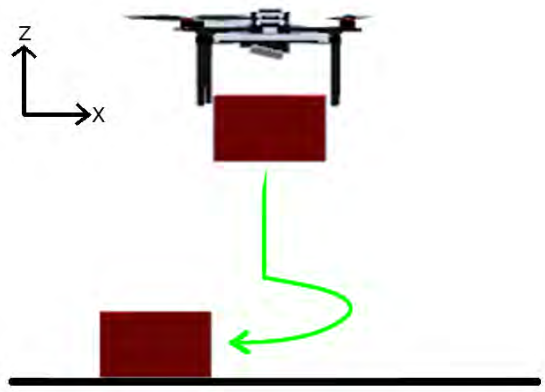
4.1 Types of trajectories for placing the brick

As mentioned above if just carried brick isn't attached directly in the middle then the error can easily originate during placing the brick. The brick will be laying on the other brick which is next to it (so the brick will lay obliquely). Therefore it is needed to design an alternative approach for placing the bricks.

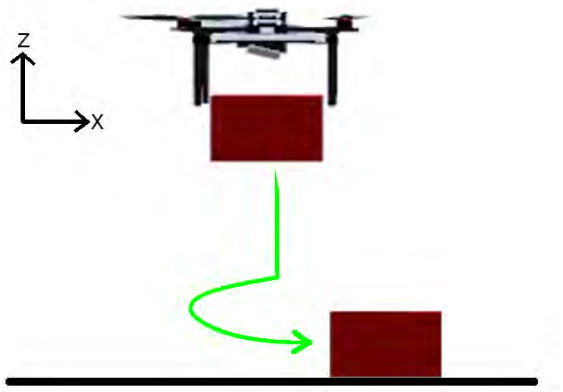
The first approach to this problem was to calculate the deviation of carrying the brick from ideal (directly in the middle and without rotation) by using the visual flow of the onboard camera. The deviation would be used to move or rotate UAV so that the UAV could place the brick as if the brick is attached in the optimal position. Unfortunately, it wasn't clear where the camera will be placed on the UAV. Therefore it isn't guaranteed that the deviation will be always detected correctly. This solution isn't robust enough and,

therefore, an alternative approach to solve the problem is developed which is using other than direct trajectories.

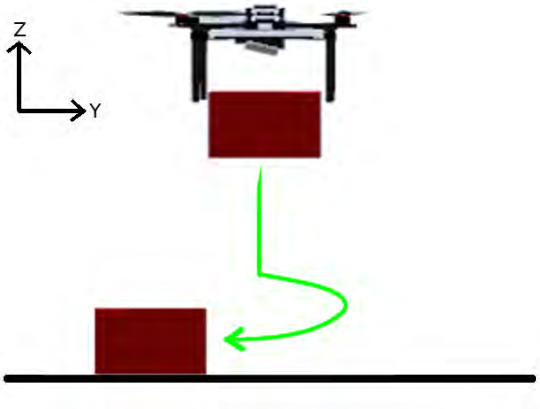
The second and used approach in this task uses different trajectories to place the brick. These trajectories depend on the state of the structure and position where the brick should be placed. The approach is able to eliminate small deviation of carrying the brick and mostly completely eliminate oblique placing of the brick. It is supposed that the structure is built gradually by individual floors otherwise it could happen collision with already placed bricks by using this approach. Five different trajectories are designed which depend on the state of the structure and are described in the figure below 4.1.



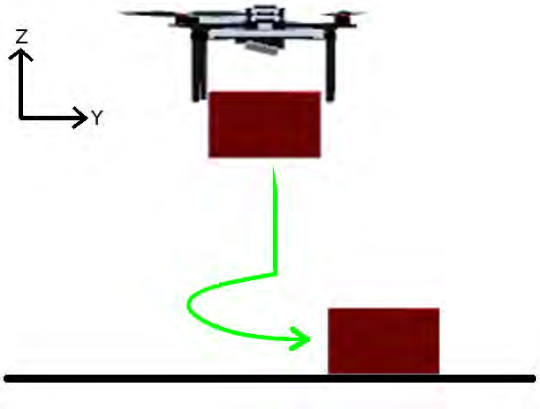
(a) Sketch of second trajectory to place the brick



(b) Sketch of third trajectory to place the brick



(c) Sketch of fourth trajectory to place the brick



(d) Sketch of fifth trajectory to place the brick

Figure 4.1: The figure contains sketches of four types of trajectories used for placing the brick.

It is visible on the figures that these four trajectories are almost the same and the difference between trajectories is only in "curve" (in direction of what axis will be executed

turnover). Accuracy and speed of executing the trajectories depend on the size of the curve and the quantity of points of which is curve created. The last type of trajectory isn't creating the curve and the brick is placed by straight trajectory as is visible on the next figure 4.2.

(a) Sketch of first (straight) trajectory to place the brick without bricks

(b) Sketch of first (straight) trajectory to place the brick with bricks

Figure 4.2: The figure contains sketches of the last type of trajectory used for placing the brick.

This type of trajectory is developed for placing the new brick on the next floor because here isn't possibility of formatting the collision with bricks 4.2b. Or on the other hand, this trajectory could be used to place the brick between two already placed bricks 4.1d. This state of the structure is very dangerous because of easy formatting the error and therefore the state shouldn't ever occur.

Thanks to the update of the assignment of competition was found that the specification of structure (where should be placed each color brick) will be communicated before starting the challenge. A sequence of actions will be planned before the challenge and therefore input message for this method is changed. The input message now contains one of five types of trajectories by which should be placed the brick, position where to place the brick and position where to move after placing the brick.

4.2 Creating of the trajectory for placing the brick

In this section will be described the creating of placing trajectory for brick with the curve, therefore, one of the trajectories from figure 4.1. The creating of a straight trajectory will not be explicitly mentioned because the trajectory is created similarly only the points are in one straight line. The shape of trajectory can be modified in several ways by changing parameters in `config` file. For example, It is possible to modify the width

and height of the curve or number of points from which is curve created. This approach provides a simple modifying of the trajectory and that's why it is possible to search the middle between fast and successful executing of the method.

The creating of the trajectory is divided into two parts because of the request of robustness solution and finishing of each part is checked individually. The first part is created by a straight trajectory for descending. Description of the part is shown in figure 4.3.

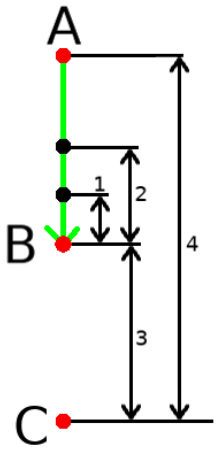


Figure 4.3: Description of straight trajectory (first part of the trajectory)

- Letter `A` is `Start position` for placing the brick
- Letter `B` is `Start spline position` which is the position where the curve is starting to be executed
- Letter `C` is `Place position` which is the position where the brick should be placed
- Number `1` is the quarter distance between letter `A` and `B`
- Number `2` is the half distance between letter `A` and `B`
- Number `3` is the distance between letter `B` and `C` which could be set by parameter `altitude curve offset`
- Number `4` is the distance between letter `A` and `B` which could be set by parameter `altitude offset`
- The approximate trajectory that the UAV will fly through is shown by green color on the figure.

Point `C` is included in the input message and by using this point and parameters (3,4) are calculated points `A` and `B`. If the points `A` and `B` are the same then this part of trajectory is skipped and only the second part is executed. On the other hand, the rest two black points are calculated and together with points `A` and `B` are sent to UAV. These points are sent to the MPC trajectory tracker, which is described below 4.2, in correct order which is from top to bottom. Finishing this part of the trajectory is detected by matching actual position of the UAV and point `B`. If the distance between UAV and the point is close enough then this part was successfully executed.

The second part is created by a curve trajectory for attainment position for placing the brick which is described in figure 4.4.

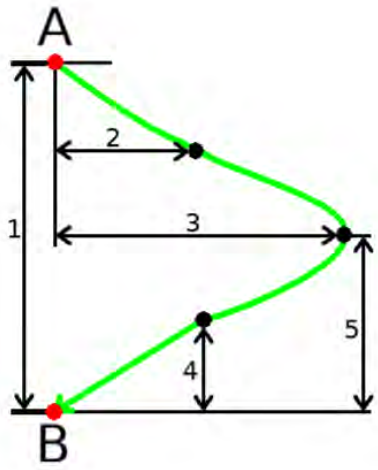


Figure 4.4: Description of curve trajectory (second part of the trajectory)

- Letter `A` is `Start spline position` which is the position where the curve is starting to be executed
- Letter `B` is `Place position` which is the position where the brick should be placed
- Number `1` is the distance between letter `A` and `B` which could be set by parameter `altitude curve offset`
- Number `2` is the half distance of the width of curve set by parameter `curve offset`.This distance is same for 2 points.
- Number `3` is the full distance of the width of curve set by parameter `curve offset`
- Number `4` is the quarter distance between letter `A` and `B`. This distance is the same for the point above this, but the distance is computed from the point `A` .

- Number `5` is the half distance between letter `A` and `B`.
- The approximate sketch of trajectory that the UAV will fly through is shown by green color on the figure.

Point `B` is included in the input message and by using this point and parameters are calculated all other points as in the first part. Next, these points are used for creating a curve trajectory by using functions from the created library named `uav localization core` [24]. The library is containing three functions for calculation curve points: `smooth path bspline`, `smooth path catmull` and `smooth path hermite`. Every function has parameters that are affecting the created trajectory. In pseudocode below 4 is described how the points for the curve trajectory are calculated by using one of these functions.

Algorithm 4 Description of function (calculate_points) which is creating points for curve trajectory

function CALCULATE_POINTS(type of trajectory, start position for spline, place position)
  Switch
    functions for individual types of trajectories ▷ Returning 5 RawPoints from Figure above
  EndSwitch
  for all points (RawPoints) **do**
    convert point to another structure ▷ Structure needed for used library
  end for
  SMOOTH PATH BSPLINE(input points, output points, reducer epsilon, line parameter, distance treshold, curve points) ▷ possibility to change trajectory
  STEP PATH(input points, output points, step len) ▷ step len is adjustable parameter
  for all point from function step path **do**
    convert point back to original structure ▷ structure usable by UAV
  end for
  return converted points
end function

In pseudocode is used function for creating a b-spline trajectory. The function has four parameters that are influencing the final points of the trajectory. Of course, the final points are much more influenced by the input points to this function. These parameters are:

- `reducer epsilon` is float number, which is used for limiting the total number of points. This algorithm is used [25] which is connecting two points and then calculate the shortest distance of the point from this straight line. If the distance is smaller than the number `reducer epsilon` then the point is removed.

- `line parameter` is float number, which is multiplying a vector, which is calculated from two consecutive points. The vector is multiplied by `line parameter` and summed with the point from which is calculated. The size of the vector can be limited by `distance threshold` parameter.

- `distance threshold` is float number, which is limiting the size of the summed vector with a point.

- `curve points` is integer number, which is setting the number of points created by cubic interpolation.

This function gradually takes a point and the previous point and calculates vector to the previous point, which is then multiplied by `line parameter`. If the vector isn't limited by `distance threshold`, then the vector is summed with the chosen point and the newly created point is added to the end of an array. Otherwise, the size of the vector is decreased and then is summed. Next, the chosen point is added to the array. The similar calculation of vector is done with the next point too and then the newly created point is added to array too. This process is executed for all input points except for the first and the last point. These points are added separately. In our case, the process is executed for 3 black points in figure 4.4.

The array is checked after each adding of a point. If the array is containing at least 4 points then the output points of the curve are created by using b-spline interpolation for the last 4 points of array. The number of created output points depends on `curve points` parameter. Equation for this interpolation is:

$$
\begin{aligned}
u =& \frac{i}{\texttt{curve points}} \\
\text{output point} =& \frac{u^3 \cdot (-P_0 + 3P_1 - 3P_2 + P_3)}{6} \\
&+ \frac{u^2 \cdot (3P_0 - 6P_1 + 3P_2)}{6} \\
&+ \frac{u \cdot (-3P_0 + 3P_2)}{6} \\
&+ \frac{P_0 + 4P_1 + P_2}{6} \\
& i \in N, < 0, \texttt{curve points} > \\
P_0, P_1, P_2, P_3 & \text{ are last four point of the array,} \\
& \text{where } P_3 \text{ is last and } P_0 \text{ is third to the last}
\end{aligned}
\tag{4.1}
$$

B-spline interpolation

If parameter `reducer epsilon` 4.2 is bigger than 0, then the number of output points can be reduced. The reduction depends on the location of points and the value of the parameter.

Example of the curve trajectory generated by B-spline interpolation with these parameters: `altitude curve offset` $= 1[m]$, `curve offset` $= 1.5[m]$, `reducer epsilon` $= 0.01$, `line parameter` $= 1.5$, `distance threshold` $= 1$, `curve point` $= 50$. The green points are input points to the function (computed points from parameters in config), the blue points are output points from function (the points are defining curve trajectory) and the red line is the final curve trajectory (connected blue points by the straight line).

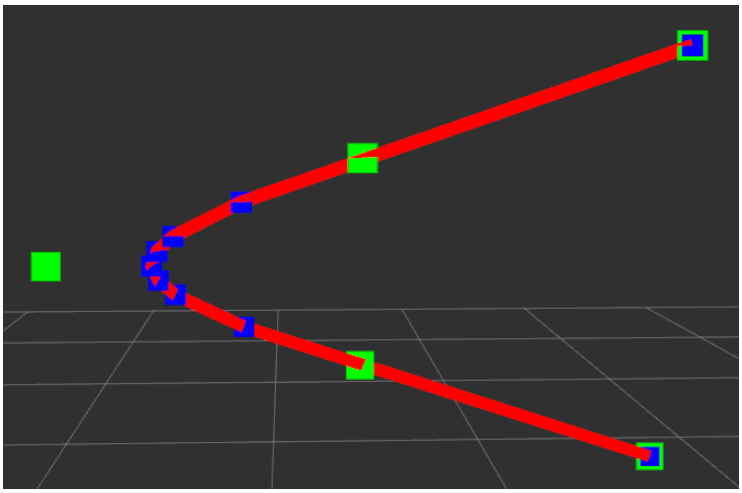


Figure 4.5: Example of curve trajectory generated by B-spline

The next function from the library is `smooth path catmull`. The function has only 2 parameters: `reducer epsilon` 4.2 and `curve points` 4.2, which are described above.

This function gradually takes all input points and adding them to the array without any change. The only exception is the first and the last point. These points are "multiplied", which means adding these points two times to the array. The array is checked after each adding of a point. If the array is containing at least 4 points then the output points of the curve are created by using Catmull-Rom spline interpolation for the last 4 points of array. The number of created output points depends on `curve points` parameter. Equation for this interpolation is:

$$
\begin{aligned}
u =& \frac{i}{\texttt{curve points}} \\
\text{output point} =& \frac{u^3 \cdot (-P_0 + 3P_1 - 3P_2 + P_3)}{2} \\
&+ \frac{u^2 \cdot (2P_0 - 5P_1 + 4P_2 - P_3)}{2} \\
&+ \frac{u \cdot (-P_0 + P_2)}{2} \\
&+ P_1 \\
& i \in N, < 0, \texttt{curve points} > \\
P_0, P_1, P_2, P_3 & \text{ are last four point of the array,} \\
& \text{where } P_3 \text{ is last and } P_0 \text{ is third to the last}
\end{aligned}
\tag{4.2}
$$

Catmull-Rom spline interpolation

Like in the function for b-spline, the number of output points can be reduced depending on the `reducer epsilon` 4.2 parameter and location of the points.

Example of the curve trajectory generated by Catmull-Rom spline interpolation with these parameters: `altitude curve offset` $= 1[m]$, `curve offset` $= 1.5[m]$, `reducer epsilon` $= 0.01$, `curve point` $= 50$. The green points are input points to the function (computed points from parameters in config), the blue points are output points from function (the points are defining curve trajectory) and the red line is the final curve trajectory (connected blue points by the straight line).

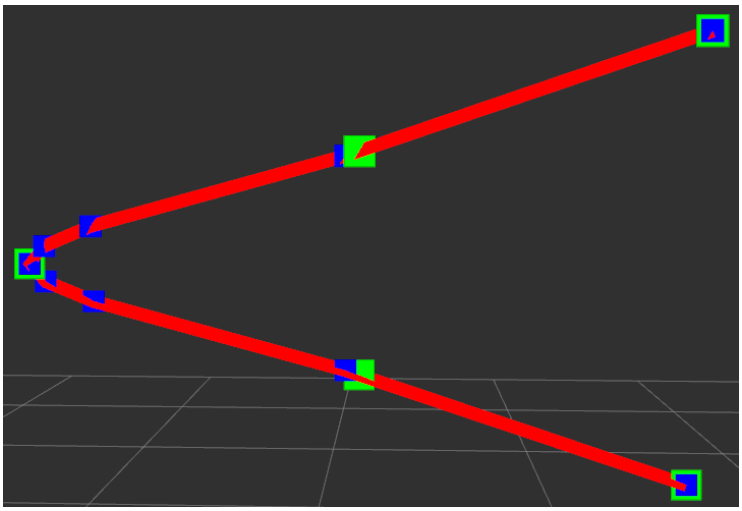


Figure 4.6: Example of curve trajectory generated by Catmull-Rom spline

The last function for computing the curve points for a trajectory is `smooth path hermite`. The function has the same parameters as the function for Catmull trajectory, that are `reducer epsilon` 4.2 and `curve points` 4.2, which are described above.

This function is using 4 input points (computed points from parameters in config) and by using these points compute one point of the curve trajectory.

$$\begin{aligned} \text{percent} =& \frac{i}{\texttt{curve points} - 1} \\ tx =& (\text{quantity of input points} - 1) \cdot \text{percent} \\ \text{index} =& \text{ full part of number tx} \\ u =& \text{ decimal part of number tx} \\ i \in& N, < 0, \texttt{curve points}) \\ P_0 =& \text{ index} - 1, P_1 = \text{ index}, \\ P_2 =& \text{ index} + 1, P_3 = \text{ index} + 2 \end{aligned} \tag{4.3}$$

If the index of the point (P_0,...,P_3) is negative (the point is not in the array), then it is taken the first point of the array (index = 0). If the index of the point is bigger or equal like the size of the array then it is taken the last point of the array. If the index is pointing in the array then it is taken this point from the array. The first and last points can be "multiplied", which means using these points two or more times as P_x. These points are then used for computing curve point by Hermite interpolation.

$$\begin{aligned} \text{output point} =& \frac{u^3 \cdot (-P_0 + 3P_1 - 3P_2 + P_3)}{2} \\ &+ \frac{u^2 \cdot (2P_0 - 5P_1 + 4P_2 - P_3)}{2} \\ &+ \frac{u \cdot (-P_0 + P_2)}{2} \\ &+ P_1 \end{aligned} \tag{4.4}$$

Hermite spline interpolation

Like in the two previous functions, the number of output points can be reduced depending on the `reducer epsilon` 4.2 parameter and location of the points.

Example of the curve trajectory generated by Hermite spline interpolation with these parameters: `altitude curve offset` = 1[m], `curve offset` = 1.5[m], `reducer epsilon` = 0.01, `curve point` = 50. The green points are input points to the function (computed points from parameters in config), the blue points are output points from function (the points are defining curve trajectory) and the red line is the final curve trajectory (connected blue points by the straight line).

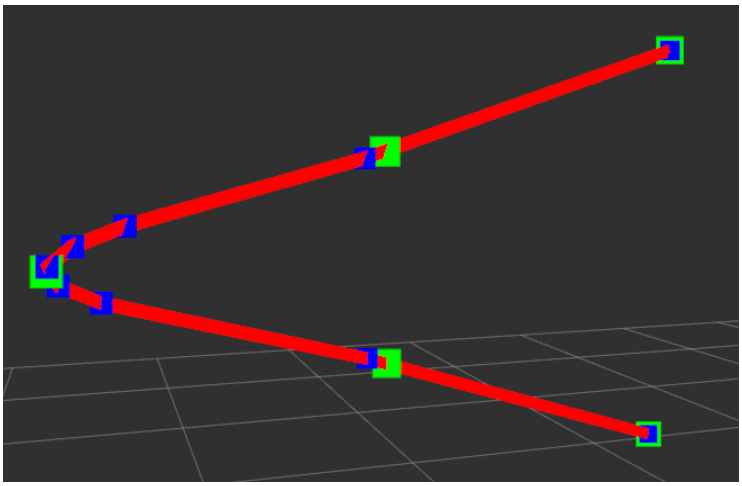


Figure 4.7: Example of curve trajectory generated by Hermite spline

By using one of the three functions mentioned above is generated curve trajectory which is defined by output points from these functions. Next, the points are split into two halves and used in the function named `step path`. The function "slices" the created curve trajectory by using parameter `step len`. The second half, of the split points, is used in the same function but with the half `step len` parameter for slower velocity. Distance between created final points is precisely `step len` (or `step len`/2) . Points with the same gap between themselves are created and sent to the MPC trajectory tracker.

Example of the sliced points from a curve trajectory. Parameter `step len` = 0.1. The green points are raw points (computed points from parameters in config), the red points are "sliced" points from the function (the points are the final points, through which the UAV will fly) and the blue line is the curve trajectory (trajectory generated from spline function).

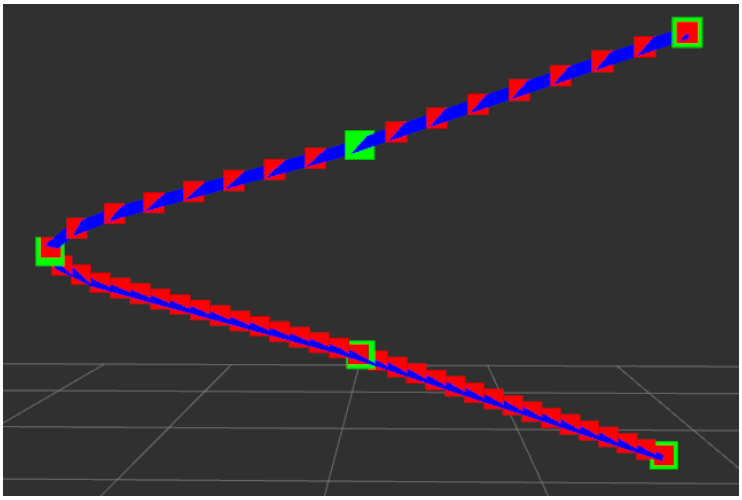


Figure 4.8: Example of "sliced" points

The final points ("sliced" points) are sent to the MPC trajectory tracker, which is more described in [26] section D. The tracker supposes that the points are sampled with 0.2s. Therefore, the velocity between each point will be equal to the distance between them divided by 0.2.

$$\text{velocity} = \frac{\texttt{step len}}{0.2} \tag{4.5}$$

Theoretical compute of velocity for "sliced" points (first half)

In reality, the velocity of the UAV will be affected by the initial state of UAV and the limitations of the MPC tracker. For example, the limitations include limitation of velocity, acceleration and cylinder radius (the sharpness of the curve). MPC tracker hence limitate the behavior of the UAV so that the UAV is able to fly through the trajectory. In general, the smaller parameter `step len` (more final points) causes the slower flying through curve trajectory and smoother placing of the brick.

Creating of curve trajectory is easily modifying by changing 3 parameters in `config` file or by changing parameters one of three functions for curve trajectory, which is just used. This easy modifying provides a changing of curve trajectory according to the requirements.

The MPC tracker is checking the surroundings of the UAV and trying to avoid collisions with other UAVs. The attempt to avoid a collision could disturb the placing of the brick and therefore it is supposed that the UAVs won't try to simultaneously place the brick near to each other. This should be provided by high-level motion planning according to the width of the curve.

4.3 State machine for placing method

- `IDLE_STATE` is an inactive state. The code is waiting for the input message from a high-level motion planning method.
- `TRAVEL1_STATE` is state which ensures the move of UAV to start position for placing the brick. The move is accomplished by using service `go_to` 4.3. Next, this state is checking an exceeding number of tries for repeating the placing and if the number is overrun then the placing of the brick is aborted. The UAV will start moving to the final position with the brick after method is aborted.
- `DROP1_STATE` is state which ensures descending to start spline position by using the trajectory described 4.3. But if the start position for placing the brick is the same as start spline position then this state is skipped. Next, if this state doesn't finish the trajectory in the predefined time period by parameter then the state is switched to `TRAVEL1_STATE` and another attempt will be tried. Finishing of the state is checked by comparing actual position with a goal position.

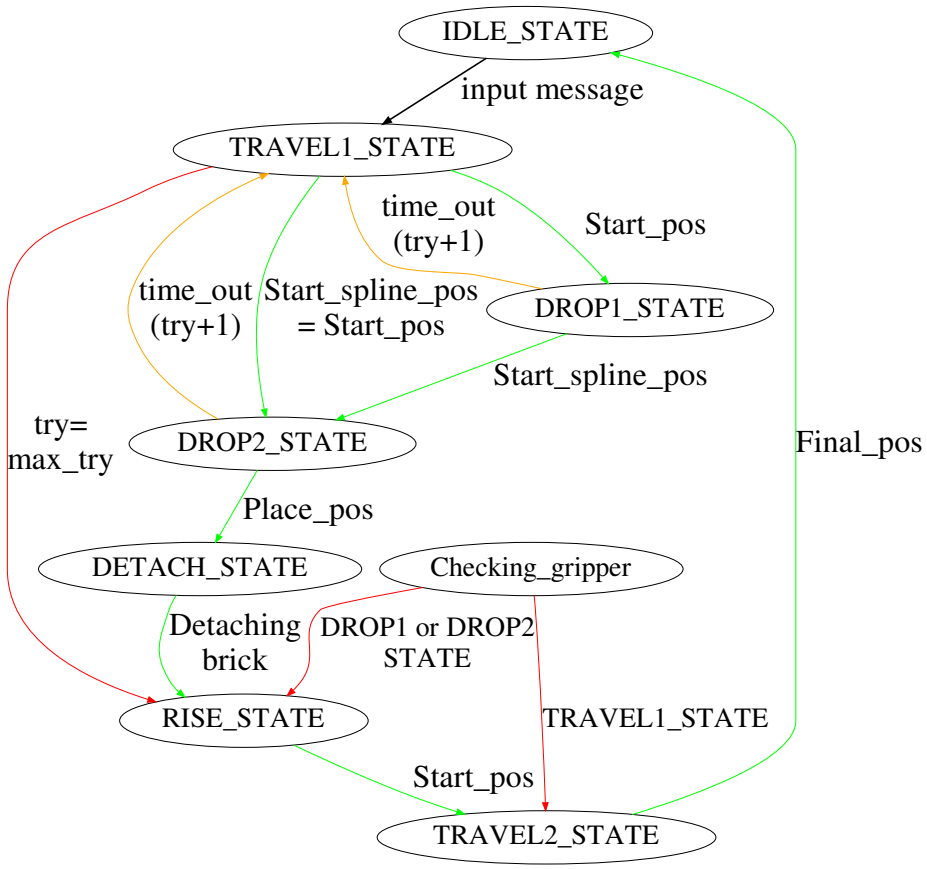


Figure 4.9: State machine for placing procedure

- `DROP2_STATE` is state which ensures executing of curve trajectory described 4.4. If the state doesn't finish the trajectory in the predefined time period by parameter then the state is switched to `TRAVEL1_STATE` and another attempt will be tried. Finishing of the state is checked by comparing actual position with a goal position.

- `DETACH_STATE` is state which ensures detaching of the brick from UAV. Finishing the state is checked by successfully releasing the brick from the gripper.

- `RISE_STATE` is state which ensures the rising of the UAV to the start position. The move is accomplished by using service `go_to` 4.3.

- `TRAVEL2_STATE` is state which ensures the last move to the final position before finishing the procedure. The move is accomplished by using service `go_to` 4.3.

- `Checking gripper` is function which is checking the actual state of the gripper. So the function is checking if the brick is still attached to the UAV. If the UAV loses the brick that is just being carried then the procedure is abandoned with error and

publish a message with the position where the brick was lost to the terminal. The function changes the state according to the current state and occurs in finishing the procedure with the error.

The service `go_to` is used for planning a trajectory of UAV with collision avoidance. If multiple UAVs are deployed then it is assumed that they are localized within the same world coordinate system. Furthermore, it is assumed that the position of each UAV is well known, e.g. with the use of a GPS sensor. The method which is used by the service is more described in this article [26].This service is included in node `mpc_tracker` and send the UAV to a goal position in global coordinates.

Chapter 5

Experiments in simulator

Contents

To test the precision of the method for placing the brick, the tests were realized in Gazebo simulator under the Robot Operating System. The map, in which were the experiments are realized, is the map created in Chapter 1 of this work. To the center of the map was added blue brick which simulated brick wall under construction 5.1. The searching of the best trajectory for placing the brick is tested only on one type of trajectory (type 2). The precision of the left four trajectory types will be tested with found best trajectory.

The main objective of the experiments is to place the brick as close as possible to the blue brick without damaging already place brick. The required coordinates of the newly placed brick are $x = 0.75$, $y = 0.00$ (for trajectory type 2). With these coordinates, the brick is touched to the already placed brick (fits perfectly). During the experiments is the biggest emphasis attached on not to damage the already placed bricks. Then the emphasis is descending attached to the precision of placing the brick, time of executing the whole trajectory and length of the trajectory. All three functions for compute spline, which are mentioned above, are used to find the optimal trajectory, which is best fulfills to the requirements. For executing each test is sent a message with this information: [`place position:` [x,y,z,yaw], `final position:` [x,y,z,yaw] and `trajectory type:` [1-5], at the beginning.

During each experiment is executing grasping of the brick. For this reason, a different result can occur though the set parameters are the same. The precision of placing the

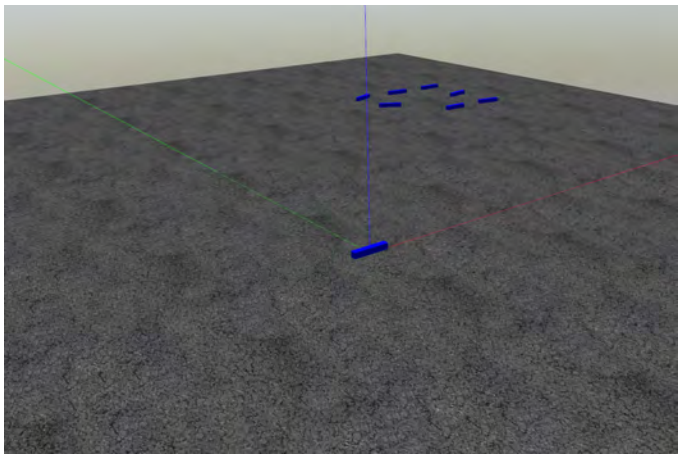


Figure 5.1: World modified for experiments

brick is strictly depending on the grasping of the brick. For the biggest possible precision of grasping the brick is used the newest update of the node for MBZIRC competition (upgraded version of method from Chapter 3).

5.1 Experiments with using of the Hermite interpolation

In the first part of the tables (light blue color) are parameters, which can be modified in config file. The parameters for Hermite interpolation are set as: reducer epsilon = 0.01 and curve points = 50. In the second part of the table (white color) are measured metrics from the simulator. Value in "deviation of placed brick" column is computed from the required coordinates ($x = 0.75$,$y = 0.00$). Value in "damage of already placed brick" column is true if the brick is shifted in any way from the original position. Value in "touch the placed brick" is true if the newly placed brick is touching the already placed brick. The last column "Success" is true if the method is finished without errors.

During these experiments was changed "place position" because of finding a more precise trajectory. For the first three rows is "place position" equals to $[0.7, 0.0, 0.1, 1.57]$. In two next rows (4,5) was decreased z coordinates, which was then again raised to $0.1m$ because the brick touched the ground sometimes. In the first five rows from the table can be seen that the grasped brick is shifted of $0.03m$ in coordinate y, that is why "place position" is modified in coordinate y to 0.03. In the seventh row is a curve trajectory enough precise so coordinate x could be changed to the required coordinate (0.75). "Place position" is not modified from the seventh row and the value of the position is [0.75,0.03,0.1, 1.57].

The parameter `altitude offset` is fixed on value $0.5[m]$. This value could be more decreased for the current state of the wall, but it could cause a collision with already placed bricks. Therefore it is left sufficient space between just carried brick and wall.

No.	Altitude offset [m]	Step len [m]	Curve offset [m]	Altitude curve offset [m]	Time of executing whole trajectory [s]	Real length of whole trajectory [m]	Planned length of whole trajectory [m]	Deviation of placed brick [m]		Damage of already placed brick	Touch the placed brick	Success
1	0.5	0.05	0.5	0.4	10.49	0.98	1.18	0.003	0.036	F	T	T
2	0.5	0.05	0.5	0.3	10.35	0.98	1.22	0.021	0.032	F	F	T
3	0.5	0.05	0.5	0.25	11.29	1.00	1.26	0.028	0.034	F	F	T
4	0.5	0.05	0.3	0.25	8.52	0.67	0.88	0.011	0.038	F	F	T
5	0.5	0.01	0.3	0.25	21.20	0.79	0.90	0.034	0.032	F	F	T
6	0.5	0.08	0.28	0.25	6.76	0.57	0.83	0.011	0.006	T	T	T
7	0.5	0.08	0.28	0.25	6.71	0.57	0.83	0.025	0.011	F	F	T
8	0.5	0.065	0.25	0.25	7.11	0.58	0.79	0.048	0.007	F	F	T
9	0.5	0.08	0.25	0.25	6.72	0.55	0.77	0.030	0.005	F	F	T
10	0.5	0.08	0.2	0.25	6.37	0.52	0.72	0.014	0.008	F	F	T
11	0.5	0.085	0.2	0.25	6.34	0.52	0.72	0.012	0.006	F	F	T
12	0.5	0.09	0.2	0.25	6.31	0.52	0.72	0.017	0.013	F	F	T
13	0.5	0.085	0.2	0.25	6.44	0.52	0.72	0.013	0.012	F	F	T
14	0.5	0.085	0.15	0.5	5.0	0.516	0.580	0.009	0.000	F	F	T
15	0.5	0.085	0.15	0.5	4.99	0.516	0.579	0.014	0.015	F	F	T

Table 5.1: Table of experimental result with using of Hermite interpolation

In the last six rows of the table are deviation x and y only around 1 cm. The size of the deviation is changing accordingly to the grasping of the brick. At the same time, the brick wall isn't damaged that's why this trajectory is satisfactory for the requirements of the task. As optimal trajectory for Hermite interpolation can be chosen two trajectories with these parameters: `altitude offset`: $0.5[m]$, `step len`: $0.085[m]$, `curve offset`: $0.2[m]$ and `altitude curve offset`: $0.25[m]$ for the first trajectory (rows 11 and 13) and `altitude offset`: $0.5[m]$, `step len`: $0.085[m]$, `curve offset`: $0.15[m]$ and `altitude curve offset`: $0.5[m]$ for the second trajectory (rows 14 and 15).

The second trajectory (rows 14,15) has similar deviations of placing the brick, but the execution of trajectory is faster. This is caused by omitting the first part of the trajectory (4.3). On the other hand, the carried brick is extremely close to the placed brick if the brick is attached far from UAV (probably can't occur in real simulations), and a collision with the wall could occur. From this reason is chosen the first trajectory (rows 11,13) as optimal which is slower but safer. The required position for placing the brick is $[0.75, 0]$ and the sent message contains: `place position`: $[0.75, 0.03, 0.05, 1.57]$, `final position`: $[0.75, 0.03, 0.9, 1.57]$, `type of trajectory`: 2.

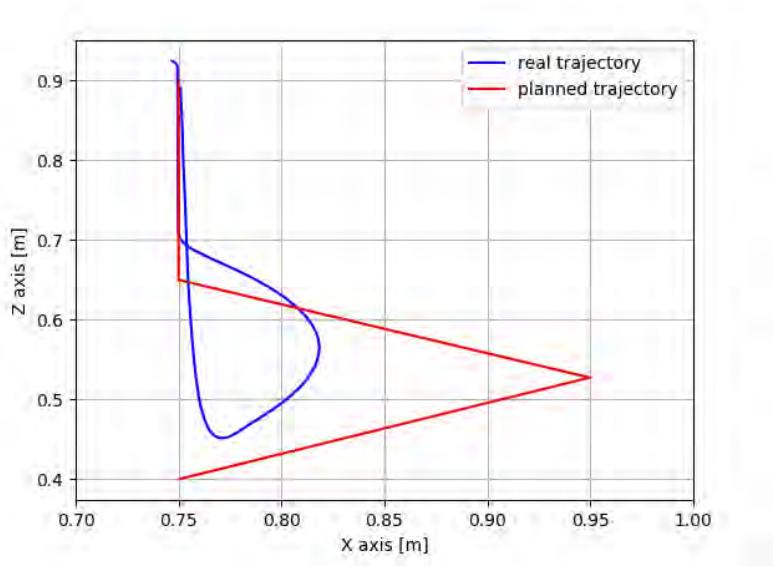
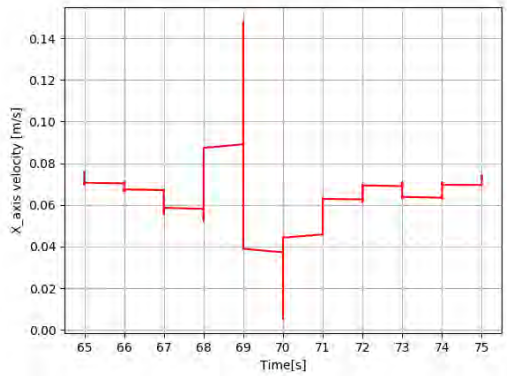


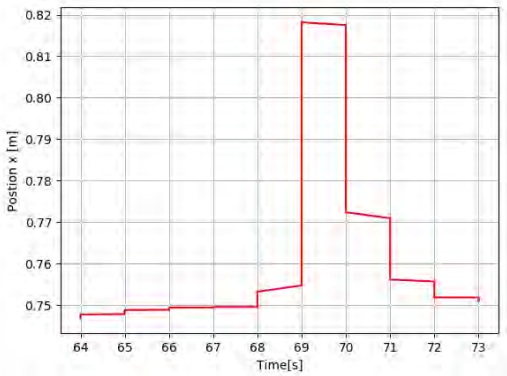
Figure 5.2: Planned and real trajectory in 2D, axis zx, Hermite interpolation

The trajectory in figure 5.2 is realized from beginning of `DROPPING1_STATE` to end of `RISE_STATE`. Red line is a planned trajectory (points sent to the MPC tracker) and blue line is a really executed trajectory (information from odometry, e.g. GPS). The red line isn't rising back with the blue line, because it is used `go_to` service for rising, which is included directly in MPC tracker. The 3D plots of this trajectory can be seen in Appendices.

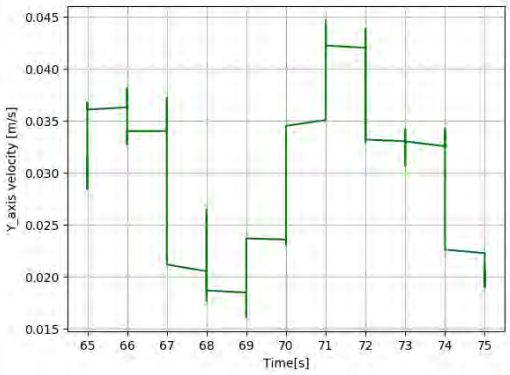
The UAV isn't able to execute the red trajectory (too sharp curve and high speed), that's why the MPC tracker changed the trajectory.



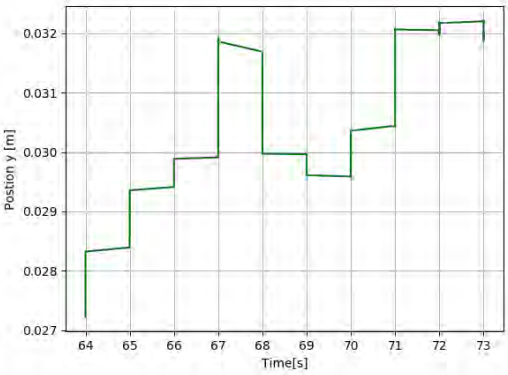
(a) Dependence of speed of UAV in coordinate x on time



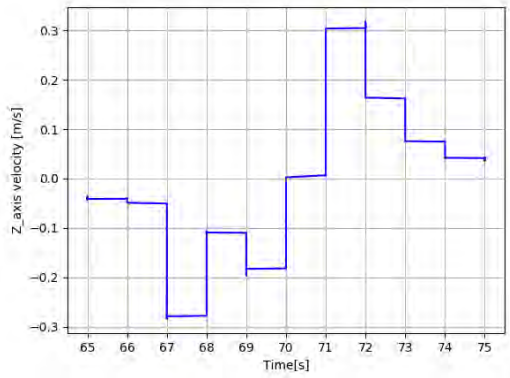
(b) Dependence of position x of UAV on time



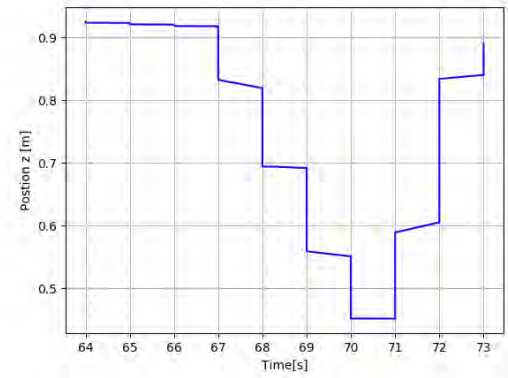
(c) Dependence of speed of UAV in coordinate y on time



(d) Dependence of position y of UAV on time



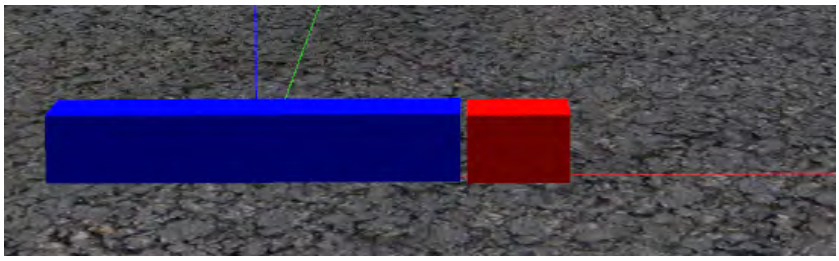
(e) Dependence of speed of UAV in coordinate z on time



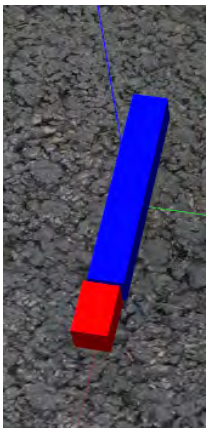
(f) Dependence of position z of UAV on time

Figure 5.3: The figure contains dependencies of velocities and positions in all coordinates on time

From figure 5.3, we can see a change of position and velocity in dependence of time. When is the velocity changed then it is following a change of position with some time response. The velocity and the position is measured by a sensor on a UAV (e.g. accelerometer, GPS), which means that the values can have some deviation.



(a) The final position of placed brick with optimal trajectory, view in coordinates x,z



(b) The final position of placed brick with optimal trajectory, view in coordinates x,y

Figure 5.4: The final position of placed brick with optimal trajectory, Hermite interpolation

The figure 5.4 shows the placed red brick next to the blue brick, which is the result of experiment, which is described by the above-mentioned figures. The deviation of placed brick is 0.018 for x coordinate and 0.018 for y coordinate.

In the figures 5.5 below is displayed a sequence of pictures from video, which is describing this experiment. Unfortunately, it wasn't possible to create the graphs and video simultaneously, that's why the video was created in the next run of simulation with the same parameters. The placed brick can have other deviations and graphs have probably different shapes. The whole video is available on: `https://www.youtube.com/watch?v=tqEX1YPlZOY`

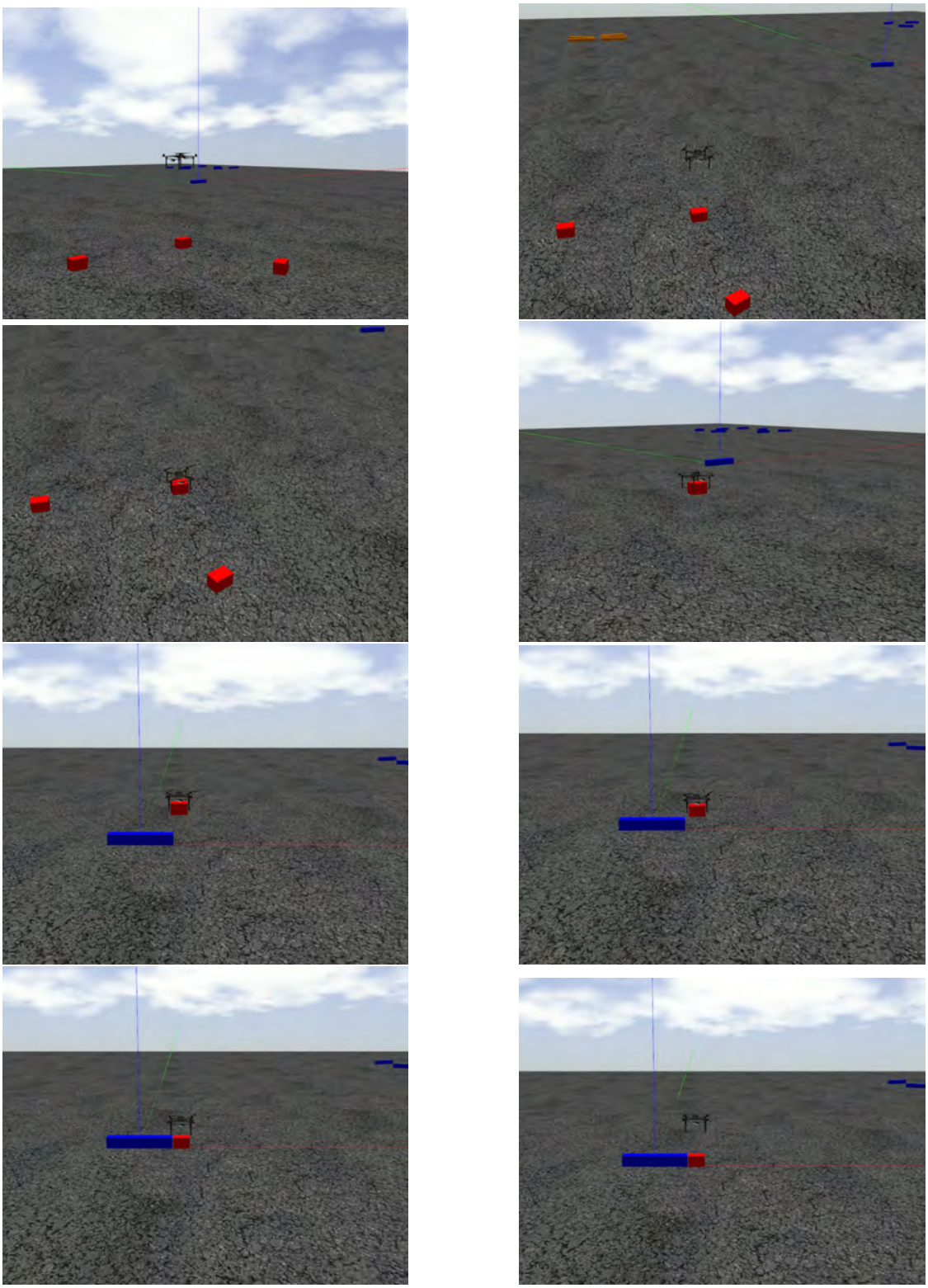


Figure 5.5: The figure displays a sequence of pictures from video, which is describing the experiment with a trajectory from Hermite interpolation

5.2 Experiments with using of the Catmull-Rom interpolation

In the first part of the tables (light blue color) are parameters, which can be modified in config file. The parameters for Catmull-Rom interpolation are set as: reducer epsilon = 0.01 and curve points = 50. In the second part of the table (white color) are measured metrics from the simulator. The same meaning of columns like in the previous table 5.1.

The Catmull-Rom interpolation has similar trajectories as Hermite interpolation, which can be seen in figures 4.7, 4.6. Therefore the searching for optimal trajectory was much easier and the first estimate of parameters is taken from optimal Hermite trajectory. The required trajectory is set at coordinates $[0.75, 0]$ from the beginning and "Place position" sent to UAV is [0.75,0.03,0.1, 1.57]. Again, the parameter `altitude offset` is fixed on value $0.5[m]$.

No.	Altitude offset [m]	Step len [m]	Curve offset [m]	Altitude curve offset [m]	Time of executing whole trajectory [s]	Real length of whole trajectory [m]	Planned length of whole trajectory [m]	Deviation of placed brick [m]	Damage of already placed brick	Touch the placed brick	Success
1	0.5	0.085	0.2	0.25	6.29	0.518	0.722	0.029	F	F	T
2	0.5	0.085	0.2	0.25	6.31	0.519	0.722	0.014	0.000	F	F	T
3	0.5	0.085	0.2	0.25	6.42	0.521	0.722	0.011	0.002	F	F	T
4	0.5	0.085	0.2	0.25	6.23	0.516	0.722	0.019	0.014	F	F	T
5	0.5	0.095	0.2	0.25	6.39	0.519	0.722	0.021	0.010	F	F	T
6	0.5	0.095	0.2	0.25	6.27	0.513	0.722	0.021	0.014	F	F	T
7	0.5	0.075	0.2	0.25	6.54	0.538	0.722	0.017	0.011	F	F	T
8	0.5	0.075	0.2	0.25	6.6	0.534	0.722	0.027	0.009	F	F	T
9	0.5	0.075	0.2	0.25	6.56	0.540	0.722	0.017	0.013	F	F	T
10	0.5	0.05	0.2	0.25	7.2	0.568	0.722	0.066	0.004	F	F	T
11	0.5	0.085	0.25	0.3	6.53	0.551	0.783	0.033	0.005	F	F	T
12	0.5	0.085	0.25	0.3	6.68	0.555	0.783	0.025	0.004	F	F	T
13	0.5	0.085	0.15	0.5	5.3	0.516	0.583	0.011	0.007	F	F	T
14	0.5	0.085	0.15	0.5	5.3	0.520	0.583	0.019	0.004	F	F	T

Table 5.2: Table of experimental result with using of Catmull-Rom interpolation

Collision with the wall didn't occur during all experiments with this interpolation. Therefore the optimal trajectory is chosen according to the smallest deviation of placed brick. The smallest values are in rows 2,3,4 and 13,14. As optimal trajectory for Catmull-Rom interpolation can be chosen two trajectories, which are with the same parameters as for Hermite interpolation. As mentioned above, these interpolations are similar for our point set, therefore it is chosen the second trajectory, which wasn't shown.

The second trajectory have these parameters: `altitude offset`: $0.5[m]$, `step len`: $0.085[m]$, `curve offset`: $0.15[m]$ and `altitude curve offset`: $0.5[m]$ for the second trajectory (rows 13 and 14). This trajectory is omitting the first part of the whole trajectory for placing (4.3).

The required position for placing the brick is $[0.75, 0]$ and the sent message contains: `place position`: $[0.75, 0.03, 0.05, 1.57]$, `final position`: $[0.75, 0.03, 0.9, 1.57]$, `type of trajectory`: 2.

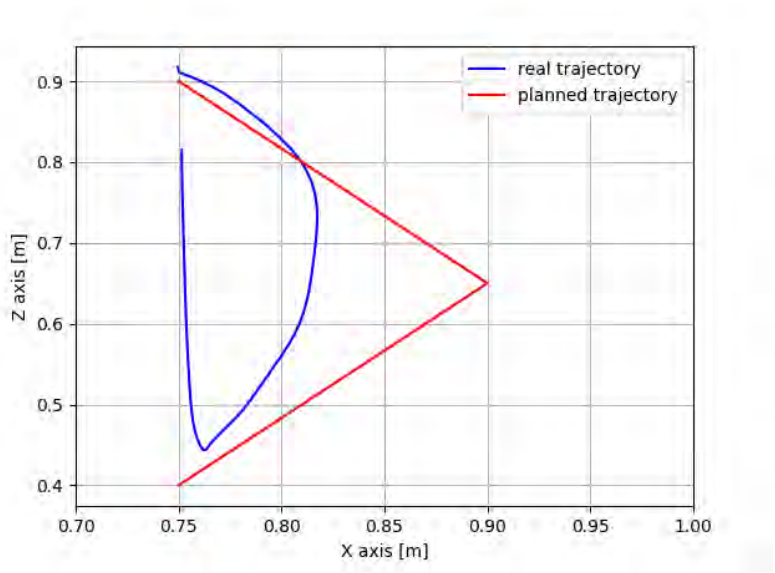
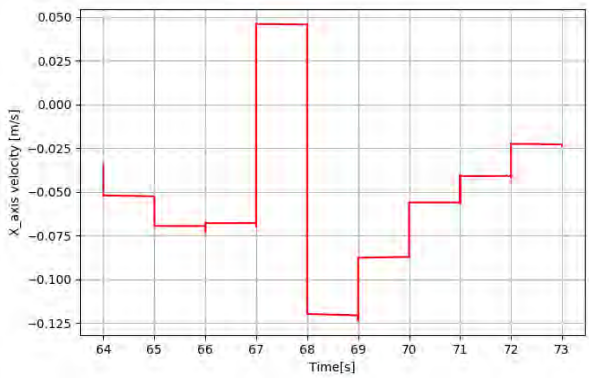


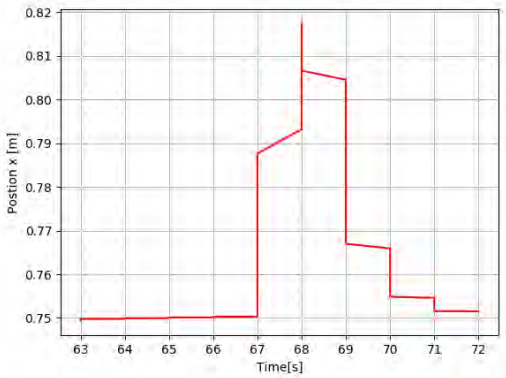
Figure 5.6: Planned and real trajectory in 2D, axis zx, Catmull-Rom interpolation

The trajectory in figure 5.6 is realized from beginning of `DROPPING1_STATE` to end of `RISE_STATE`. Red line is a planned trajectory (points sent to the MPC tracker) and blue line is a really executed trajectory (information from odometry, e.g. GPS). The red line isn't rising back with the blue line, because it is used `go_to` service for rising, which is included directly in MPC tracker. The 3D plots of this trajectory can be seen in Appendices.

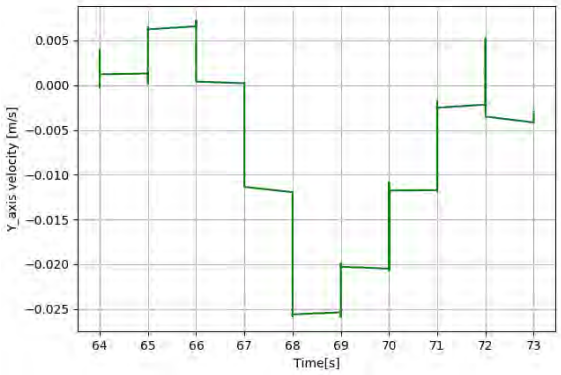
The UAV isn't able to execute the red trajectory similarly as in the Hermite interpolation, but the planned trajectory isn't so difficult and therefore the real trajectory is much closer to the planned trajectory.



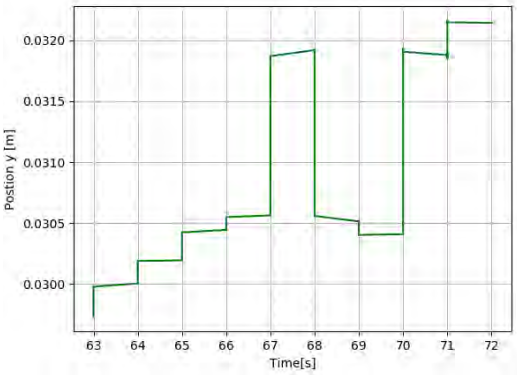
(a) Dependence of speed of UAV in coordinate x on time



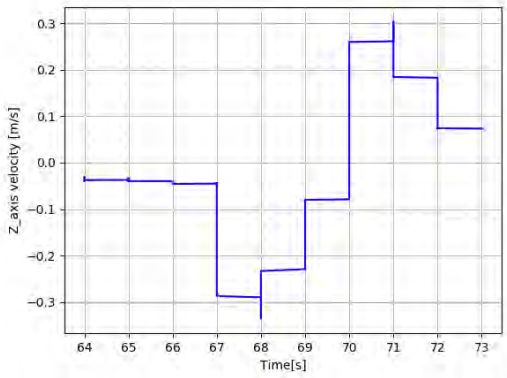
(b) Dependence of position x of UAV on time



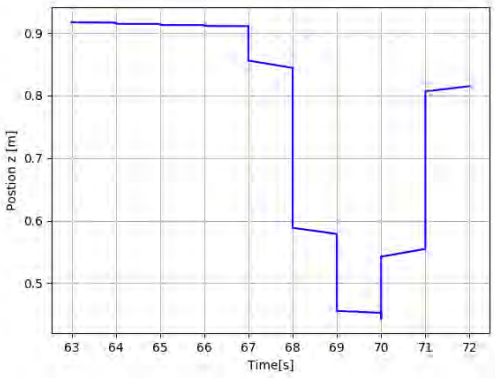
(c) Dependence of speed of UAV in coordinate y on time



(d) Dependence of position y of UAV on time



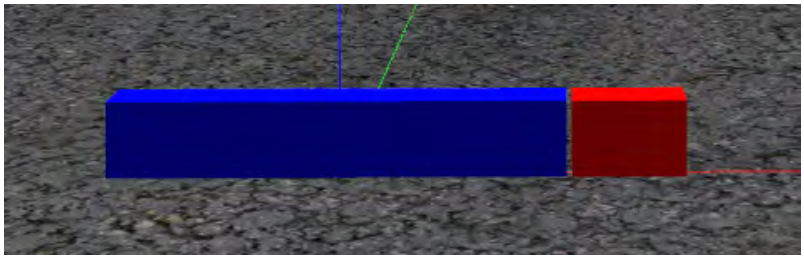
(e) Dependence of speed of UAV in coordinate z on time



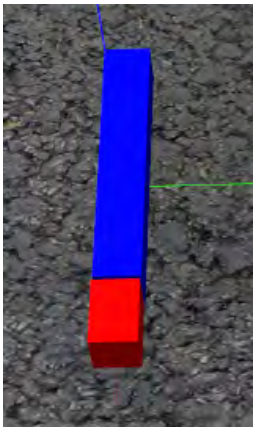
(f) Dependence of position z of UAV on time

Figure 5.7: The figure contains dependencies of velocities and positions in all coordinates on time, Catmull-Rom interpolation

From figure 5.7, we can see a change of position and velocity in dependence of time. The velocity and the position is measured by a sensor on a UAV (e.g. accelerometer, GPS), which means that the values can have some deviation.



(a) The final position of placed brick with optimal trajectory, view in coordinates x,z



(b) The final position of placed brick with optimal trajectory, view in coordinates x,y

Figure 5.8: The final position of placed brick with optimal trajectory, Catmull-Rom interpolation

The figure 5.8 shows the placed red brick next to the blue brick, which is the result of experiment, which is described by the above-mentioned figures. The deviation of placed brick is 0.014 for x coordinate and 0.013 for y coordinate.

In the figures 5.9 below is displayed a sequence of pictures from video, which is describing this experiment. As in the previous experiments, it wasn't possible to create the graphs and video simultaneously, that's why the video was created in the next run of simulation with the same parameters. The graphs and deviations of placed brick are not corresponding with the video. The whole video is available on: `https://www.youtube.com/watch?v=2pzTVb4q7p8`

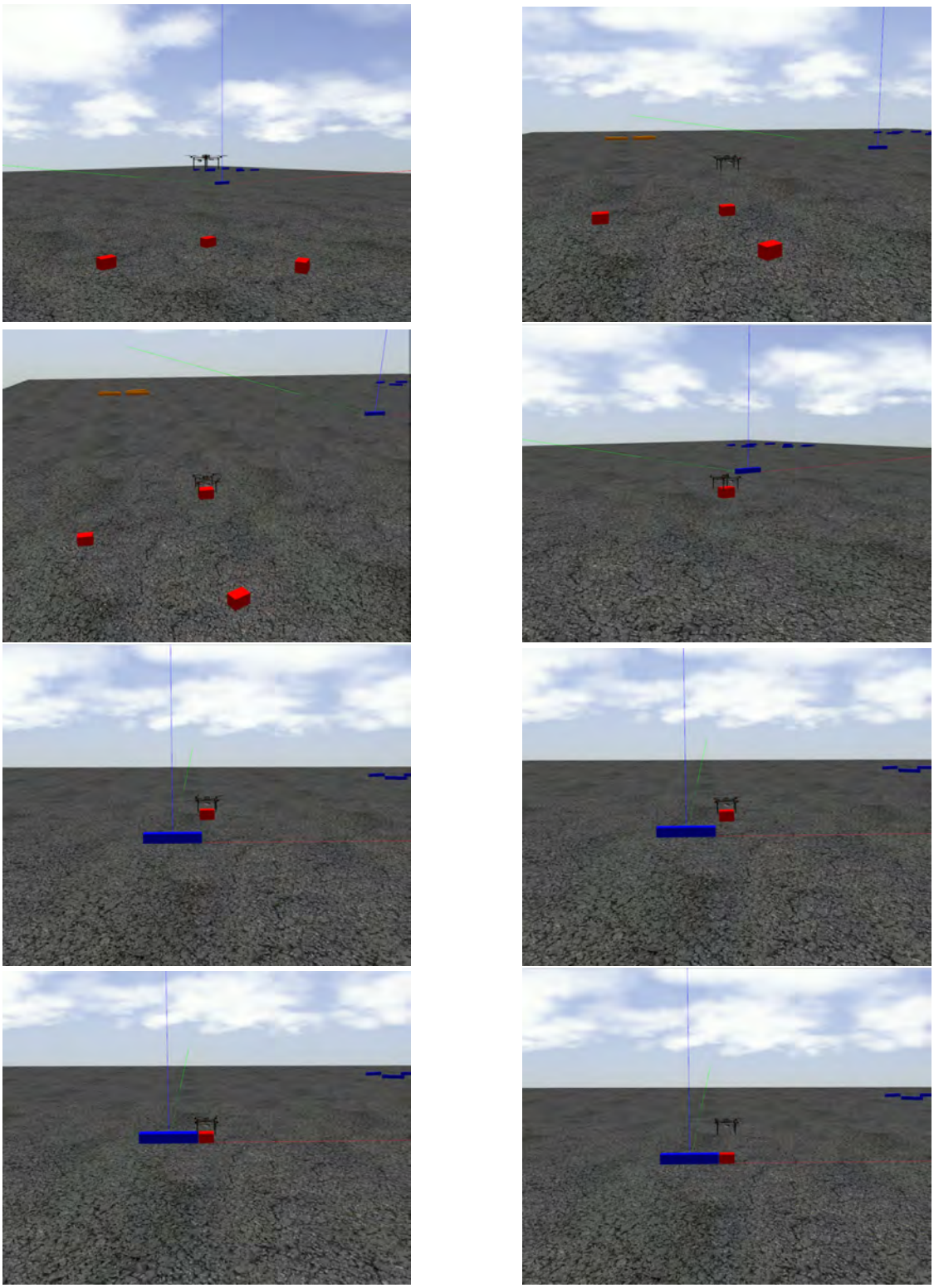


Figure 5.9: The figure displays a sequence of pictures from video, which is describing the experiment with a trajectory from Catmull-Rom interpolation

5.3 Experiments with using of the B-spline interpolation

In the first part of the tables (light blue color) are parameters, which can be modified in config file. Next two columns contain parameters to set function for B-spline, which are `line parameter` and `distance threshold`. Left two parameters are set as: `reducer epsilon` $= 0.01$ and `curve points` $= 50$. In the second part of the table (white color) are measured metrics from the simulator. The same meaning of columns like in the previous table 5.1.

The B-spline interpolation has the possibility to change function parameters, against two mentioned interpolations. The first estimate of parameters is again taken from optimal Hermite trajectory. It was executed more experiments with this interpolation although the approximate results are known from previous two interpolations. This is caused by the possibility to modify the trajectory through parameters in the function. The required trajectory is set at coordinates $[0.75, 0]$ from the beginning and "Place position" sent to UAV is [0.75,0.03,0.1, 1.57]. Again, the parameter `altitude offset` is fixed on value $0.5[m]$.

No.	Altitude offset [m]	Step len [m]	Curve offset [m]	Altitude curve offset [m]	Line parameter	Distance threshold	Time of executing whole trajectory [s]	Real length of whole trajectory [m]	Planned length of whole trajectory [m]	Deviation of placed brick [m]	Damage of already placed brick	Touch the placed brick	Success
1	0.5	0.085	0.20	0,25	1.5	1	7.79	0.520	0.672	0.020	F	F	T
2	0.5	0.085	0.20	0.25	1.5	1	6.19	0.509	0.670	0.020	F	F	T
3	0.5	0.085	0.20	0.25	0.5	10	6.31	0.514	0.670	0.013	F	F	T
4	0.5	0.085	0.20	0.25	0.5	10	6.19	0.508	0.670	0.020	F	F	T
5	0.5	0.085	0.30	0.35	0.5	10	6.58	0.560	0.765	0.019	F	F	T
6	0.5	0.070	0.30	0.35	0.5	10	6.74	0.580	0.768	0.040	F	F	T
7	0.5	0.080	0.25	0.50	0.5	10	5.62	0.552	0.657	0.024	F	F	T
8	0.5	0.080	0.25	0.50	0.5	10	5.72	0.557	0.657	0.018	F	F	T
9	0.5	0.080	0.15	0.50	0.5	10	5.31	0.512	0.562	0.009	F	F	T
10	0.5	0.085	0.15	0.50	0.5	10	5.17	0.517	0.562	0.009	F	F	T
11	0.5	0.085	0.15	0.50	1.5	1	5.20	0.511	0.562	0.011	F	F	T
12	0.5	0.085	0.15	0.50	1.5	1	5.10	0.514	0.562	0.015	F	F	T
13	0.5	0.060	0.15	0.50	1.5	1	5.66	0.525	0.562	0.015	F	F	T
14	0.5	0.060	0.15	0.50	1.5	1	6.96	0.528	0.562	0.021	F	F	T
15	0.5	0.060	0.15	0.50	0.5	10	5.73	0.522	0.562	0.025	F	F	T
16	0.5	0.070	0.15	0.50	0.5	10	5.56	0.521	0.562	0.017	F	F	T
17	0.5	0.070	0.15	0.50	0.5	10	5.50	0.519	0.562	0.014	F	F	T
18	0.5	0.080	0.15	0.50	0.5	10	5.20	0.514	0.562	0.020	F	F	T

Table 5.3: Table of experimental result with using of B-spline interpolation

Collision with the wall didn't occur during all experiments with this interpolation. Therefore the optimal trajectory is chosen according to the smallest deviation of placed brick. The smallest values are in rows 9,10,11. If we compare row 9 with row 18, which has the same parameters, we can see the bigger dispersion against the rows 10,11. Therefore, the optimal parameters are chosen according to row 10 which has smaller deviations. The parameters are : `altitude offset`: $0.5[m]$, `step len`: $0.085[m]$, `curve offset`: $0.15[m]$, `altitude curve offset`: $0.5[m]$, `line parameter`: 0.5 and `distance threshold`: 10. This trajectory is omitting the first part of the whole trajectory for placing (4.3).

The required position for placing the brick is $[0.75, 0]$ and the sent message contains: `place position`: $[0.75, 0.03, 0.05, 1.57]$, `final position`: $[0.75, 0.03, 0.9, 1.57]$, `type of trajectory`: 2.

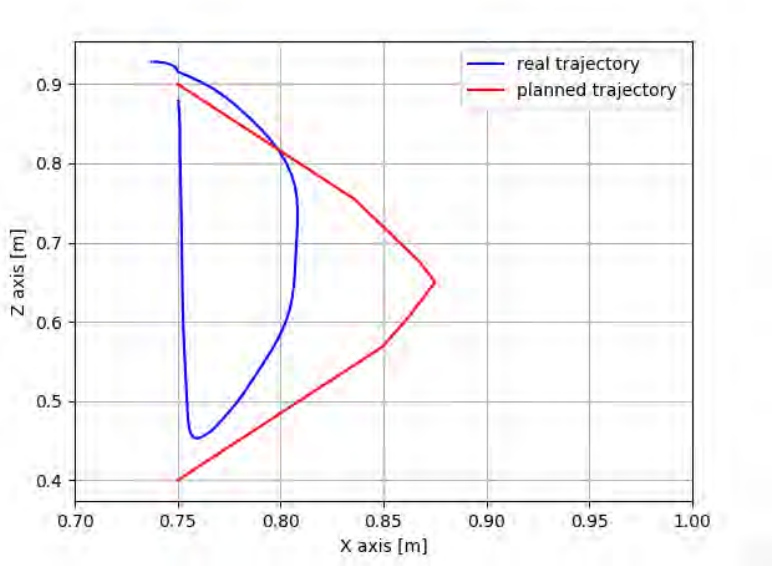
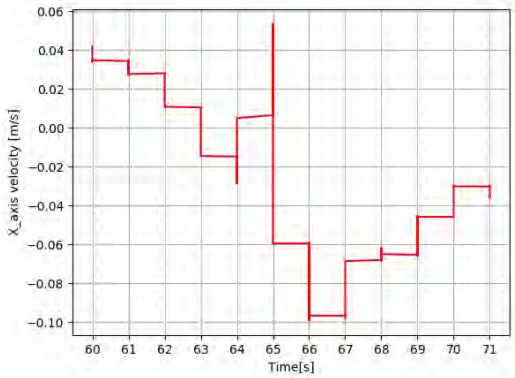


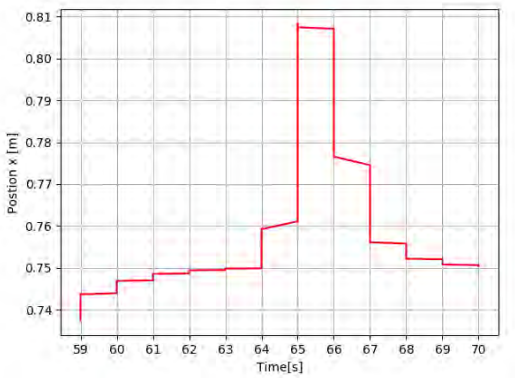
Figure 5.10: Planned and real trajectory in 2D, axis zx, B-spline interpolation

The trajectory in figure 5.10 is realized from beginning of `DROPPING1_STATE` to end of `RISE_STATE`. Red line is a planned trajectory (points sent to the MPC tracker) and blue line is a really executed trajectory (information from odometry, e.g. GPS). The red line isn't rising back with the blue line, because it is used `go_to` service for rising, which is included directly in MPC tracker. The 3D plots of this trajectory can be seen in Appendices.

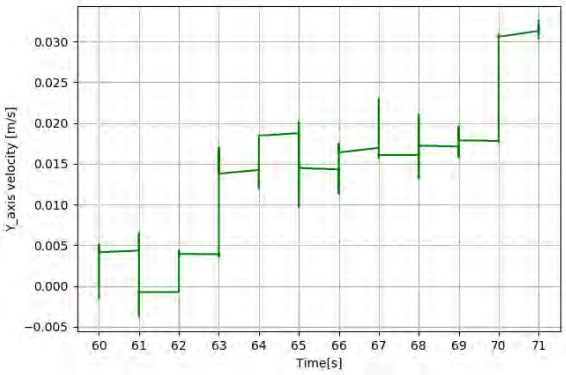
The UAV isn't able to execute the red trajectory similarly as in the previous interpolations. We can see that the UAV is rising up before reaching the place position. This is happening during all interpolations. The reaching of a position is checked by flying into a "circle" around place position. For fixing this problem is needed to decrease the circle in code (number in if condition).



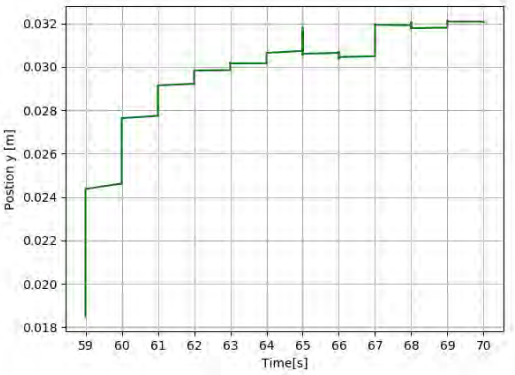
(a) Dependence of speed of UAV in coordinate x on time



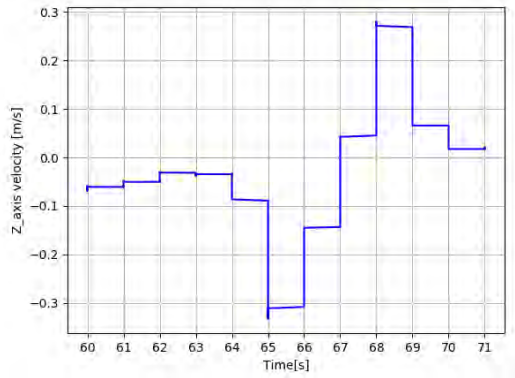
(b) Dependence of position x of UAV on time



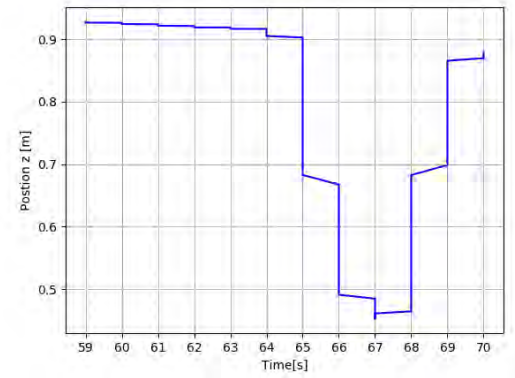
(c) Dependence of speed of UAV in coordinate y on time



(d) Dependence of position y of UAV on time



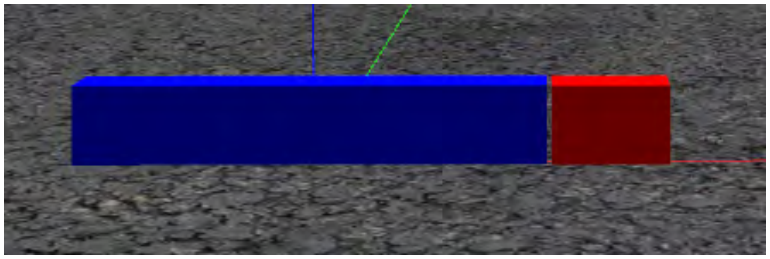
(e) Dependence of speed of UAV in coordinate z on time



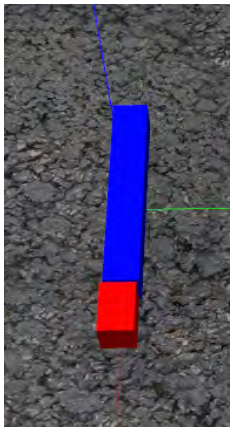
(f) Dependence of position z of UAV on time

Figure 5.11: The figure contains dependencies of velocities and positions in all coordinates on time, B-spline interpolation

From figure 5.11, we can see a change of position and velocity in dependence of time. The velocity and the position is measured by a sensor on a UAV (e.g. accelerometer, GPS), which means that the values can have some deviation.



(a) The final position of placed brick with optimal trajectory, view in coordinates x,z



(b) The final position of placed brick with optimal trajectory, view in coordinates x,y

Figure 5.12: The final position of placed brick with optimal trajectory, Catmull-Rom interpolation

The figure 5.12 shows the placed red brick next to the blue brick, which is the result of experiment, which is described by the above-mentioned figures. The deviation of placed brick is 0.013 for x coordinate and 0.014 for y coordinate.

In the figures 5.13 below is displayed a sequence of pictures from video, which is describing this experiment. As in the previous experiments, it wasn't possible to create the graphs and video simultaneously, that's why the video was created in the next run of simulation with the same parameters. The graphs and deviations of placed brick are not corresponding with the video. The whole video is available on: `https://www.youtube.com/watch?v=itcU1tpQaRQ`

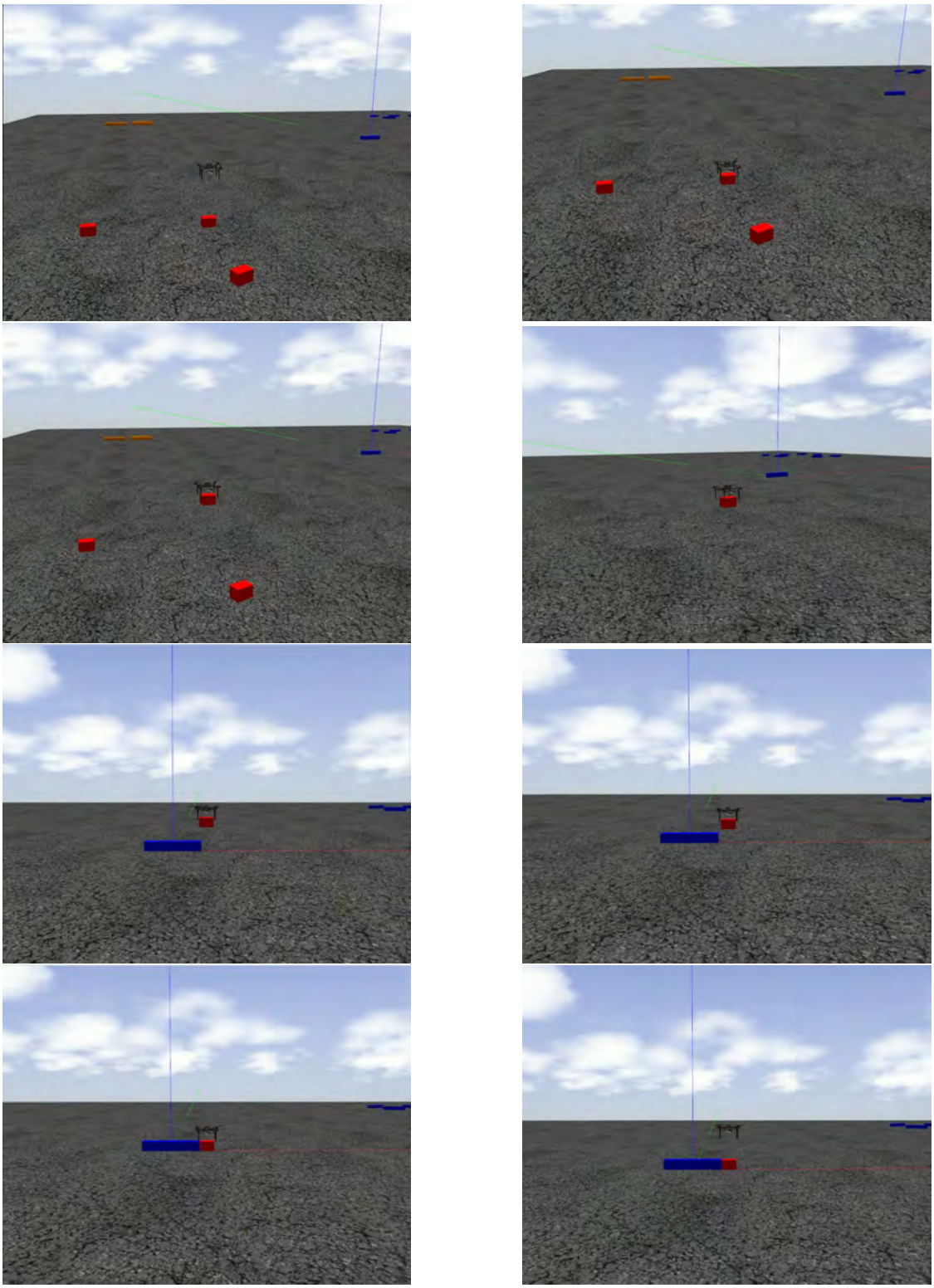


Figure 5.13: The figure displays a sequence of pictures from video, which is describing the experiment with a trajectory from B-spline interpolation

5.4 Experiments with all other types of trajectories by using optimal B-spline trajectory

For all experiments above was used only trajectory of type 2. In this section will be shown all other types of trajectories with using B-spline interpolation with optimal parameters found in section 5.3.

5.4.1 Trajectory with type 3

This type of trajectory is similar to type 2. The only difference between these types is in math sign. Type two is a curve in +axis x and type 3 is a curve in -axis x. The closer description is in 4.1b

Required position for this type of trajectory is $[-0.75, 0]$ and the sent message contains: `place position`: $[-0.75, 0.03, 0.05, 1.57]$, `final position`: $[-0.75, 0.03, 0.9, 1.57]$, `type of trajectory`: 3.

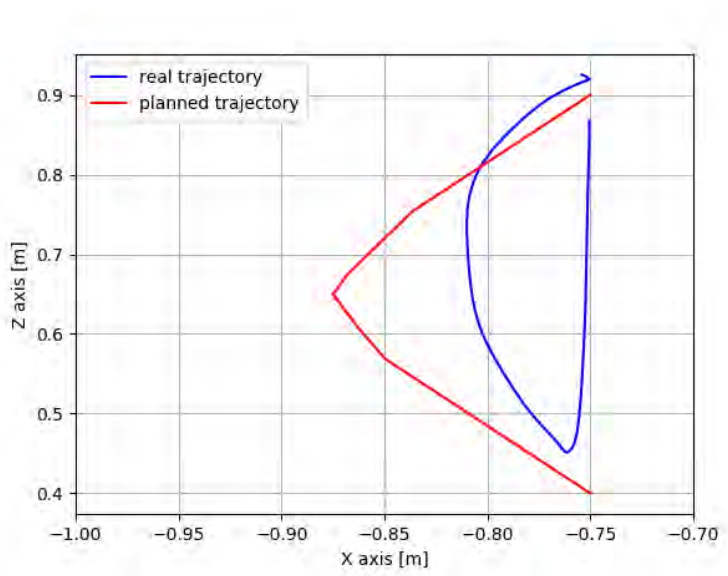
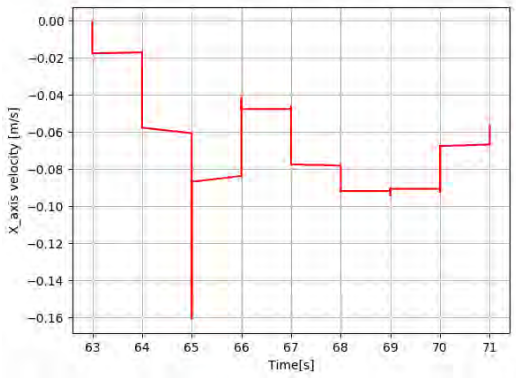
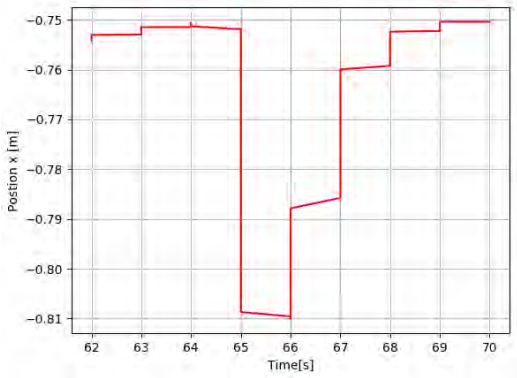


Figure 5.14: Planned and real trajectory in 2D, axis zx, B-spline interpolation, trajectory type 3

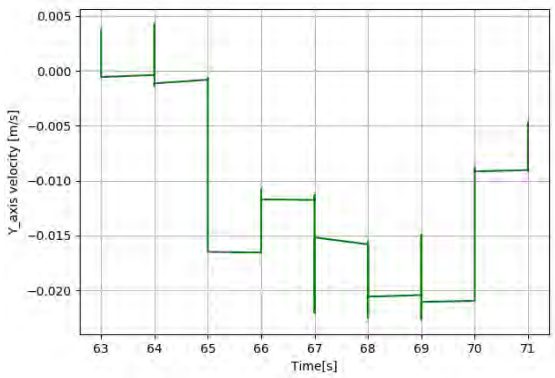
The trajectory in figure 5.14 is realized from beginning of `DROPPING1_STATE` to end of `RISE_STATE`. Red line is a planned trajectory (points sent to the MPC tracker) and blue line is a really executed trajectory (information from odometry, e.g. GPS). The 3D plots of this trajectory can be seen in Appendices. The MPC tracker again changing the trajectory so as the UAV is able to fly through the trajectory.



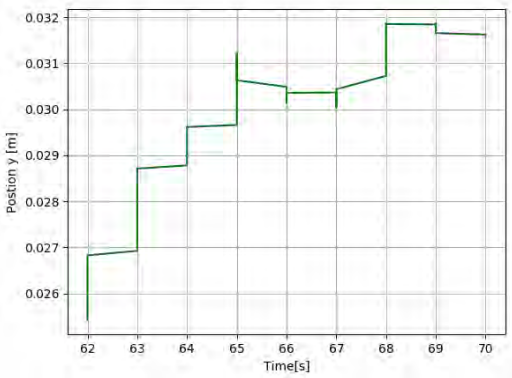
(a) Dependence of speed of UAV in coordinate x on time



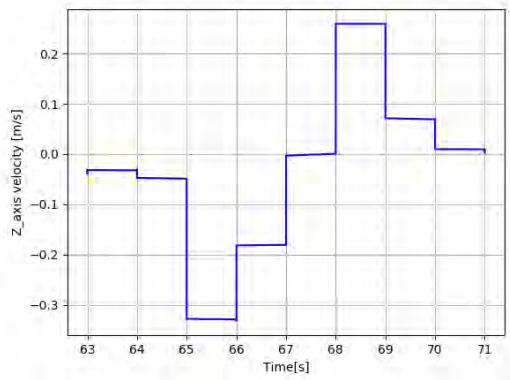
(b) Dependence of position x of UAV on time



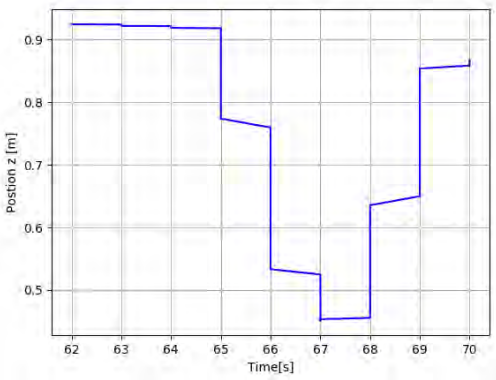
(c) Dependence of speed of UAV in coordinate y on time



(d) Dependence of position y of UAV on time



(e) Dependence of speed of UAV in coordinate z on time



(f) Dependence of position z of UAV on time

Figure 5.15: The figure contains dependencies of velocities and positions in all coordinates on time, B-spline interpolation, trajectory type 3

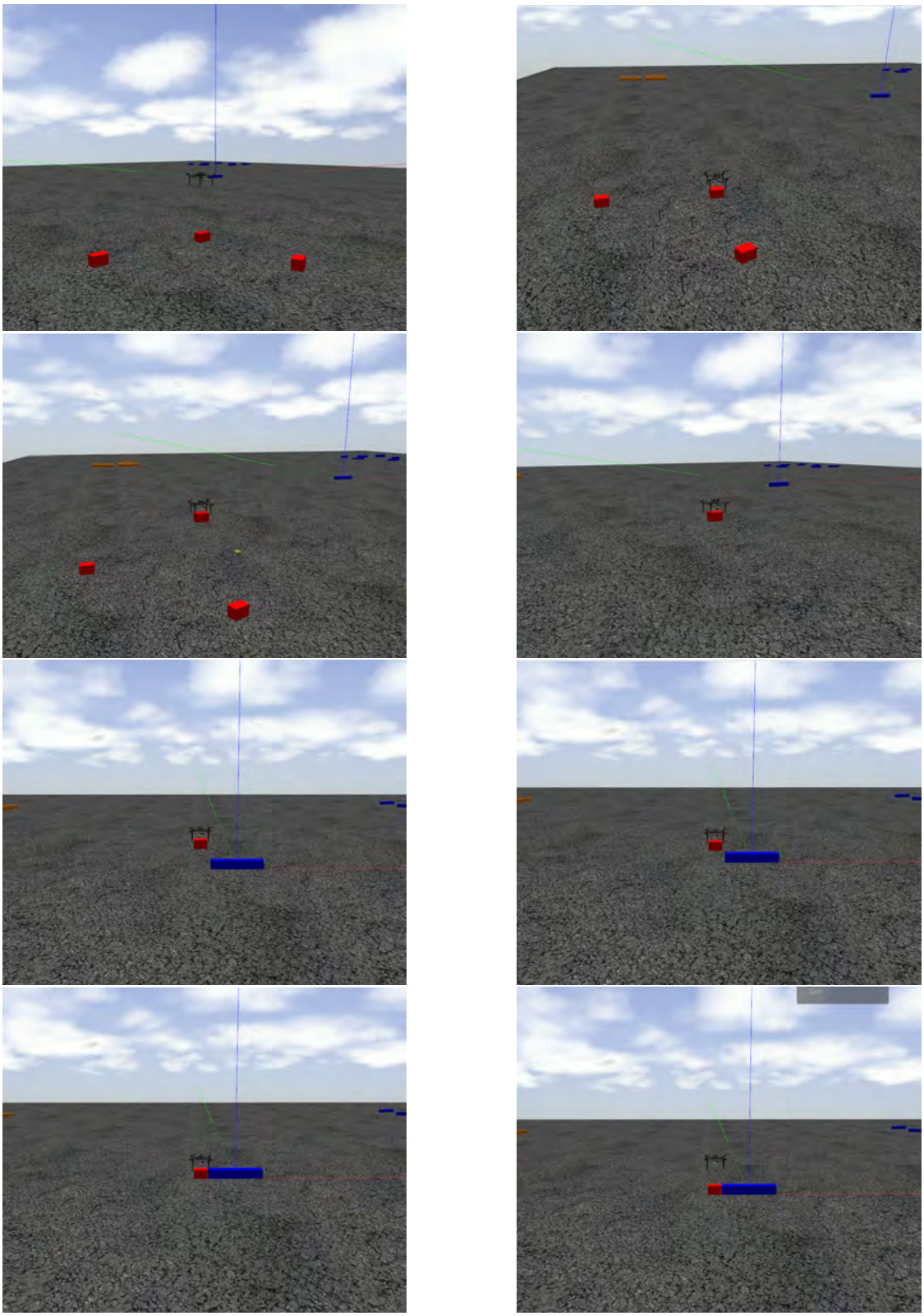


Figure 5.16: The figure displays a sequence of pictures from video, which is describing the experiment with a trajectory of type 3 from B-spline interpolation

In the figures 5.16 above is displayed a sequence of pictures from video, which is describing this experiment. The deviation of place brick is $[0.013, 0.005]$. As in the previous experiments, it wasn't possible to create the graphs and video simultaneously, that's why the video was created in the next run of simulation with the same parameters. The graphs are not corresponding with the video. The whole video is available on: `https://www.youtube.com/watch?v=tVhH71Km5u0`

5.4.2 Trajectory with type 4

Type four trajectory is a curve in +axis y. The closer description is in 4.1c

Required position for this type of trajectory is $[0, 0.25]$ and the sent message contains: `place position`: $[0.03, 0.25, 0.05, 0]$, `final position`: $[0.03, 0.25, 0.9, 0]$, `type of trajectory`: 4.

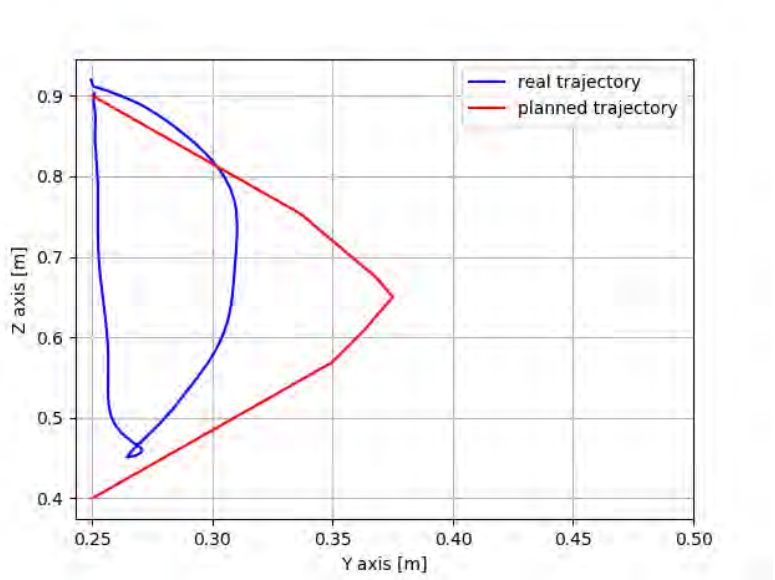
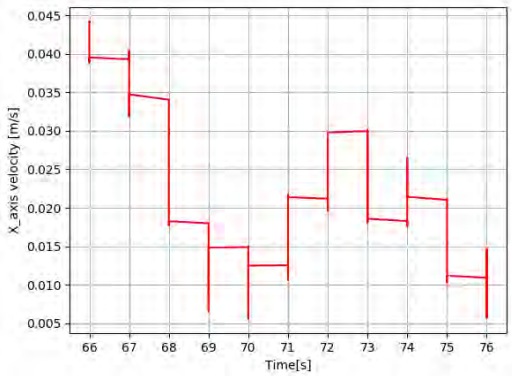


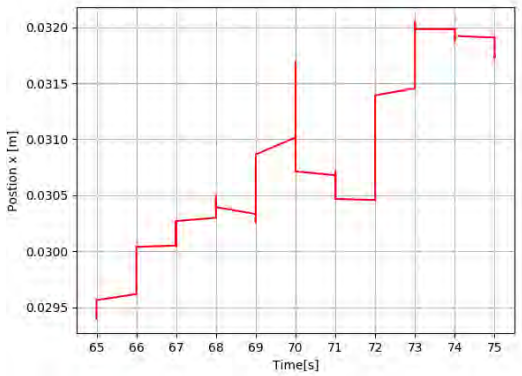
Figure 5.17: Planned and real trajectory in 2D, axis zy, B-spline interpolation, trajectory type 4

The trajectory in figure 5.17 is realized from beginning of `DROPPING1_STATE` to end of `RISE_STATE`. Red line is a planned trajectory (points sent to the MPC tracker) and blue line is a really executed trajectory (information from odometry, e.g. GPS). The 3D plot of these trajectories can be seen in Appendices. The MPC tracker again changing the trajectory so as the UAV is able to fly through the trajectory.

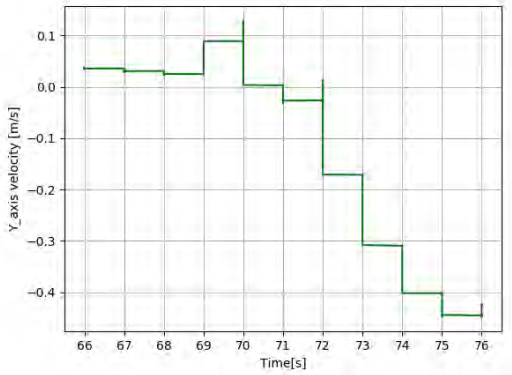
In the figure can be seen small loop on the bottom, which occurred during the damaging the wall. The "leg" of the UAV encountered to the wall and the already placed brick was partly shifted. The deviation of already placed brick (blue) is $[0, 0.007]$ and deviation of placed brick (red) is $[0.004, 0.011]$. The main reason for this error is described below 5.4.2.



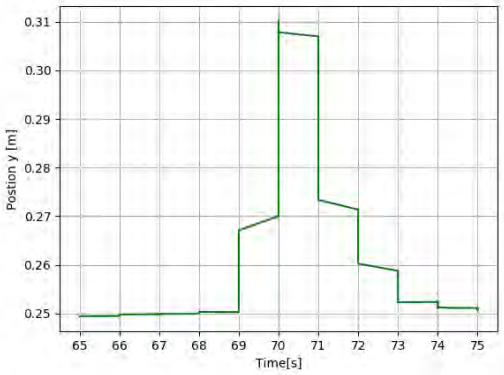
(a) Dependence of speed of UAV in coordinate x on time



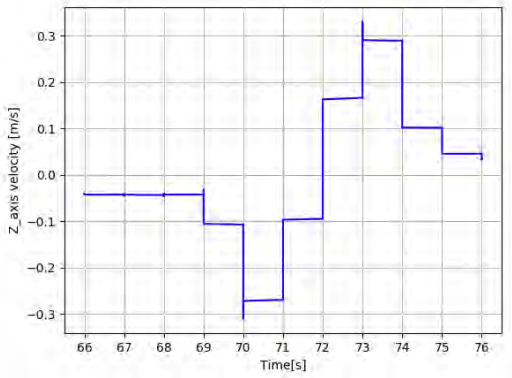
(b) Dependence of position x of UAV on time



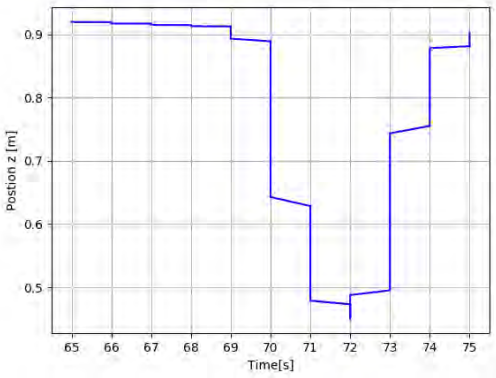
(c) Dependence of speed of UAV in coordinate y on time



(d) Dependence of position y of UAV on time



(e) Dependence of speed of UAV in coordinate z on time



(f) Dependence of position z of UAV on time

Figure 5.18: The figure contains dependencies of velocities and positions in all coordinates on time, B-spline interpolation, trajectory type 4

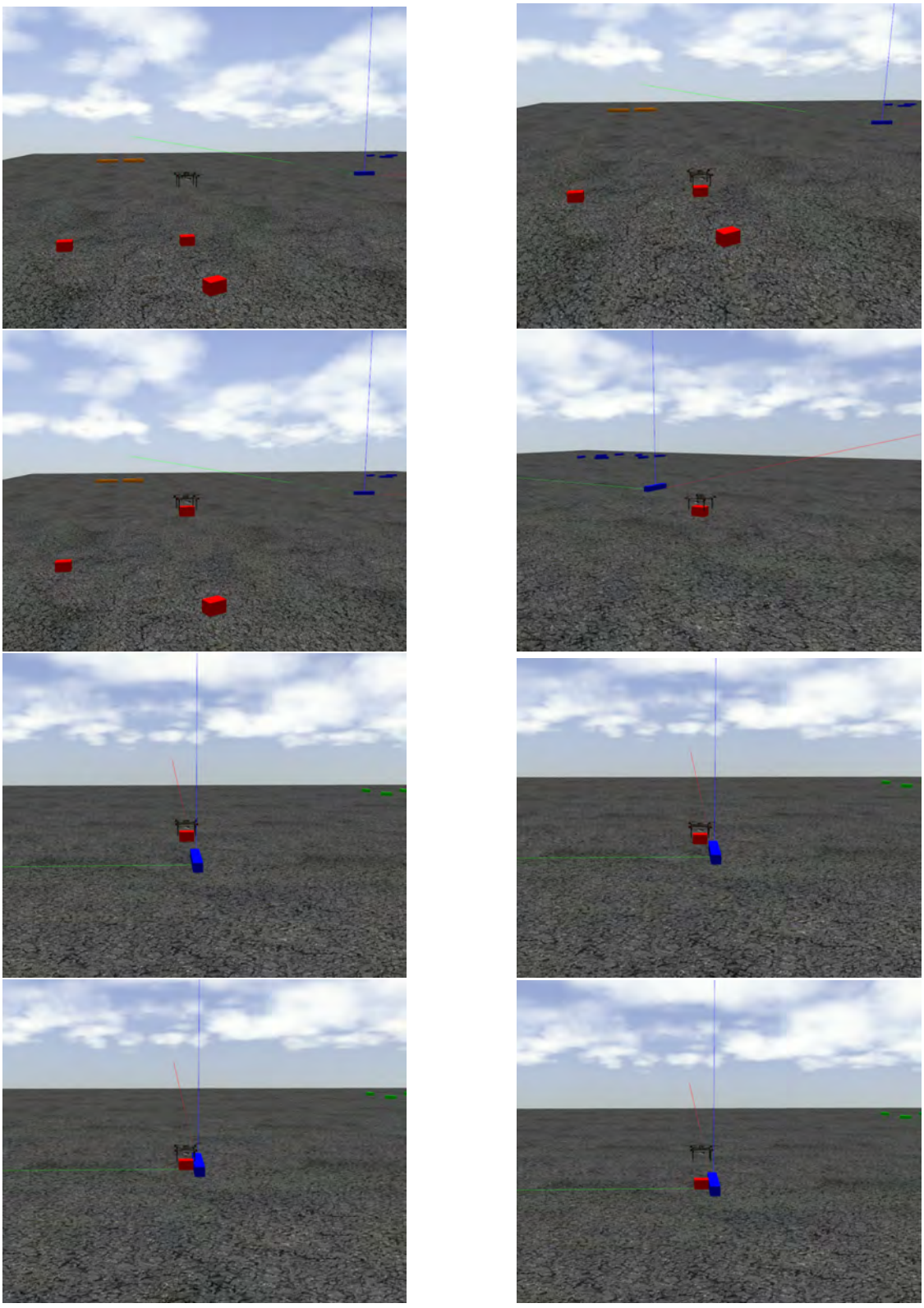


Figure 5.19: The figure displays a sequence of pictures from video, which is describing the experiment with a trajectory of type 4 from B-spline interpolation

In the figures 5.19 above is displayed a sequence of pictures from video, which is describing this experiment. As in the previous experiments, it wasn't possible to create the graphs and video simultaneously, that's why the video was created in the next run of simulation with the same parameters. The graphs are not corresponding with the video. The whole video is available on: `https://www.youtube.com/watch?v=P3XRGrC0gBc`

Unfortunately, the error occurred only in the first run of the experiment and can't be seen in the video. As mentioned, the error occurred by encounter a "leg" of the UAV to the wall, therefore it is possible to safely place the bricks only in one straight line in the simulator. This error can be solved by "longer" gripper, so the "legs" can't encounter the wall or the brick could be dropped from a higher altitude. The second solution will much decrease the precision of the placing the brick.

5.4.3 Trajectory with type 5

Type five trajectory is a curve in -axis y. The closer description is in 4.1d

Required position for this type of trajectory is $[0, -0.75]$ and the sent message contains: `place position`: $[0.03, -0.75, 0.05, 0]$, `final position`: $[0.03, -0.75, 0.9, 0]$, `type of trajectory`: 5.

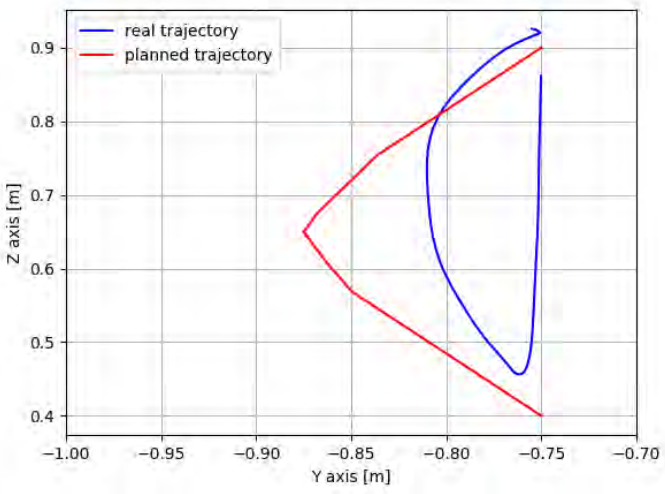
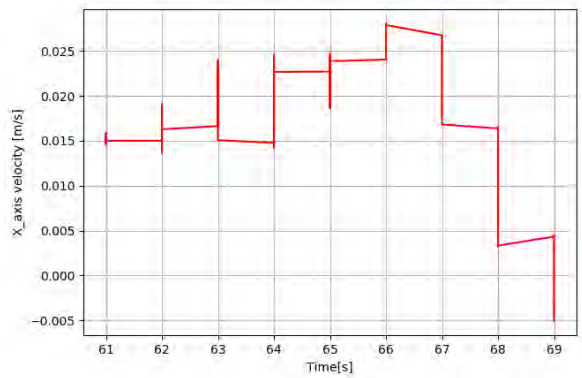
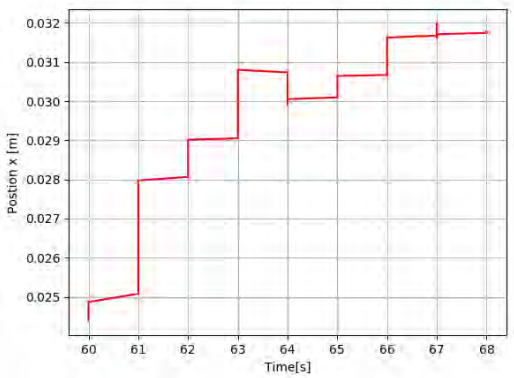


Figure 5.20: Planned and real trajectory in 2D, axis zy, B-spline interpolation, trajectory type 5

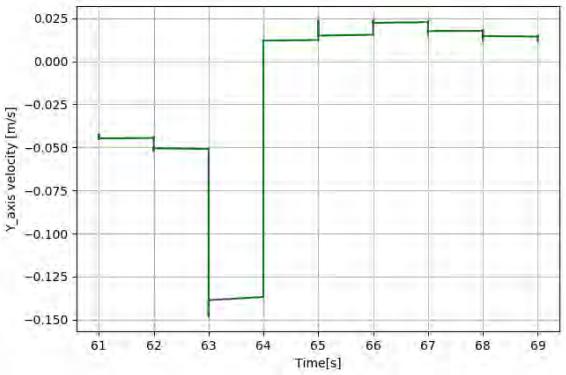
The trajectory in figure 5.20 is realized from beginning of `DROPPING1_STATE` to end of `RISE_STATE`. Red line is a planned trajectory (points sent to the MPC tracker) and blue line is a really executed trajectory. The 3D plots of this trajectory can be seen in Appendices. The MPC tracker again changing the trajectory so as the UAV is able to fly through the trajectory.



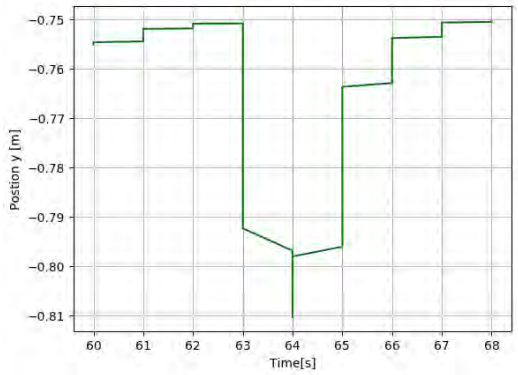
(a) Dependence of speed of UAV in coordinate x on time



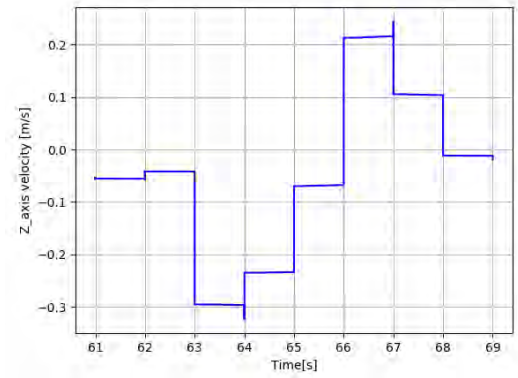
(b) Dependence of position x of UAV on time



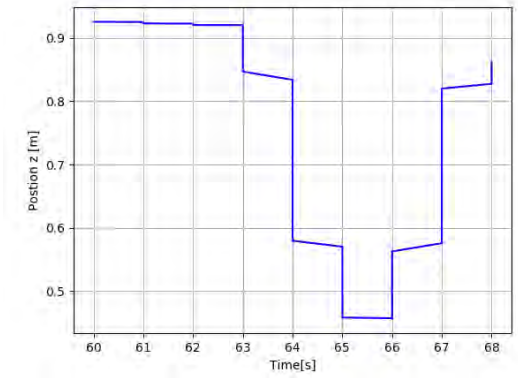
(c) Dependence of speed of UAV in coordinate y on time



(d) Dependence of position y of UAV on time



(e) Dependence of speed of UAV in coordinate z on time



(f) Dependence of position z of UAV on time

Figure 5.21: The figure contains dependencies of velocities and positions in all coordinates on time, B-spline interpolation, trajectory type 5

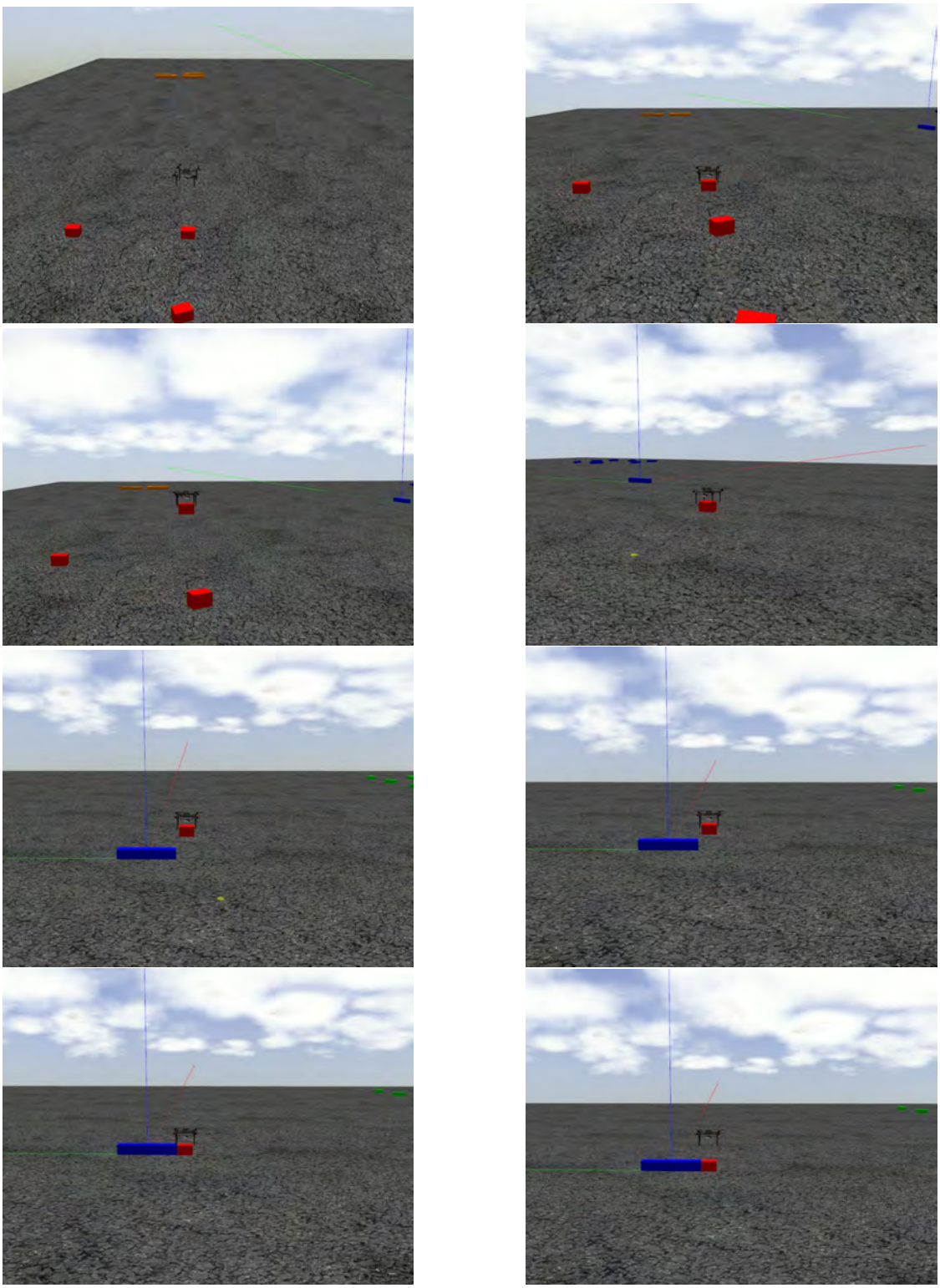


Figure 5.22: The figure displays a sequence of pictures from video, which is describing the experiment with a trajectory of type 5 from B-spline interpolation

In the figures 5.22 above is displayed a sequence of pictures from video, which is describing this experiment. The deviations of placed brick is $[0.005, 0.012]$. As in the previous experiments, it wasn't possible to create the graphs and video simultaneously, that's why the video was created in the next run of simulation with the same parameters. The graphs are not corresponding with the video. The whole video is available on: `https://www.youtube.com/watch?v=yN-YYa9zlYU`

5.4.4 Trajectory with type 1

Type one trajectory is a straight trajectory. The closer description is in 4.2a and 4.2b.

Required position for this type of trajectory is $[0, 0]$ and the sent message contains: `place position`: $[0, 0.03, 0.2, 1.57]$, `final position`: $[0, 0.03, 1.2, 1.57]$, `type of trajectory`: 1.

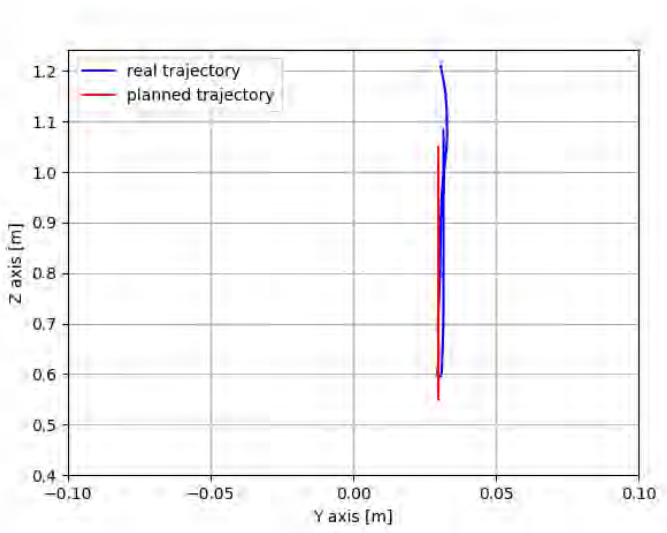
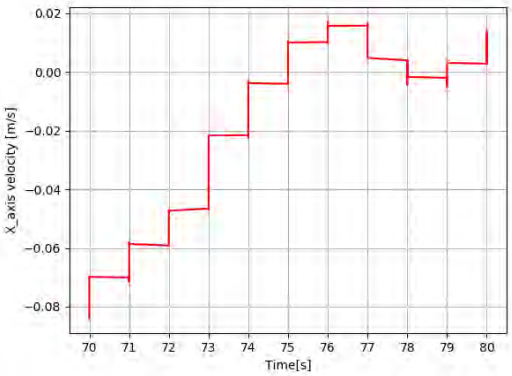
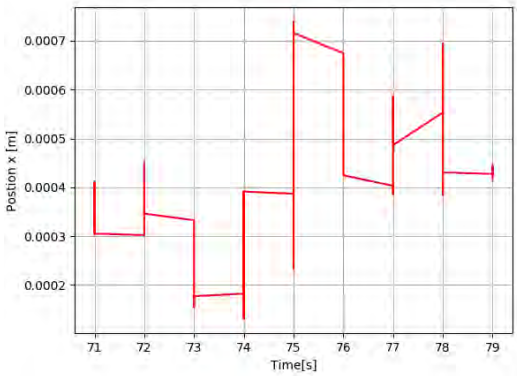


Figure 5.23: Planned and real trajectory in 2D, axis zy, B-spline interpolation, trajectory type 1

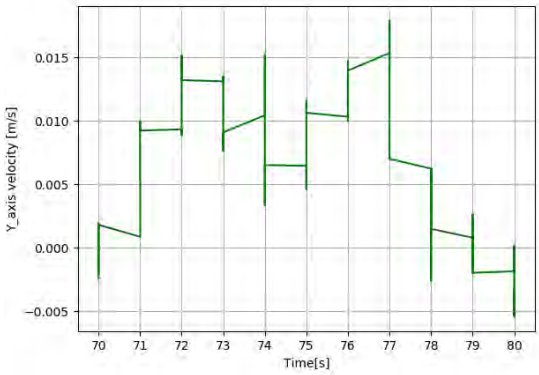
The trajectory in figure 5.23 is realized from beginning of `DROPPING1_STATE` to end of `RISE_STATE`. Red line is a planned trajectory (points sent to the MPC tracker) and blue line is a really executed trajectory. The 3D plot of this trajectory can be seen in Appendices. The MPC tracker again changing the trajectory so as the UAV is able to fly through the trajectory.



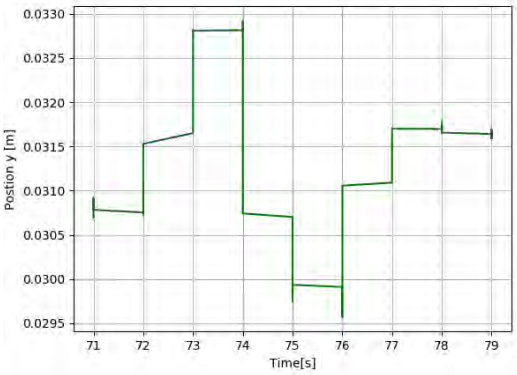
(a) Dependence of speed of UAV in coordinate x on time



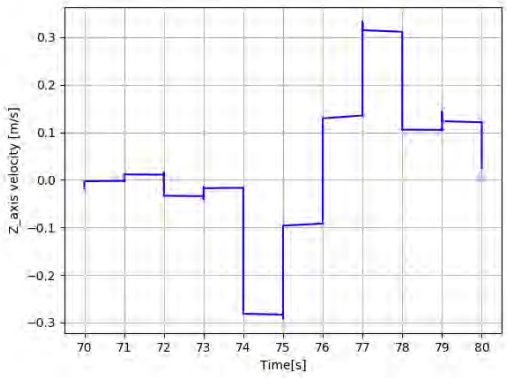
(b) Dependence of position x of UAV on time



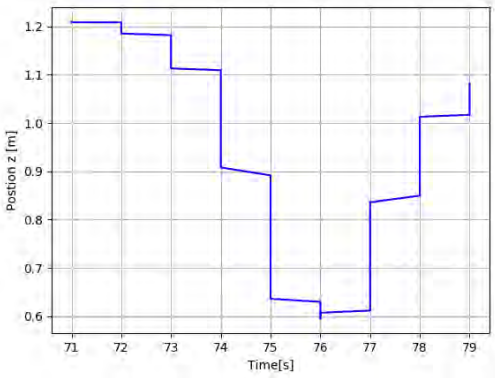
(c) Dependence of speed of UAV in coordinate y on time



(d) Dependence of position y of UAV on time



(e) Dependence of speed of UAV in coordinate z on time



(f) Dependence of position z of UAV on time

Figure 5.24: The figure contains dependencies of velocities and positions in all coordinates on time, B-spline interpolation, trajectory type 1

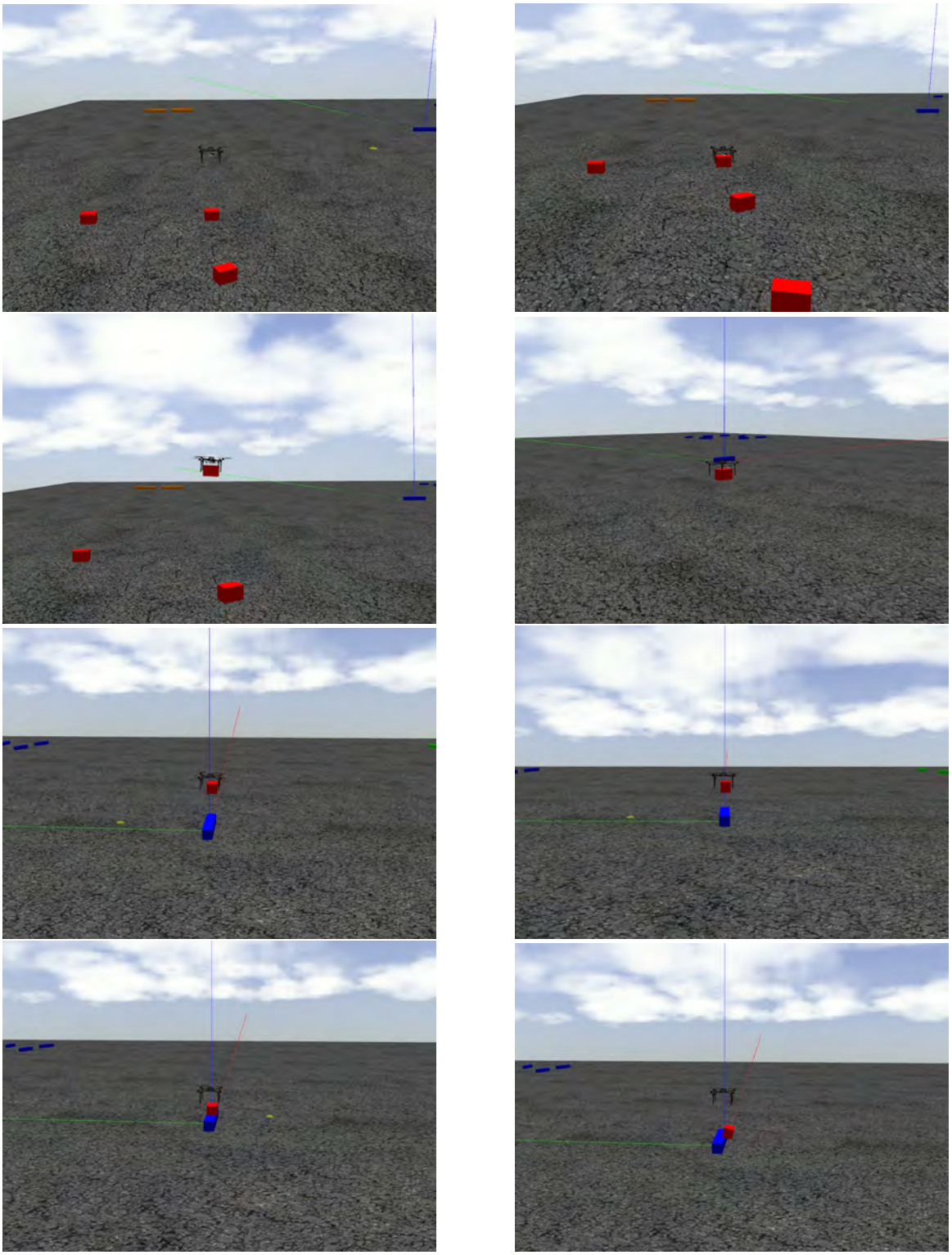


Figure 5.25: The figure displays a sequence of pictures from video, which is describing the experiment with a trajectory of type 1 from B-spline interpolation

In the figures 5.25 above is displayed a sequence of pictures from video, which is describing this experiment. The deviations of placed brick is $[0.004, 0.046]$. As in the previous experiments, it wasn't possible to create the graphs and video simultaneously, that's why the video was created in the next run of simulation with the same parameters. The graphs are not corresponding with the video. The whole video is available on: `https://www.youtube.com/watch?v=d2NpvDsH4UM`

The straight trajectory was tested only for place brick on the new "floor" (place brick without bricks) 4.2a. The second usage of the trajectory 4.2b wasn't tested, because the state of the wall couldn't occur with good high-level motion planning.

The brick slips down from the wall in the simulator. This error can be seen in the video or on the last picture in figure 5.25. The error causes a bigger deviation in axis y and damaging of the wall (blue brick). The error is caused by the Gazebo simulator or by the created world (bad physics or material of the bricks). In real experiments, the error won't occur with high probability.

Chapter 6

Conclusion

The motion planning method for grasping and placing the color bricks is presented in this work. The grasping method described in chapter 3 is able to grasp any the brick if the `fake object detector` substitutes information from the camera. After the software for recognizing the brick was created, the method is able to grasp any brick with 80% success. The method was upload on git for mbzirc competition and the method was upgraded by members of Multi-robot System group since that. The node was split into two nodes grasping and estimation. Now, this method is grasping bricks with 100% success and the brick is attached only with small deviations in x,y coordinates. This method is used for the task in mbzirc competition.

The method for placing the bricks, described in chapter 4, contains five types of trajectories, where each trajectory is created for precise fitting the brick into the wall depending on state of the wall. The shape of these trajectories can be easily modified by parameters according to the requirements. The curve trajectory can be created by three different interpolations (B-spline, Catmull-Rom and Hermite). The all mentioned types of trajectories are working similarly and have similar precision.

The different interpolation almost doesn't have an influence on our simple curve trajectory created by five points with similar gaps. If the parameters are the same, then the experiments have similar results for all mentioned interpolations. The small difference between result is probably caused by grasping the brick.

During experiments was found error when trying to place the brick from the side (not directly in one straight line). This error can be seen in section 5.4.2. The "leg" of the UAV encountered the wall and damaged the wall. The error can be solved by using longer gripper, so the brick will be attached under UAV and the "leg" can't encounter the wall. The other solution is to drop bricks from higher altitude, but this solution decrease the precision of placing the brick.

Now, if the optimal parameters are chosen then deviation from the required position is around 1.5 cm. But, if the complete fitting of the bricks (bricks are touching each other)

is required then it is needed to change the condition of releasing the brick. It will be needed to shift the required position of the brick "into the wall" so as the brick will touch the wall during placing. Then, it will be needed to release the brick according to some pressure on the UAV. This solution includes a more complicated approach to the problem.

Bibliography

[1] D. F. AL-MASKARI. Mbzirc 2020. [Online]. Available: https://www.mbzirc.com/challenge/2020

[2] M. Sakin and Y. C. Kiroglu, "3d printing of buildings: Construction of the sustainable houses of the future by bim," *Energy Procedia*, vol. 134, pp. 702 – 711, 2017, sustainability in Energy and Buildings 2017: Proceedings of the Ninth KES International Conference, Chania, Greece, 5-7 July 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1876610217346969

[3] X. Zhang, M. Li, J. H. Lim, Y. Weng, Y. W. D. Tay, H. Pham, and Q.-C. Pham, "Large-scale 3d printing by a team of mobile robots," *Automation in Construction*, vol. 95, pp. 98 – 106, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0926580518304011

[4] V. Spurný, T. Báča, M. Saska, R. Pěnička, T. Krajník, J. Thomas, D. Thakur, G. Loianno, and V. Kumar, "Cooperative autonomous search, grasping, and delivering in a treasure hunt scenario by a team of unmanned aerial vehicles," *Journal of Field Robotics*, 10 2018.

[5] J. G. Everett and A. H. Slocum, "Automation and robotics opportunities: Construction versus manufacturing," *Journal of Construction Engineering and Management*, vol. 120, no. 2, pp. 443–452, 1994. [Online]. Available: https://ascelibrary.org/doi/abs/10.1061/%28ASCE%290733-9364%281994%29120%3A2%28443%29

[6] H. Ardiny, S. Witwicki, and F. Mondada, "Construction automation with autonomous mobile robots: A review," in *2015 3rd RSI International Conference on Robotics and Mechatronics (ICROM)*, Oct 2015, pp. 418–424.

[7] K. S. Saidi, T. Bock, and C. Georgoulas, *Robotics in Construction*. Cham: Springer International Publishing, 2016, pp. 1493–1520. [Online]. Available: https://doi.org/10.1007/978-3-319-32552-1_57

[8] K. Dörfler, T. Sandy, M. Giftthaler, F. Gramazio, M. Kohler, and J. Buchli, *Mobile Robotic Brickwork*. Cham: Springer International Publishing, 2016, pp. 204–217. [Online]. Available: https://doi.org/10.1007/978-3-319-26378-6_15

[9] T. Bruckmann, H. Mattern, A. Spengler, C. Reichert, A. Malkwitz, and M. Konig, "Automated construction of masonry buildings using cable-driven parallel robots," in *ISARC. Proceedings of the International Symposium on Automation and Robotics in Construction*, vol. 33. Vilnius Gediminas Technical University, Department of Construction Economics ..., 2016, p. 1.

[10] H. Mattern, T. Bruckmann, A. Spengler, and M. Konig, "Simulation of automated construction using wire robots," in *Proceedings of the 2016 Winter Simulation Conference*, ser. WSC '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 3302–3313. [Online]. Available: http://dl.acm.org/citation.cfm?id=3042409.3042503

[11] J. P. Sousa, C. G. Palop, E. Moreira, A. M. Pinto, J. Lima, P. Costa, P. Costa, G. Veiga, and A. Paulo Moreira, *The SPIDERobot: A Cable-Robot System for On-site Construction in Architecture*. Cham: Springer International Publishing, 2016, pp. 230–239. [Online]. Available: https://doi.org/10.1007/978-3-319-26378-6_17

[12] S. Kalantari, A. T. Becker, and R. Ike, "Designing for digital assembly with a construction team of mobile robots." ACADIA, 2018.

[13] Y. Ham, K. K. Han, J. J. Lin, and M. Golparvar-Fard, "Visual monitoring of civil infrastructure systems via camera-equipped unmanned aerial vehicles (uavs): a review of related works," *Visualization in Engineering*, vol. 4, no. 1, p. 1, 2016.

[14] J. Chudoba, M. Kulich, M. Saska, T. Báča, and L. Přeučil, "Exploration and mapping technique suited for visual-features based localization of mavs," *Journal of Intelligent & Robotic Systems.*, vol. 84, no. 1, pp. 351–369, 2016.

[15] J. Faigl, P. Váňa, R. Pěnička, and M. Saska, "Unsupervised learning-based flexible framework for surveillance planning with aerial vehicles," *Journal of Field Robotics*, vol. 36, no. 1, pp. 270–301, 2019. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/rob.21823

[16] D. Zahrádka, R. Pěnička, and M. Saska, "Route planning for teams of unmanned aerial vehicles using dubins vehicle model with budget constraint," in *Modelling and Simulation for Autonomous Systems*, J. Mazal, Ed. Cham: Springer International Publishing, 2019, pp. 365–389.

[17] P. Štěpán, T. Krajnik, M. Petrlik, and M. Saska, "Vision techniques for on-board detection, following and mapping of moving targets," *Journal of Field Robotics*, vol. 36, no. 1, pp. 252–269, 2019.

[18] G. Loianno, V. Spurny, T. Baca, J. Thomas, D. Thakur, T. Krajnik, A. Zhou, A. Cho, M. Saska, and V. Kumar, "Localization, grasping, and transportation of magnetic objects by a team of mavs in challenging desert like environments," *IEEE Robotics and Automation Letters*, vol. 3, no. 3, pp. 1576–1583, 2018.

[19] V. Walter, T. Novák, and M. Saska, "Self-localization of unmanned aerial vehicles based on optical flow in onboard camera images," in *Lecture Notes in Computer Science, vol 10756.* Cham: Springer International Publishing, 2018.

[20] W. Giernacki, D. Horla, T. Báča, and M. Saska, "Real-time model-free minimum-seeking autotuning method for unmanned aerial vehicle controllers based on fibonacci-search algorithm," *Sensors*, vol. 19, pp. 1–30, 01 2019.

[21] R. support. Ros forum. [Online]. Available: https://answers.ros.org

[22] T. Baca, P. Stepan, V. Spurny, M. Saska, J. Thomas, G. Loianno, and V. Kumar, "Autonomous landing on a moving vehicle with an unmanned aerial vehicle," *Journal of Field Robotics - online first*, 2019.

[23] T. Baca, P. Stepan, and M. Saska, "Autonomous landing on a moving car with unmanned aerial vehicle," in *2017 European Conference on Mobile Robots (ECMR)*, Sep. 2017, pp. 1–6.

[24] M. Nemec and D. Suster, "uav localization core," https://mrs.felk.cvut.cz/gitlab/nemecm43/uav-localization-core, 2020, [Online; accessed 5-January-2020].

[25] Wikipedia contributors, "Ramer–douglas–peucker algorithm — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Ramer%E2%80%93Douglas%E2%80%93Peucker_algorithm&oldid=931337557, 2019, [Online; accessed 18-December-2019].

[26] T. Baca, D. Hert, G. Loianno, M. Saska, and V. Kumar, "Model predictive trajectory tracking and collision avoidance for reliable outdoor deployment of unmanned aerial vehicles," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2018.

Appendices

CD Content

In Table 1 are listed names of all root directories on CD.

Directory name	**Description**
thesis	the thesis in pdf format
thesis_sources	latex source codes
source_codes	program source codes
videos	videos from experiments in simulator
trajectory bags	data sets for graphs and python code for print

Table 1: CD Content

List of abbreviations

In Table 2 are listed abbreviations used in this thesis.

Abbreviation	**Meaning**
MBZIRC	Mohamed Bin Zayed International Robotics Challenge
GPS	Global Positioning System
RTK	Real-time kinematic
MPC	model predictive controller
UAV	Unmanned Aerial Vehicle
UGV	Unmanned Ground Vehicle
ROS	Robot Operating System
No.	Number of experiment

Table 2: Lists of abbreviations

3D graphs with planned and real trajectories

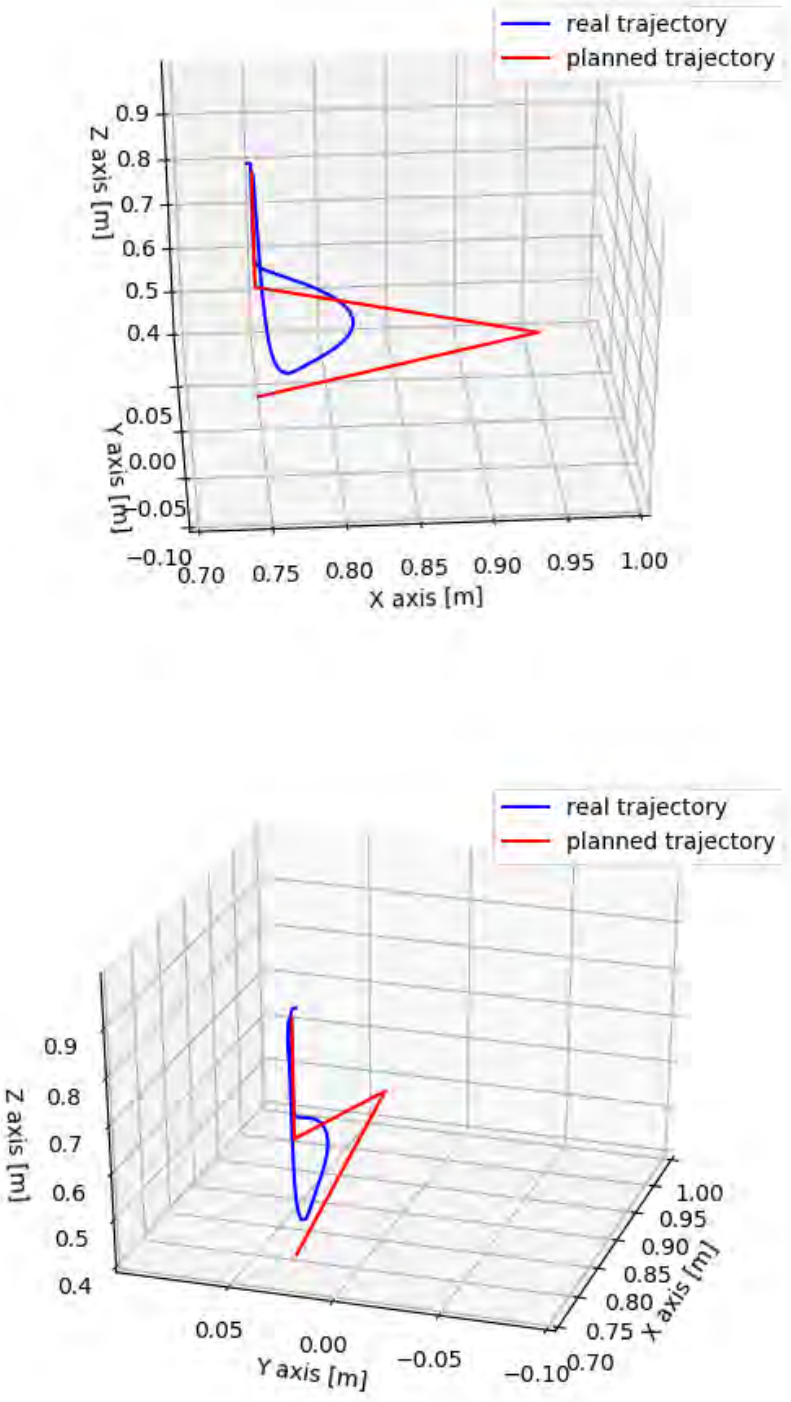


Figure 1: Planned and real trajectory in 3D, Hermite

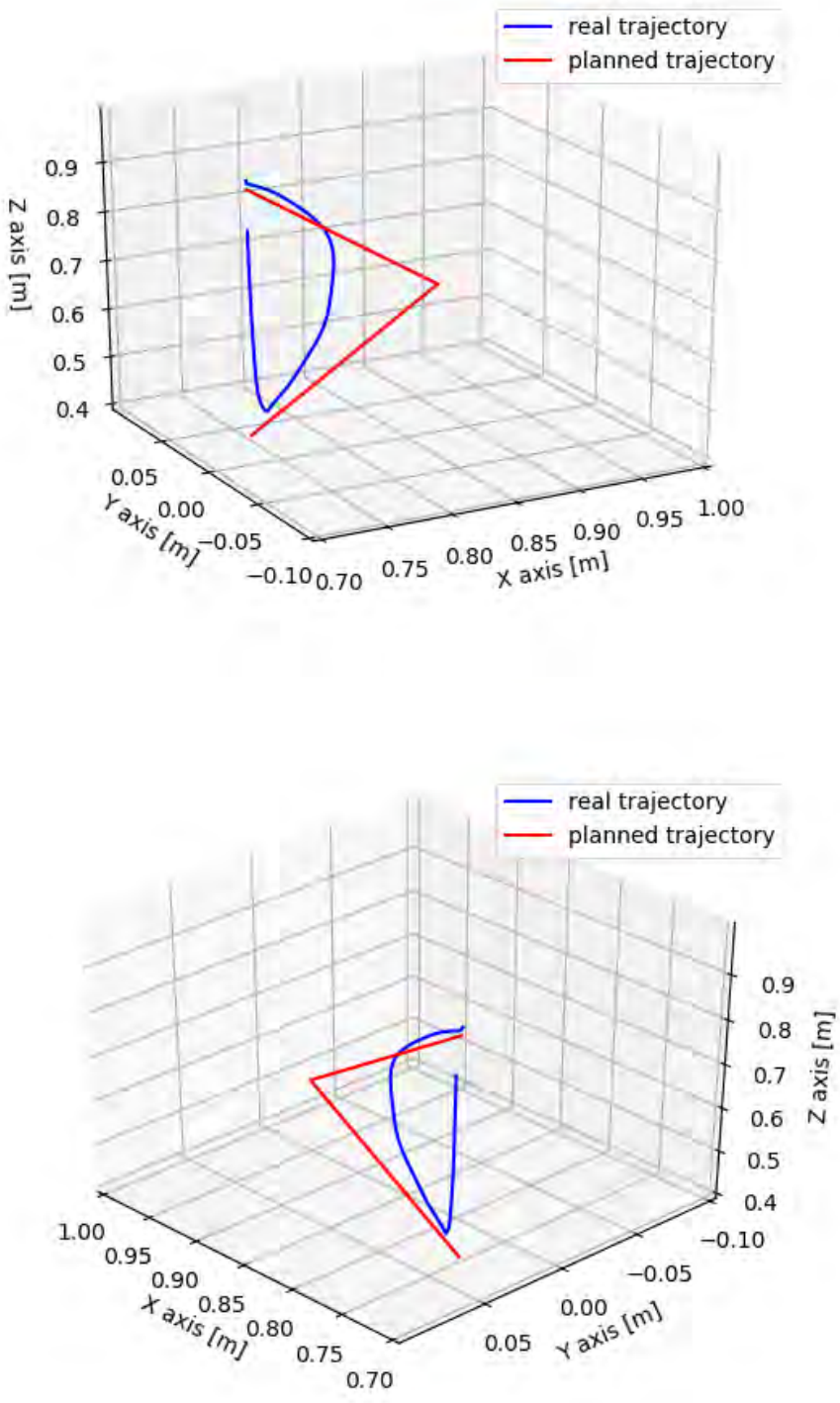


Figure 2: Planned and real trajectory in 3D, Catmull

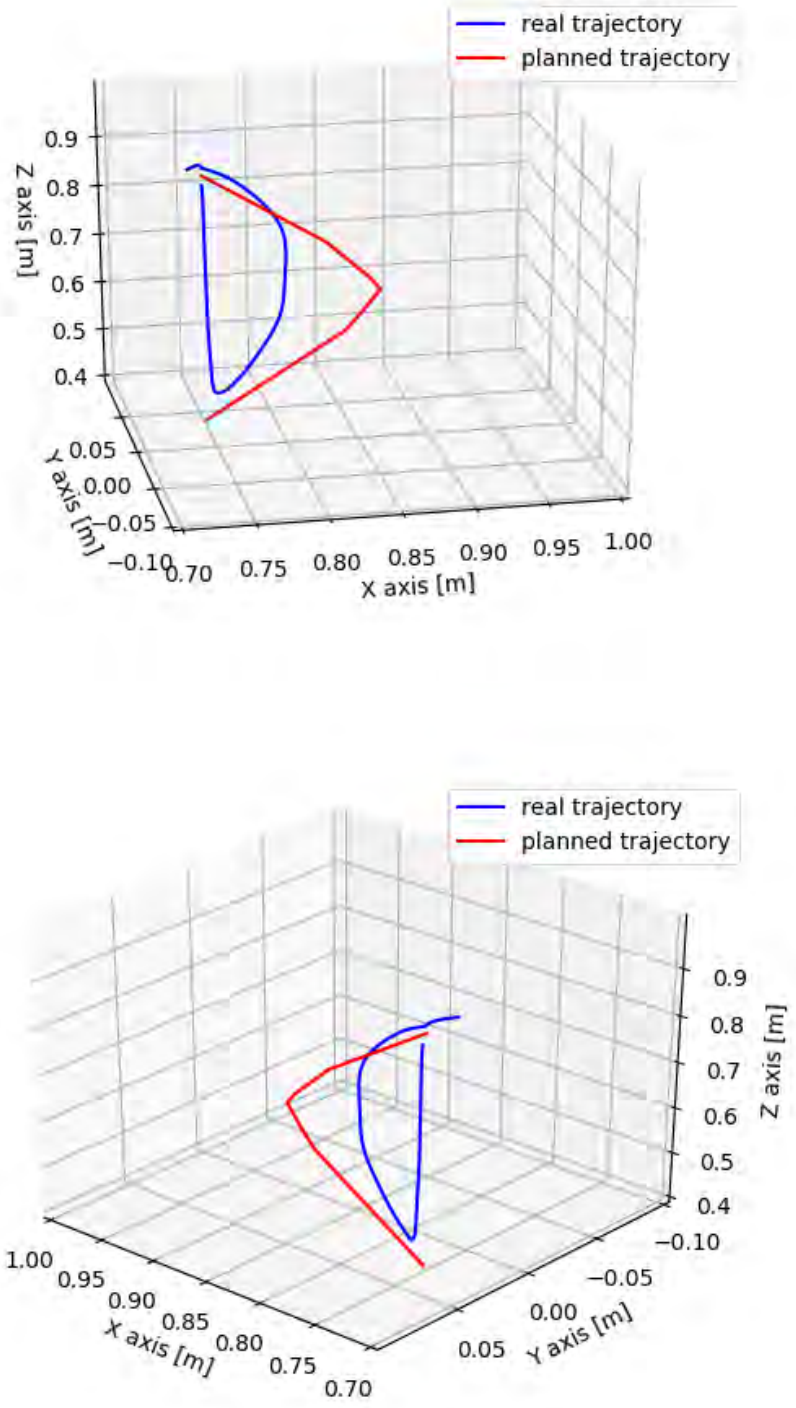


Figure 3: Planned and real trajectory in 3D, B-spline

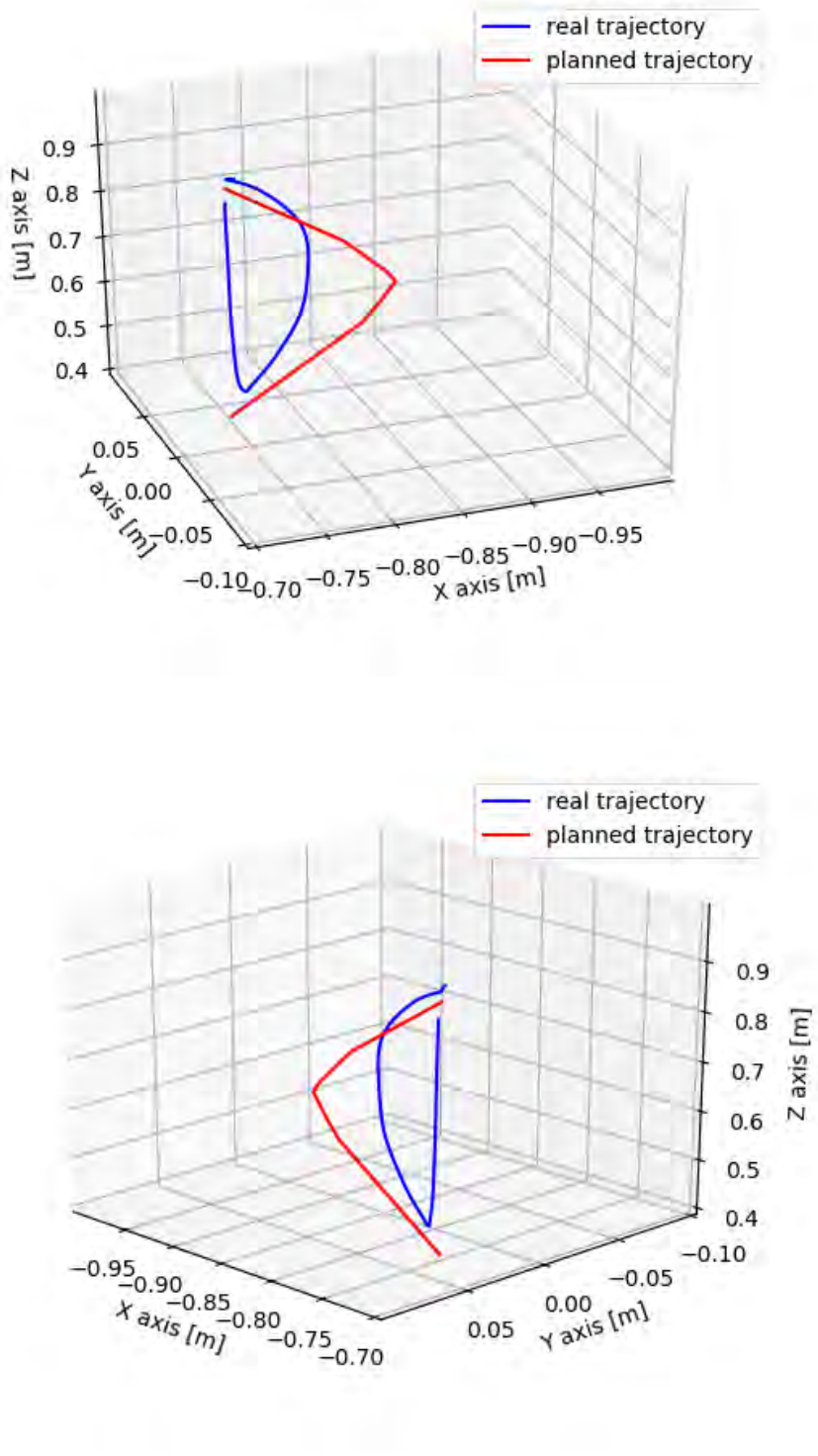


Figure 4: Planned and real trajectory in 3D, type 3 trajectory

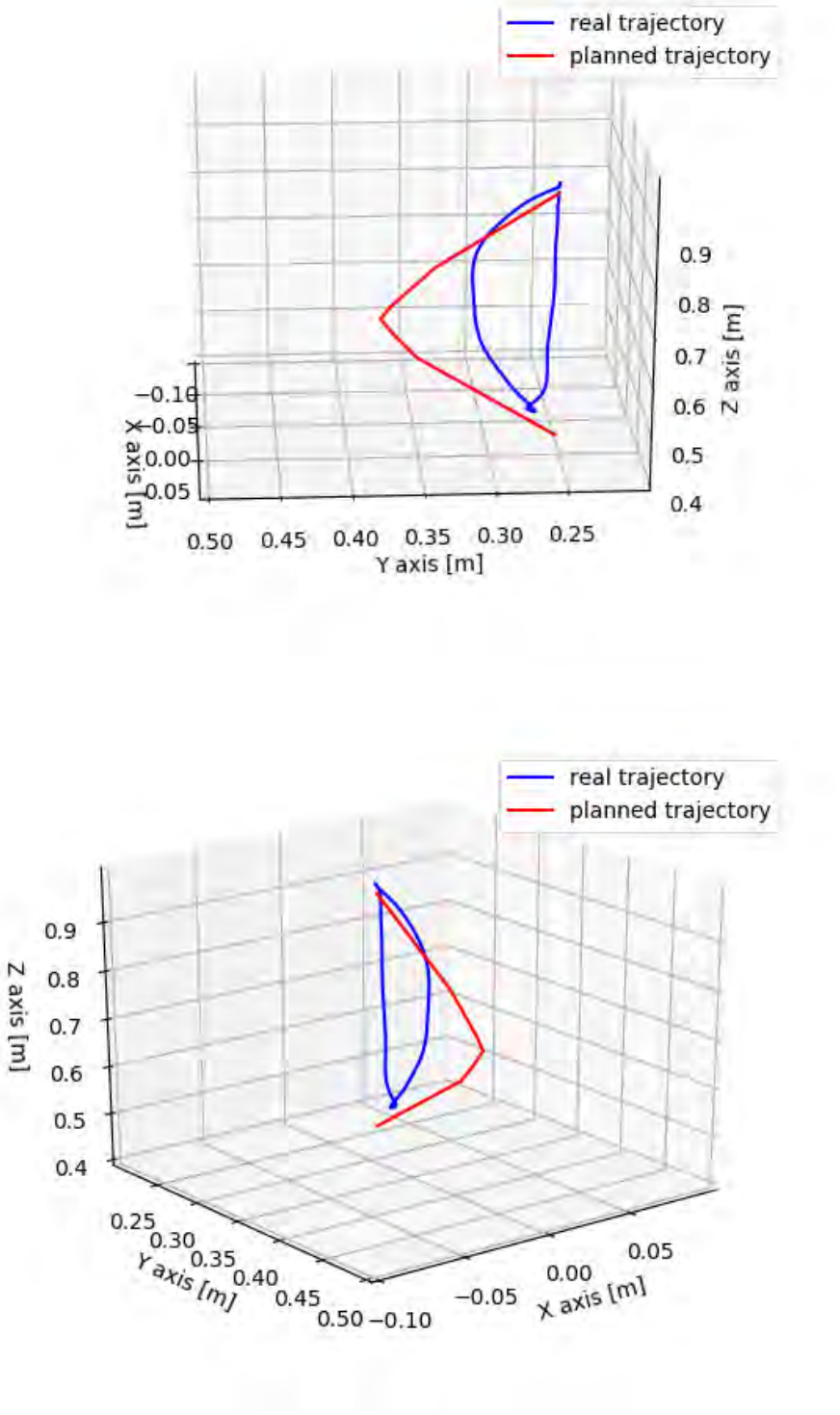


Figure 5: Planned and real trajectory in 3D, type 4 trajectory

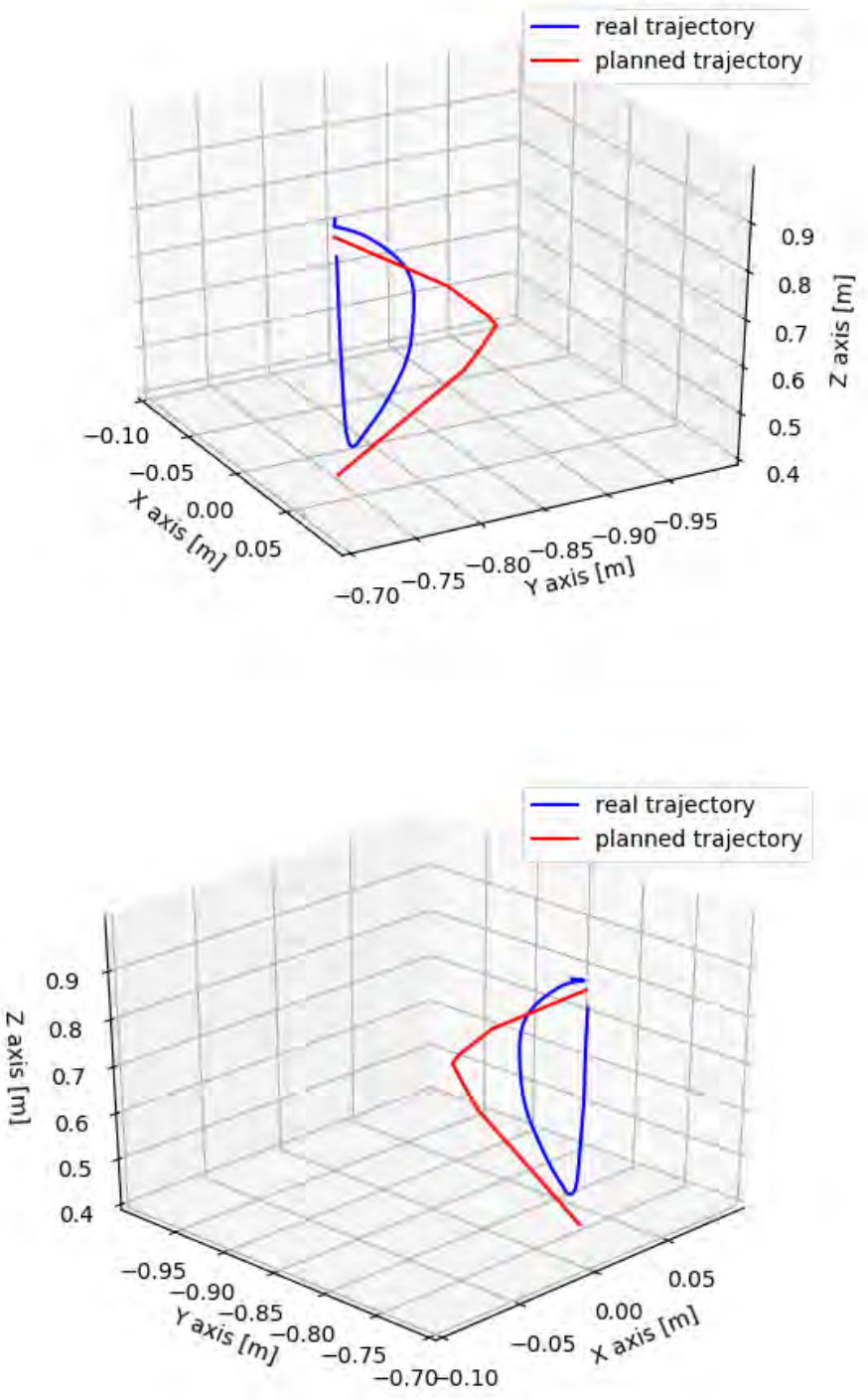


Figure 6: Planned and real trajectory in 3D, type 5 trajectory

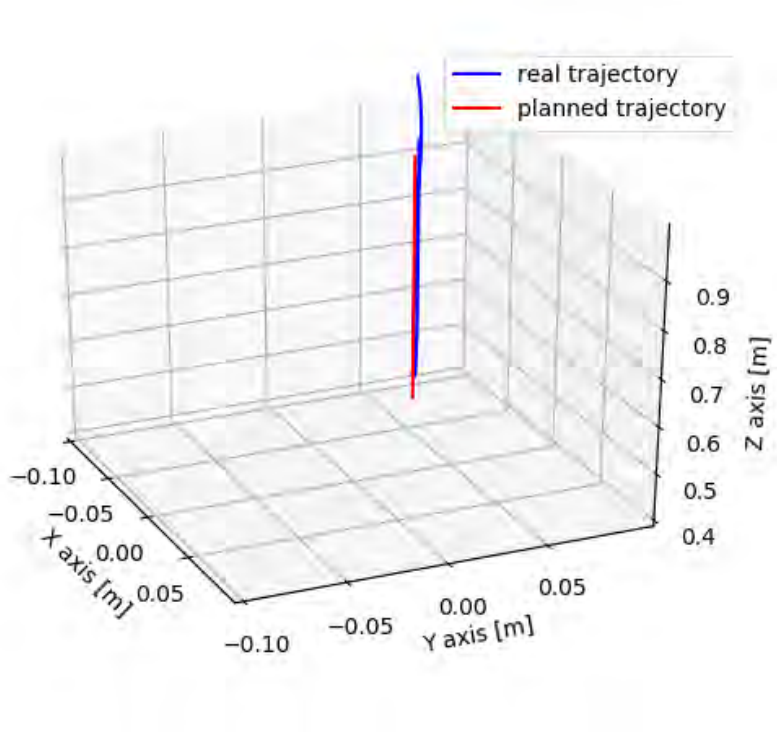


Figure 7: Planned and real trajectory in 3D, type 1 trajectory