Real-Time Optimistic Planning for the Control of Nonlinear Systems

M.H. Wensveen
Real-Time Optimistic Planning for the Control of Nonlinear Systems

MASTER OF SCIENCE THESIS

For the degree of Master of Science in Electrical Engineering at Delft University of Technology

M.H. Wensveen

February 24, 2014
Abstract

Optimistic Planning is a model-based online planning algorithm that guarantees near-optimal actions for the control of arbitrarily nonlinear systems. Planning algorithms aim to find optimal actions by starting from the current state and developing a tree representation of sequences of actions and resulting states, using a model to simulate state-transitions. Typically, online planning algorithms return a sequence of actions, apply the first action (or several actions at the start of the sequence) and start planning again from the new state, resembling the receding horizon principle as seen in Model Predictive Control.

Several optimistic planning algorithms exist, of which in this work only Optimistic Planning for Deterministic systems (OPD) is considered. OPD works for large, possibly infinite state spaces, but only for finite, discrete action spaces. Unfortunately, while OPD shows good theoretical near-optimality guarantees, there is no record yet of OPD being applied to control nonlinear physical systems in real-time. This is because of the (long) computation required by OPD.

This work analyzes two main methods that can be used to make OPD suitable for real-time applications. The first approach is to increase the computational speed of the planning process by parallelizing the algorithm. Unfortunately, while parallelization has been proven to be able to increase the computational speed in classical planning, in experiments no improvement is found yet for OPD using parallelization. However, a potential benefit from creating a parallel version of OPD is not ruled out and it is expected that more research and more efficient implementations could still lead to an increase in the computational speed.

The second approach is to apply sequences of actions instead of single actions, which increases the time available for the planning process. Re-planning starts immediately after a sequence is returned, using as initial state a prediction of the state at the end of the previous sequence. The resulting algorithm is called Real-Time Optimistic Planning with Action Sequences (RTOPS). Extensive analysis is performed to find restrictions on the parameters of the algorithm that, when met, can guarantee real-time applicability. Additionally, the effect of using sequences of actions on the performance of the algorithm is investigated and bounds are put on the maximum performance loss.

The performance of RTOPS has been tested in various experiments on different problems: a
cart-pole simulation, an acrobot simulation and a real inverted pendulum. Different settings are compared and, overall, RTOPS proves to perform well, without violating real-time constraints. The experiments prove that RTOPS allows for the use of optimistic planning for real-time control of physical nonlinear systems.

Future work should focus on applying the ideas used to develop RTOPS to other optimistic planning algorithms, such as those that allow for continuous actions or stochastic systems. Furthermore, a parallelization of RTOPS could be developed that increases its computational speed.
# Table of Contents

Preface v

1 Introduction 1
  1-1 Optimal control .................................................. 2
  1-2 Online and offline planning ...................................... 2
  1-3 Optimistic Planning ............................................... 3
  1-4 Contributions ..................................................... 4

2 Background 7
  2-1 Optimal control problem .......................................... 7
  2-2 Optimistic planning for deterministic systems ................. 8
  2-3 A* search .......................................................... 11
  2-4 Summary and concluding remarks ................................ 12

3 Parallel N-Best-First Optimistic Planning 13
  3-1 Introduction ....................................................... 13
  3-2 Related work ...................................................... 14
  3-3 Parallel n-best-first optimistic planning ..................... 15
  3-4 Implementation details .......................................... 16
  3-5 Results ............................................................ 17
  3-6 Analysis of NBFOP performance .................................. 21
  3-7 Summary and concluding remarks ................................ 21

4 Real-Time Optimistic Planning with Action Sequences 25
  4-1 Introduction ....................................................... 25
  4-2 Related work ...................................................... 26
  4-3 Real-time guarantees .............................................. 28
    4-3-1 Fixed planning depth ...................................... 29
# Table of Contents

4-3-2 Fixed computational budget ................................................. 32
4-4 Performance analysis .......................................................... 36
  4-4-1 Perfect model ................................................................. 37
  4-4-2 Model mismatches ........................................................... 39
4-5 Summary and concluding remarks .......................................... 43

5 Implementation and Experiments for RTOPS .......................... 45
  5-1 Cart-pole simulation ............................................................ 45
    5-1-1 Experiment design ......................................................... 46
    5-1-2 Experimental results ..................................................... 51
  5-2 Acrobot simulation .............................................................. 59
    5-2-1 Experiment design ......................................................... 59
    5-2-2 Experimental results ..................................................... 61
  5-3 Real inverted pendulum ....................................................... 66
    5-3-1 Experiment design ......................................................... 66
    5-3-2 Experimental results ..................................................... 69
  5-4 Summary and concluding remarks ...................................... 74

6 Conclusion and Future Work ............................................... 75

Bibliography .............................................................................. 79

Glossary ..................................................................................... 83
  List of Acronyms ........................................................................ 83
  List of Symbols ......................................................................... 83
This document is the thesis written as part of the final MSc project for the degree of Electrical Engineering. The project is carried out for the largest part at Delft Center for Systems and Control (DCSC), located at the faculty of Mechanical, Maritime and Materials Engineering at TU Delft.

My MSc program is that of Electrical Engineering and in particular the track Signals and Systems. The choice to carry out my final project at DCSC is not, however, a strange or even uncommon choice. Before 2003, the Electrical Engineering department still had its own systems and control group, which merged with, among others, the systems and control group of Mechanical Engineering, to establish DCSC. The strong relation between Electrical Engineering and Systems and Control has never ceased to exist.

I would like to thank my MSc coordinator Richard Heusdens and professor Michel Verhaegen for making it possible for me to focus primarily on systems and control subjects, within the framework of the MSc Electrical Engineering.

I would also like to thank Jean-Francois Hren. Although we never met, it was his excellent work that provided the C implementations which I used to develop and test RTOPS.

Of course, sincere thanks are also due to professor Robert Babuška and dr. Lucian Buşoniu for their support and advice during the entire project. Special thanks to Lucian for inviting me for a visit to the Technical University of Cluj-Napoca, for making the visit go as smoothly as it could have possibly been and for making me feel very welcome in Romania.

Last but not least, I want to thank my family and friends for their support during this final year of my MSc.

The entirety of the project, including a literature review, has taken me almost a year and while – as I expect to be the case in any project – it has seen its ups and downs, I have very much enjoyed working on this and I feel its a great conclusion to the years I have spent at TU Delft.

Delft, University of Technology  M.H. Wensveen
February 24, 2014

Master of Science Thesis  M.H. Wensveen
Online Optimistic Planning (OP) is a method to solve optimal control problems. These optimal control problems, modeled as Markov Decision Processes (MDPs), are to be solved by maximizing the cumulative value of sequences of actions (the return). In a typical OP algorithm, immediately after applying the first action, a sequence is re-planned. OP finds action sequences by exploring a tree of possible sequences in an optimistic fashion, exploring first in the direction of sequences that have the largest upper bound on the return. Searching in an optimistic way increases the efficiency of tree search and hence decreases the search time.

While working algorithms for OP do exist, no real-time performance has been achieved yet. That is, to our knowledge there is no published record yet of an OP algorithm being able to compute a near-optimal action sequence for motion control of general nonlinear systems in real-time. We focus on motion control of nonlinear mechanical systems, because – as opposed to e.g. systems in process control – the sampling time of these systems is relatively fast, leaving less time available for planning.

It is expected that there are several methods that could increase the computational speed and therefore decrease the computational delay in OP. For example, parallelizing algorithms is known to increase computational speed. Also in the field of classical planning, parallelization has proven to increase the overall speed of the algorithms.

Another option could be to apply sequences of actions instead of single actions, to increase the time available for planning. The typical usage of OP is to apply the first action of a sequence, and then start planning again from the new state. This is similar to receding horizon control as seen in Model Predictive Control (MPC). Instead of re-planning after applying just one action, one could apply multiple actions from the planned sequence and start planning the next sequence right away. That way, the available planning time is multiplied by the number of actions of one sequence that is applied.
1-1 Optimal control

As said before, the control problems we consider can be modeled as MDPs. In an MDP the state $x$ of the system, which can be anywhere in the space $X$, can be influenced by an action $u \in U$. In this work, we consider only deterministic dynamics, meaning that the model always gives the same next state for a given state-action pair. We also consider only small, discrete action spaces $U$, while large, possibly infinite, state spaces are allowed. Each state transition receives a reward and the cumulative discounted reward (the return) over an infinite horizon is used as a measure to be maximized. Optimal control aims to find the actions that maximize the return. Note that for some problems the value function might be more intuitive when it is defined as a cost to be minimized. This is easily adapted to a return to be maximized, however, by simply defining the reward as a negative cost.

Consider as an example the problem of moving a car from point $A$ to point $B$. The states in such a problem could include the position and velocity of the car. The goal could be for example to minimize the time it takes to arrive at point $B$, or to arrive there with minimum fuel consumption. These goals can be formulated as a return to be maximized. In case the goal is minimum time, the reward should be equal to one when the car is at point $B$ and zero elsewhere. When fuel conservation is the goal, the reward should be proportional to the negative of fuel consumption (and therefore proportional to the negative of the gas pedal position). Optimal control aims to find the optimal actions, that maximize the chosen value function.

Other examples of optimal control problems include the moving of containers using a crane, with the goal to minimize the swinging motion, navigation of a mobile robot or the manipulation of a robot arm.

1-2 Online and offline planning

Many approaches to solve an optimal control problem as described in the previous section, like dynamic programming [1], aim at finding a global optimal policy $\pi(x)$ that describes the optimal action for each possible state in the state space $X$. For nonlinear systems and large, possibly infinite, state spaces such a task is far from easy. Another option, that avoids the need to find an optimal action for each possible state, is to search locally at state $x$ only for an optimal sequence of actions when starting from the current state $x$.

There are two ways to search for such a sequence of actions; one can either compute the entire sequence beforehand (offline) and then fully apply it, or one could compute a sequence, apply the first action (or several actions at the start of the sequence) and then start over from the next state. The latter approach is called online planning.

That is, online planning considers at each step the current state of the system and determines its response to sequences of actions by using the system’s model. Typically, the best found action is then applied to the system, and the planning starts over from the resulting state. Figure 1-1 shows the difference between online and offline planning graphically.

Planning under limited computational resources, and planning in real-time in general, usually calls for online planning algorithms, where planning and execution are interleaved, see again
1-3 Optimistic Planning

OP searches for near-optimal sequences of actions that maximize the return, by developing a tree representation of the state transitions in an optimistic way [2]. Starting from an initial state, a model of the system is used to simulate state transitions using each of the available actions. Then, the state is selected that has the highest upper bound on the future rewards, and the transitions from that state are simulated. The fact that OP uses upper bounds on future rewards to develop the tree, is what makes OP an optimistic approach to planning.

Several OP algorithms exist, but we will only consider the algorithm Optimistic Planning for Deterministic systems (OPD) [2]. This algorithm is discussed in more detail in Chapter 2. OPD works only for a finite number of discrete actions, but is relatively unaffected by the state dimensionality [2].

Typically, OPD is used with a fixed computational budget, meaning that the algorithm can perform a fixed number of node expansions before a sequence of actions is returned. OPD has the property that the performance of the algorithm grows with the number of expansions it performs, making it an anytime algorithm [2].

Another setting is possible where, instead of imposing a computational budget, OPD is used with a fixed target depth. Then, OPD plans until the tree is developed to the target depth, after which the best found action sequence is returned.
1-4 Contributions

The main contributions of this work are the introduction of the OPD variant Real-Time Optimistic Planning with Action Sequences (RTOPS), the characterization of the computational complexity of RTOPS and the use of this characterization to derive conditions that guarantee real-time applicability. RTOPS applies sequences of actions instead of single actions, in order to increase the time available for planning. Re-planning starts immediately after the previous planning instance is finished and the action sequence is used to predict the state at the end of the sequence. The prediction of the state is used as initial state when re-planning. Several settings are introduced for RTOPS regarding the length of the applied sequence, the computational budget and/or the target depth. It is the characterization of the computational complexity that ensures that the conditions derived in Chapter 4, that guarantee real-time applicability, are usable in practice, which is proven by several real-time experiments.

In addition, we introduce a parallelization of OPD, called N-Best-First Optimistic Planning (NBFOP). This algorithm selects not just one optimistic node, but \( N \) nodes with highest upper bound on future reward. These \( N \) nodes are distributed among \( N \) threads, which expand the nodes concurrently. The performance of NBFOP has been examined in several experiments.

Next, Chapter 2 discusses in more detail the optimal control problem and optimistic planning. Chapter 3 reviews parallelization in classical planning and examines the application of one parallelization approach to OPD. Chapter 4 introduces an adaptation of OPD that uses sequences of actions. Conditions are derived that guarantee real-time applicability of the algorithm. In Chapter 5 the introduced algorithm from Chapter 4 is tested on different systems. Finally, Chapter 6 concludes the thesis and proposes some ideas for future work.

Note that readers not interested in the parallel implementation can skip Chapter 3 and readers not interested in RTOPS can skip Chapters 4 and 5. These two alternative reading threads are shown graphically in Figure 1-2.
Figure 1-2: This diagram shows two alternative reading threads.
This chapter discusses in more detail the optimal control problem that Optimistic Planning for Deterministic systems (OPD) aims to solve. Also, OPD itself is reviewed in detail and its relation to the classical A* search algorithm is shown.

2-1 Optimal control problem

An optimal control problem is the problem of controlling a system’s states such that a predefined cost function is minimized, or, equivalently, such that a predefined return or value function is maximized.\(^1\)

We consider deterministic, discrete-time nonlinear systems, of which the dynamics are described by

$$x_{k+1} = f(x_k, u_k)$$ \hspace{1cm} (2-1)

with \(x_k \in X\) the state and \(u_k \in U\) the action, where \(X\) and \(U\) are the state space and action space respectively. To each state transition a reward is assigned, defined by

$$r_{k+1} = \rho(x_k, u_k),$$ \hspace{1cm} (2-2)

where \(\rho : X \times U \to \mathbb{R}\) is the reward function. In the sequel it is assumed that all rewards are in the interval \([0, 1]\). Note, however, that any bounded reward function can be scaled to the interval \([0, 1]\), without changing the optimal solution.

Define for any policy \(\pi : X \to U\), which defines for all states in \(X\) which action to take, the value function \(V^\pi\) as the sum of discounted rewards starting from state \(x\):

$$V^\pi(x) = \sum_{k \geq 0} \gamma^k \rho(x_k, \pi(x_k)),$$ \hspace{1cm} (2-3)

\(^1\)In settings where costs are used instead of rewards, each state-action pair is assigned a cost instead of a reward. The two settings, however, are equivalent and would result in identical actions determined by a controller. Cost can simply be seen as a negative reward: for each cost \(c_k\), using a reward \(r_k = -c_k\) would result in the “reward-setting” as used in this work.
where \( x_0 = x \).

Also define the \( Q \)-function as the value of starting in state \( x \), applying action \( u \) and continuing from there according to policy \( \pi \), over an infinite-time horizon:

\[
Q^\pi(x, u) = \rho(x, u) + \gamma V^\pi(f(x, u))
\tag{2-4}
\]

The discount factor \( \gamma \) is constrained to the interval \([0, 1)\). When the action \( u \) taken in state \( x \) is equal to the action according to the policy \( \pi \), i.e. \( u = \pi(x) \), the value function and the \( Q \)-function are equal:

\[
V^\pi(x) = Q^\pi(x, \pi(x)).
\]

The optimal policy is defined as the policy that maximizes the value function, \( \pi^*(x) = \arg\max_\pi V^\pi(x) \). The resulting optimal value function \( V^*(x) = \max_\pi V^\pi(x) \) can be expressed as the solution to the Bellman equation:

\[
V^*(x) = \max_{u \in U} \left[ \rho(x, u) + \gamma V^*(f(x, u)) \right]
\tag{2-5}
\]

Similarly, the optimal \( Q \)-function \( Q^*(x, u) = \max_\pi Q^\pi(x, u) \) is the solution to the following Bellman equation:

\[
Q^*(x, u) = \rho(x, u) + \gamma \max_{u' \in U} Q^*(f(x, u), u')
\tag{2-6}
\]

### 2-2 Optimistic planning for deterministic systems

Optimistic Planning (OP) has been analyzed in much detail by various authors [3, 2, 4, 5]. This section aims to clearly explain the ideas of OPD, without going into very detailed analyses, for which the reader is instead referred to the mentioned publications. We adopt for a large part the notations as used in [6], particularly because of their suitability for the analysis of the algorithm that is introduced in Chapter 4.

In OPD limited computational resources \( n \) are considered and the algorithm is expected to return the best possible actions, after \( n \) units of computation. The limited computational budget means that the number of calls to the state-transition function (and reward function) is limited. Preferably, the performance of the selected action should be non-decreasing with increasing computational budget \( n \). This is a property of anytime algorithms [2].

Denote by \( u_d \) an action sequence of length \( d \). OPD searches the space of possible infinite length action-sequences starting from state \( x_0 \) with the goal to maximize the following value function:

\[
v_{x_0}(u_\infty) = \sum_{k \geq 0} \gamma^k \rho(x_k, u_k),
\tag{2-7}
\]

where \( u_k \), for \( k = 0, 1, \ldots, \infty \), are the actions from the infinite length sequence \( u_\infty \). Then, the optimal value for any infinite sequence is \( v^*_0 = \sup_{u_\infty} v_{x_0}(u_\infty) = V^*(x_0) \). Note that OPD works only with discrete actions and that the action sequences consist only of actions from the action space \( U \).

To evaluate performance, the simple regret \( R(x_k, u_k) \) is used [4]:

\[
R(x_k, u_k) = \max_{u \in U} Q^*(x_k, u) - Q^*(x_k, u_k)
\tag{2-8}
\]
That is, the simple regret is the loss in performance by applying one possibly sub-optimal action $u_k$ and continuing optimally instead of applying the optimal action right away. Note that applying the optimal action in state $x_k$ gives a regret equal to zero. As shown by Hren and Munos [2] a regret of at most $\epsilon$ for every state $x$ limits the maximum difference between the optimal value of an infinite length sequence of actions and the obtained value to

$$V^*(x) - \sum_{k \geq 0} \gamma^k \rho(x_k, u_k) \leq \frac{\epsilon}{1 - \gamma}.$$  

To find the near-optimal actions, OPD develops a look-ahead tree, where nodes represent states and branches represent actions. Figure 2-1 shows a visualization of the look-ahead tree. Each node is reachable following a unique path in the tree, corresponding to a unique sequence of actions. Note, however, that states in the tree are not necessarily unique, as it is possible that identical states are reachable from the initial state via different paths.

Three quantities are defined for action sequences $u_d$ of length $d$ starting from state $x$: a lower bound $\nu_x(u_d)$, a true value $v_x(u_d)$ and an upper bound $b_x(u_d)$.

The lower bound on the value of $u_d$ with initial state $x_0$ is defined as follows:

$$\nu_{x_0}(u_d) = \sum_{k=0}^{d-1} \gamma^k \rho(x_k, u_k),$$  

(2-9)

In some publications nodes are said to represent sequences of actions instead of states. The principles remain the same, however, since each action leads to a certain state (in deterministic systems) and any unique path in the tree represents both a unique action sequence and the state sequence that follows from it.
where $u_k$ are again actions from the sequence $u_d$. This quantity is a lower bound on the value of any infinite length action sequence that starts with $u_d$, applied in state $x_0$, because all future rewards are non-negative.

The upper bound on any infinite length sequence starting with $u_d$ is defined as:

$$b_{x_0}(u_d) = \nu_{x_0}(u_d) + \frac{\gamma^d}{1-\gamma},$$

(2-10)

where $\gamma^d + \gamma^{d+1} + \cdots = \frac{\gamma^d}{1-\gamma}$ is the reward obtained if all future rewards after depth $d$ are equal to one, which is the highest possible reward considering $r \in [0, 1]$.

Finally, the true value of an infinite length sequence starting with $u_d$,

$$v_{x_0}(u_d) = \nu_{x_0}(u_d) + \gamma^d V^*(x_d),$$

(2-11)

is the sum of discounted rewards obtained by following actions $u_d$ from $x_0$ and continuing optimally from $x_d$ onwards.

Note that by definition $\nu_{x_0}(u_d) \leq v_{x_0}(u_d) \leq b_{x_0}(u_d)$. Also note that these quantities can also be seen as values of leaf nodes\(^3\) at depth $d$, reachable by the path defined by $u_d$ and with the root node $x_0$. The values of all nodes that have already been expanded are defined recursively, by back-propagating the values of the leaf nodes. Each expanded node is then assigned the maximum values among its child nodes, starting at the leaf nodes, going back to the root. Then, OPD can easily find the leaf node with highest upper (or lower) bound on the value by starting from the root and selecting always the child with highest $b$ (or $\nu$) value until a leaf node is reached.

OPD develops the tree optimistically, by always expanding the leaf node with the highest upper bound $b_x(u_d)$, motivating the algorithm’s name. Note that an expansion is a call to the model (dynamics and reward functions) for all $K$ actions available in the current state. That is, $K$ new nodes are generated, each corresponding to the state resulting from one of the available actions. After $n$ expansions, OPD returns the (first action of the) sequence that has the highest $\nu_{x_0}(u_d)$ among all explored action sequences. The length $d$ of the returned action sequence is guaranteed to be the highest reached depth in the tree. The proof for this is found in [2].

Hren and Munos [2, 3] prove that the simple regret for optimistic planning applied in state $x_0$ with computational budget $n$ is bounded by

$$R_n(x_0) \leq \frac{\gamma^{d_n}}{1-\gamma},$$

(2-12)

where $d_n$ is the depth of the deepest expanded node and the subscript in $R_n$ indicates the cumulative regret after $n$ expansions is considered.

The depth that is reached by OPD is dependent on the asymptotic branching factor $\kappa(x) \in [1, K]$ of the near-optimal subtree $T^*(x)$, defined as follows [6, 3]:

$$T^*(x) = \left\{ u_d \mid d \geq 0, \ v^*_x - v_x(u_d) \leq \frac{\gamma^d}{1-\gamma} \right\}$$

(2-13)

\(^3\)A leaf node is a node in the tree that has not yet been expanded.
That is, $\kappa(x)$ is defined as follows:

$$
\kappa(x) = \lim_{d \to \infty} \sup |T_d^*|^{1/d},
$$

(2-14)

where $|T_d^*|$ represents the number of nodes in $T^*$ at depth $d$. Branching factor $\kappa(x)$ is a measure of the complexity of OPD. If $\kappa(x) > 1$, to reach a depth $d$ in the tree OPD requires $n = O(\kappa(x)^d)$ expansions [6]. In the special case that $\kappa(x) = 1$, only $n = O(d)$ expansions are needed to reach depth $d$. Intuitively this can be explained as follows: if there is only one action at each depth that is near-optimal as $d \to \infty$ (i.e. $\kappa(x) = 1$), each expansion will increase the depth of the tree. If more near-optimal actions are available at each depth, however, the near-optimal subtree grows exponentially, meaning the depth of the tree only grows with the logarithm of the number of expansions.

Then, for $\kappa(x) > 1$, it is shown that the regret of optimistic planning applied in state $x_0$ is [2, 3]:

$$
R_n(x_0) = O(n^{-\frac{\log 1/\gamma}{\log \kappa(x_0)}})
$$

(2-15)

For $\kappa(x) = 1$ the regret decreases exponentially with budget $n$:

$$
R_n(x) = O(\gamma^c n),
$$

(2-16)

with $c$ a constant.

2-3 A* search

A search algorithm takes as input a search problem and returns as a solution a sequence of actions, which – when applied starting from the initial state (node) – leads to the goal state (node) [7]. The typical search problem consists of a model of the dynamics $x_{k+1} = f(x_k, u_k)$, a search space $X$ and an action space $U$. In addition, it contains a way to evaluate nodes, i.e. to assign either a cost or a reward to a state-action pair. Finally, an initial state and a definition of the goal state(s) completes the problem. Note the clear relation of this setting with the optimal control problem as described in Section 2-1. While search algorithms – also known as classical planning – usually work with costs instead of rewards, recall that the two are interchangeable. A cost can simply be a negative reward and vice-versa.

One of the most well-known search algorithms – which also shows a clear resemblance to OPD – is the A* algorithm. A* is a form of best-first search, introduced by Hart et al. in [8]. In best-first search [7] an evaluation function $F(i)$ assigns a value to a node $i$, and based on the values of different nodes a choice is made on which node to expand next. The term best-first refers to the way of choosing the node to expand, because the node that seems best judging from the evaluation function is selected. Often the evaluation function will be the estimated cost or distance to the goal and the node $i$ with the lowest value $F(i)$ will be expanded.

A* assigns two values to each node $i$: the cost $G(i)$ of reaching node $i$ from the root and a heuristic estimate $H(i)$ of the cost of the remaining path from node $i$ to the goal node. The sum of these two values $F(i) = G(i) + H(i)$ is used to guide the search to the most promising paths, in order to reduce the total number of expansions needed to find an optimal path to a
That is, $A^*$ selects for expansion always the node $i$ with the lowest value $F(i)$. It is shown that if the heuristic estimate $H(i)$ is at most equal to the true minimal cost from $i$ to the goal, then $A^*$ eventually returns the optimal solution.

It is easy to see how OPD and $A^*$ are related. In OPD, the upper bound on the future rewards from a node $i$ at depth $d$, $\gamma^d$, can be seen as an optimistic heuristic estimate $H_{OP}(i)$ of the rewards from node $i$ to the goal. The lower bound on the value of a node at depth $d$ is also clearly defined in $\nu_x(u_d)$, and is now denoted as $G(i)$. Then, OPD selects for expansion the node with the lowest value $F_{OP}(i) = -G_{OP}(i) - H_{OP}(i)$, which clearly shows the similarity to $A^*$.

The similarity between OPD and $A^*$ motivates the use of e.g. parallelization approaches as applied to $A^*$ for OPD, as done in Chapter 3.

### 2-4 Summary and concluding remarks

The optimal control problem, modeled as a Markov Decision Process, was reviewed here. Optimal control aims to find the actions that maximize a value function, called the return, which is defined as the discounted sum of rewards. Optimal control problems with finite, discrete action spaces, but large, possibly infinite state spaces, can be solved by Optimistic Planning for Deterministic systems.

OPD uses a model of the system to simulate state transitions to develop a tree representation of sequences of actions. The tree is developed in an optimistic fashion, by expanding nodes that have the possibility of being part of the best possible path (i.e. the best possible sequence of actions). The upper bound on the future rewards, clearly defined because all rewards are constrained to the interval $[0, 1]$, is used to find these possibly best nodes.

The performance of OPD depends on the structure of the problem. More specifically, the proportion of near-optimal actions influences the performance of OPD. The performance is measured by the cumulative simple regret after $n$ node expansions, where the simple regret is the difference in value between the found action and the optimal action. For problems where the branching factor of the near-optimal subtree is equal to one, this cumulative regret decreases exponentially with the computational budget.

Finally, the classical $A^*$ search algorithm was reviewed and its similarity to OPD was clarified. This similarity motivates the use of e.g. parallelization approaches that were proven to work for $A^*$, for parallelization of OPD, as discussed in the next chapter.
3-1 Introduction

Because of the recent tendency towards multi-core machines instead of increased computational power per core, parallelization is a popular way to increase the computational speed of algorithms. Since long computation is an obstacle for the real-time application of Optimistic Planning for Deterministic systems (OPD), we looked into the possibilities of parallelizing the OPD algorithm. That is, spreading the total work of the planning over multiple processor cores, or threads, might reduce the total time needed to find a solution. Such an increase in computational speed by parallelization is found for example in classical planning, as reviewed in Section 3-2.

We highlight one of the methods reviewed in Section 3-2, used to parallelize the A* algorithm, and investigate its applicability for OPD. The method we discuss is a N-Best-First Search (NBFS)\(^1\) approach for OPD, where the \(N\) most optimistic nodes are expanded concurrently. The reasons for selecting this method are that it has shown to perform well in certain cases in the context of classical planning [10] and the fact that it should be relatively easy to implement.

Section 3-3 discusses the resulting parallel OPD algorithm, that we call (parallel) N-Best-First Optimistic Planning (NBFOP). More details on the implementation of the algorithm are given in Section 3-4. The NBFOP algorithm is also studied in experiments, discussed in Section 3-5. The performance of NBFOP is analyzed in Section 3-6 and finally, Section 3-7 concludes this chapter.

\(^1\)Note that this approach is called K-Best-First Search in [10]. To avoid confusion with the number of actions \(K\), we refer to it as N-Best-First.
3-2 Related work

Since OPD is closely related to some classical planning algorithms (also known as search algorithms), and since parallelization has been examined extensively in the context of classical planning, we review some relevant work here. More specifically, we review parallelization methods as applied to the A* algorithm from Section 2-3.

The simplest way to parallelize A* would be to always assign the $N$ current best unexpanded nodes to the $N$ available processors, taking the nodes from a global, shared list [11]. To prevent two threads from accessing the same data at the same time, mutexes are used which effectively lowers the potential speed-up due to the synchronization overhead or lock contention that is introduced. Contention means that one thread tries to get a lock on data which is already locked by another thread, forcing the former thread to wait until the data becomes free again. This parallel variant of A* has been shown to perform worse than the serial version of A* [10].

Burns [10] tried to improve parallel A* by introducing one master thread that handles all synchronization between the $N$ worker threads, in order to avoid contention. One approach he investigated is to parallelize a variation of A* called NBFS that selects the $N$ best nodes to expand, instead of selecting just one [12]. An advantage of NBFS is that it does not rely too much on the accuracy of the heuristic. In the parallelization of NBFS the master thread takes the $N$ best nodes and distributes them to the $N$ worker threads, which in turn perform the node expansions. Burns et al. [10] show, however, that this method does not scale well for problems where node expansion is fast, because this could lead to a relatively large waiting time, as the master should wait until all workers have finished their node expansions. However, since we are interested in scenarios where node expansion means simulating nonlinear dynamics, which is relatively slow, the scaling might not be an issue and this method remains a candidate for parallelization of OPD.

Another parallelization of a different version of A* is proposed by Evett in [13]. The details of this specific version of A*, called Retracting A* (RA*), are irrelevant here and are therefore omitted. The parallelization, Parallel RA* (PRA*), uses local lists of nodes for each thread. Newly generated nodes are assigned to one of the threads by a hashing function. Each thread can then expand nodes from its own local list, selecting first the one with the lowest $F$-value. To reduce the performance decrease due to colliding messages (several threads trying to broadcast nodes to one thread simultaneously), Evett et al. [13] implemented a two-phase communication scheme. First, a thread broadcasts only its own address, which is a very small message compared to those of the node information, which is sent in the second phase. The receiving thread enqueues the addresses and after receiving all messages, the thread iterates through the queue and receives the node information. This way, only very small messages, which take less time to send, are subject to collisions.

Burns et al. [10], suggest a variant of PRA* that uses asynchronous communication, without the extra overhead of the message passing. Because of asynchronous communication, threads can continue working after sending node information, instead of waiting for an acknowledgment from the receiving thread. In this method – called Hash Distributed A* (HDA*) – each thread has one queue for incoming nodes and one queue for each thread for outgoing nodes. When a node is to be transferred to another thread, the sending thread attempts to obtain a lock on the receiving thread’s incoming queue. If the lock is successful, the node is transferred; if not, the node is put in the corresponding outgoing queue instead and the thread
continues working. At some later time, a thread attempts to clear all outgoing queues and receive from the incoming queue, immediately going back to work if locks cannot be obtained. This method is shown to perform better than its synchronous counterpart, PRA*.

An additional improvement to HDA* proposed by Burns [10] is to add state space abstraction, giving the algorithm AHDA*. In AHDA*, instead of assigning single nodes to threads, blocks of nodes are assigned. This improvement should decrease the communication time, since node expansions often stay in the same thread, removing the need for communicating the node’s information to another thread. AHDA* is shown by Burns [10] to outperform serial A*, even for problems with relatively fast node expansions.

Finally, Burns [10] proposes the algorithm Parallel Best-NBlock-First (PBNF), where a many-to-one mapping from the original state space to an abstract state space is made, as done by Zhou and Hansen in [14]. The mapping creates blocks of nodes, called nbblocks, which can be assigned to threads for expansion. PBNF uses a heap of free nbblocks ordered according to the lowest $F(i)$ among nodes $i$ in the nbblocks. Threads select the nbblock with the best $F$-value to expand nodes. The threads keep expanding nodes in their acquired nbblock, until the first nbblock in the heap contains a better $F$-value than the current nbblock. When the latter is the case, the first nbblock on the heap is selected and the previous nbblock is freed. An nbblock is only on the heap of free blocks if the nbblock itself and its duplicate-detection scope are unused. The duplicate-detection scope of an nbblock $B$ is defined as all nbblocks to which any child of any node in $B$ can belong. That means that all threads can freely expand the nodes in their acquired nbblocks, without generating nodes belonging to an nbblock that is being used by another thread. PBNF was shown by Burns [10] to give the best improvement in computational speed, both for relatively slow and relatively fast node expansions.

### 3-3 Parallel $n$-best-first optimistic planning

The parallel NBFS approach [10] is implemented here to OPD. Where in A* this approach selects the $N$ nodes with lowest $F$-value, the parallelization of OPD, that we call NBFOP, selects the $N$ nodes with highest upper bound on future reward. The $N$ selected nodes are then expanded concurrently.

At the start of the program, $N + 1$ threads are created: one “master” thread and $N$ “worker” threads. The master thread selects the $N$ nodes to be expanded according to the optimistic upper bound. Instead of selecting only the node with the highest upper bound on the future reward, the $N$ best nodes are selected for expansion. If less than $N$ leaf nodes are available, all available leaf nodes are selected.

After the node selection is completed, each worker thread obtains one of the selected nodes and the master thread signals the workers to start expanding the nodes. The nodes can then be expanded freely, without interfering with other threads. The master thread waits for all worker threads to complete their expansions, after which the master thread handles the updating of the values of all nodes in the tree, by back-propagating the values of the newly generated nodes up the tree, back to the root node.

This process repeats until a total of $n$ sequential node expansions have been performed, where $n$ is the computational budget expressed as a maximum number of node expansions. That is, if $N$ threads are all expanding nodes concurrently, this only counts as one node expansion,
while in fact \( n \times N \) nodes are expanded (and \( n \times N \times K \) nodes are generated, where \( K \) is the size of the action space \( U \)). High-level pseudo-code for NBFOP is shown in Algorithm 3.1.

With the same computational budget, NBFOP can expand \( N \) times as many nodes as the serial version of Optimistic Planning (OP). Equivalently, the computational budget could be significantly lower, while still allowing the same number of nodes to be expanded.

Algorithm 3.1 Pseudo-code for NBFOP

\begin{algorithm}
\begin{algorithmic}
\State \textbf{Input:} initial state \( x_0 \), budget \( n \), number of threads \( N \)
\State 1: create \( N \) worker threads and one master thread
\State 2: Master: \( i \leftarrow 0 \)
\State 3: \textbf{while} \( i < n \) \textbf{do}
\State 4: \hspace{1em} Master: select \( N \) optimistic leaf nodes
\State 5: \hspace{1em} concurrently, \( j = 1, \ldots, N \):
\State 6: \hspace{2em} Worker \( j \): expand selected node \( j \)
\State 7: \hspace{1em} Master: Back-propagate value of \( N \times K \) generated nodes
\State 8: \hspace{1em} Master: \( i \leftarrow i + 1 \)
\State 9: \textbf{end while}
\State \textbf{Output:} \( u_d \) with highest lower bound \( \nu_{x_0}(u_d) \)
\end{algorithmic}
\end{algorithm}

3-4 Implementation details

The implementation as described in the previous section is realized in the C programming language. To realize the parallelism, the widely used POSIX thread implementation for UNIX systems is utilized. The library that provides all relevant functions is referred to as the pthreads library. The \( N \) worker threads that are created are given a unique integer ID from the set \( \{0, \ldots, N - 1\} \).

The node selection is done by keeping a sorted array of pointers to all available leaf nodes. Pointers to the \( N \) best nodes are put in a separate array, to be accessed by the workers. Each thread accesses the array at the index corresponding to their unique thread ID, to obtain one of the pointers to the nodes. The worker threads then expand the obtained nodes, and the generated nodes are stored temporarily in a thread-specific array, in order to avoid data corruption caused by two threads writing to the same memory simultaneously. Each thread increments a counter when it’s done expanding the node, to keep track of the number of threads that have finished. To avoid two threads trying to increment the counter at the same time, it is protected by a mutex. The last thread to complete the node expansion signals the master thread that all workers are done.

The master thread then copies the pointers to the generated nodes from the thread-specific arrays and puts them in an array with all leaf nodes. In addition to adding the newly generated nodes, the nodes that were expanded are removed from the array, leaving an array with only leaf nodes. This array is sorted by optimistic value and used to again select the \( N \) best nodes.

Note that when less than \( N \) leaf nodes are available, one or more of the threads will not obtain a node for expansion and be in an idle state, not doing any work.
Figure 3-1: The task in this experiment is to move the ball to the desired position (black circle). The red circle depicts the initial position. The action set consists of a “push” to the right and a “push” to the left. Because there is no friction, the best achievable solution is an oscillation around the desired position. Node expansion (applying the dynamics function) is relatively cheap for this problem.

3-5 Results

Several experiments have been done with NBFOP. Results of these experiments are shown and analyzed in this section.

Simple moving ball problem

The first experiments were done on the simple problem of moving a ball to a desired position. The action-set in this problem consists of two actions, one being a “push” to the left, the other a “push” to the right. There is no friction, so the best obtainable result is a small oscillation around the desired position. The problem is visualized in Figure 3-1. The desired position is the centre of the line.

The system has a two-dimensional state space, where the states $x$ are defined as follows:

$$ x = \begin{bmatrix} x \\ \dot{x} \end{bmatrix}, \quad (3-1) $$

where $x$ is the position of the ball in meters and $\dot{x}$ is the ball’s velocity.

The dynamics of the system are described by the following simple equations:

$$ \dot{x} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} x + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u, \quad (3-2) $$

Table 3-1: Comparison of the computational times for various parts of the serial and parallel OP algorithm. Data obtained from the simple moving ball problem.

<table>
<thead>
<tr>
<th></th>
<th>Serial</th>
<th>Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computational budget</td>
<td>55</td>
<td>55</td>
</tr>
<tr>
<td>Total time [nsec]</td>
<td>12.1</td>
<td>849.3</td>
</tr>
<tr>
<td>Selection time [nsec]</td>
<td>0.52</td>
<td>3.89</td>
</tr>
<tr>
<td>Expansion time [nsec]</td>
<td>2.6</td>
<td>51.5</td>
</tr>
</tbody>
</table>

Table 3-2: Comparison of the computational times with different computational budgets for the serial and parallel OP implementation. Data obtained from the simple moving ball problem.

<table>
<thead>
<tr>
<th></th>
<th>Serial</th>
<th>Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computational budget</td>
<td>55</td>
<td>20</td>
</tr>
<tr>
<td>Total time [nsec]</td>
<td>12.1</td>
<td>552.1</td>
</tr>
</tbody>
</table>

where $u$ is the control action.

In several experiments the computational time for different parts of the algorithm has been measured, both for the regular serial algorithm and for NBFO. The results can be seen in Table 3-1. From the measured times in the table, it is easily seen that the parallelization in this case only has a negative effect on the computational time. It is likely that for this problem, where node expansion is relatively fast, the overhead due to parallelization is much larger than the benefit from expanding several nodes concurrently.

The same results are found when setting a lower computational budget for the parallel algorithm, which still allows for at least the same number of nodes being expanded. The computational times for this case are show in Table 3-2. Note that the budgets were chosen such that the performance in terms of obtained reward were equal for the serial and parallel algorithms.

Pendulum swing-up with obstacles

Further experiments were done on a pendulum swing-up problem, depicted in Figure 3-2. For the model equations of the system, see Section 5-1-1. The pendulum is to be swung up by moving the cart, while avoiding the obstacles. Note that checking for obstacle collision is part of the node expansion.

Unfortunately, initialexperimentsusingNBFO for this problem also did not show the desired speed-up over the serial algorithm. In fact, with the parallel algorithm more time was needed to find a near-optimal action sequence.

To be able to compare the computational time of several aspects of the two algorithms, the computational budgets $n$ were tuned for the serial and parallel implementations such that the two algorithms showed the same performance. That is, the computational budgets were found such that the reward for each time step was equal for the serial and the parallel algorithm. This is shown in figure 3-3, where the computational budgets for the serial and parallel implementations were found as $n_{ser} = 227$ and $n_{par} = 187$, respectively. To clarify,
Figure 3-2: The red pendulum and cart depict the initial position, and the black pendulum on the right shows the desired angle for the rod. Due to the obstacles, the desired position to be able to achieve the desired angle is constrained to the location where the black pendulum is depicted. The action set consists of two actions: moving the “cart” left, and moving it right.
For the serial and parallel algorithms the computational budgets have been chosen as low as possible, such that the problem is solved and the rewards for the two algorithms are equal. These budgets are found to be $n = 227$ for the serial algorithm and $n = 187$ for the parallel variant.

A budget $n_{par}$ for the parallel algorithm means that in fact, $n_{par} \times N$ node expansions are permitted, as long as $N$ expansions are always done concurrently.

With the two algorithms using as few node expansions as possible to still solve the task and obtain equal rewards, a good comparison can be made between the two implementations in terms of computational times. Some time measurements are shown in Table 3-3, including the total time for one planning iteration, the time needed to select nodes and the time needed to expand nodes.

The results show that while the relative time spent on expanding nodes is decreased, the absolute time for node expansions is significantly higher using NBFOP compared to the serial algorithm. The time needed to select the nodes to expand is also significantly higher in the parallel version. The latter is not entirely unexpected, because the way nodes are selected is likely to be inferior to the way nodes are selected in the serial algorithm.

The effects of increasing the computational complexity for node expansions has also been examined. The computational time for one expansion was increased, by using more samples per time step for the numerical integration. Figures 3-4 and 3-5 below show the comparison of the total time spent on expansions for different numbers of samples used. In these experiments, again a maximum number of 227 node expansions was used for the serial algorithm, and 187 for the parallel algorithm. Figure 3-6 shows the total time spent on expansions relative to the total time spent to complete one planning iteration.

These figures show that for very slow node expansions there could indeed be a benefit from
expanding nodes in parallel. The computational time per expansion is relatively low for NBFOP when the computational complexity of node expansions is high. The percentage of time spent on expanding nodes is also high for NBFOP with slow expansions, while a relatively large amount of the total computational time is spent on other actions with slower node expansions.

This indicates that NBFOP could possibly expand the same number of nodes as the serial version, while less time is needed to do this. This means that it NBFOP might indeed decrease the computational time for a given computational budget, if node expansions are relatively expensive.

3-6 Analysis of NBFOP performance

As explained in for example [2], OPD shows the biggest improvement over uniform planning for a specific class of problems, where there is only a small fraction of near-optimal nodes. The best performance guarantees of OPD arise when the branching factor $\kappa$ of the near-optimal subtree is $\kappa = 1$. This case means that (possibly after a non-asymptotic regime at small depths) at each depth there is only one interesting (optimistic) node. In this case, selecting for expansion $N > 1$ nodes, as is done in NBFOP, results in $N - 1$ uninteresting nodes being expanded. Then, the extra expansions done by NBFOP are useless and only introduce extra overhead compared to serial OPD.

In case $\kappa > 1$, possibly after a small number of expansions, all nodes that are expanded by NBFOP belong to the near-optimal subtree and thus need to be expanded to find an optimal solution. This leads to the expectation that, for problems where $\kappa \neq 1$, NBFOP should in theory be able to introduce a speedup over serial OPD. With a more efficient implementation it might be possible to reduce the overhead and to realize the speedup in practice.

3-7 Summary and concluding remarks

A parallel N-Best-First approach as found in [10] is applied to OPD. The resulting NBFOP algorithm is tested in experiments on simulated systems. While no improvement over serial OPD has been obtained yet, it is shown that the computational time per node expansion...
Figure 3-4: Time needed per expansion, compared to the computational complexity of simulating a state transition. For the parallel implementation, a distinction is made between the time needed for expansions in one “round” of expansion, and the average time per expansion. That is, with \( N \) threads, in one “round”, \( N \) expansions are done. The average time per expansion is then the total time for one round divided by the number of threads \( N \).

Figure 3-5: Comparison of total time spent on expansions during one planning iteration. Again, a distinction is made for the parallel algorithm between actual time spent and time per thread.
Figure 3-6: Time spent on expansions relative to the total time of one planning iteration.
relatively decreases by using this parallelization if node expansions are computationally ex-
spensive. The total computational time for an iteration of the planning algorithm, however,
is still larger for NBOP than for OPD. This is likely due to the overhead introduced by
parallelization and the sub-optimal implementation for selecting multiple optimistic nodes.

It is expected, however, that with more efficient implementations of this algorithm, or with
other parallelization approaches, a decrease in total computational time could be achieved. A
more efficient implementation would preferably decrease the overhead of the parallelization
and select nodes more efficiently. In the serial OPD implementation, nodes are selected by
recursively following the best path in the tree using pointers, while in the parallel NBOP
implementation, an array is kept and frequently sorted to find multiple best leaf nodes. An
implementation of the node selection in NBOP more along the lines of the node selection in
the serial algorithm, would likely decrease the time needed to select nodes.
4-1 Introduction

In order to make Optimistic Planning for Deterministic systems (OPD) applicable in real-time, the use of sequences of actions instead of single actions is investigated.

The approach is to use OPD to obtain a sequence of actions, which is no different from the typical OPD setting. However, instead of just applying the first action of the sequence, the entire sequence or a sub-sequence thereof will be used. At the same time, the next action sequence will be computed based on a prediction of the state at the end of the current sequence. The resulting algorithm is referred to as Real-Time Optimistic Planning with Action Sequences (RTOPS).

When an action sequence is sent, assume a “smart” actuator handles applying one action at each sampling time, each time taking the next action from the current sequence. The “smart” actuator can be seen as a combination of a buffer and an actuator, where the buffer ensures the appropriate action from the sequence is given to the actuator at the right time instances. The actuator then sends the received action to the system. See Figure 4-1 for a graphical representation of this scheme.

While the buffer handles the action selection from the available sequence, the state after applying that sequence is predicted and used as initial state for the computation of the next action sequence. The time available for the controller to search for a near-optimal sequence of actions is increased by a factor \( d' \), where \( d' \) is the length of the (sub-)sequence to be applied.

Since OPD returns sequences of actions by default, the complexity of the planning is not increased with this approach, effectively increasing the ability to meet real-time constraints.

Note that the following notations are used throughout this chapter: \( u_{dm}^{m} \) is an action-sequence of length \( d_m \), computed at calculation time \( m \), while \( u_k \) is a single action, used at system time \( k \). The superscript \( m \) in \( u_{dm}^{m} \) is omitted when the calculation time is not relevant. The
system’s state at time instance $k$ is denoted by $x_k$. Prediction of variables are indicated with a hat, e.g. $\hat{x}_{k+d'}$ is a prediction of state $x$ at time instance $k+d'$.

### 4-2 Related work

In [6] the use of (sub-)sequences is discussed for application in Networked Control Systems. While a limited computational time is not the issue in that paper – instead the necessity for efficient use of the network is the issue – the approach of applying sequences of actions is a solution in both settings.

An important difference to notice here is that [6] aims to minimize the number of instances at which data is sent through the network, by applying a sequence of actions before re-planning to obtain a new sequence. On the contrary, for real-time control, the re-planning needs to start while the previous action sequence is still being applied to the system. The research in [6], however, does offer relevant results on the use of action sequences. And in fact, in case a perfect model is available, the performance when re-planning right away will be identical as when re-planning is started after the sequence of actions is applied to the system. This is because re-planning from state $x_k$ while that state is not yet reached, only introduces problems if state prediction $\hat{x}_k$ is not guaranteed to be equal to $x_k$.

Busoniu et al. [6] introduce two different ways to use sequences of actions. The first method imposes a desired planning depth $d$ and Optimistic Planning (OP) then returns either the entire sequence up to depth $d$ or a sub-sequence thereof, of length $d' < d$. Because the length of the action sequence is fixed for each received sequence, the communication between system and controller occurs at a fixed interval, namely after every $d'$ applied actions. This method is called Clock-triggered Optimistic Planning.

The second method does not involve fixed-length action sequences. The planning algorithm in this case uses a fixed computational budget $n$ and plans until the budget is depleted. Depending on the specific complexity around the states where the system operates, the reached depth $d$ can vary among different planning instances. Instead of defining a desired sub-sequence length, a fraction $\alpha \in (0,1]$ could be defined, which results in a received action sequence of length$^1$ $\lceil \alpha d \rceil$. Here the planner receives a new state measurement exactly after

---

$^1\lceil \bullet \rceil$ is the ceiling operator.
[\alpha d] actions are applied, where \( d \) can be different for each sequence. This second method is called Self-Triggered Optimistic Planning (STOP). STOP is similar to the original running mode of OPD, where the computational budget is limited.

The use of sequences of actions instead of single actions to increase the time available for computation is not a novel idea. In 1999, Ronco et al. [15] introduced the idea of intermittently moving the horizon in a predictive control setting, applying a sequence of actions and then shifting the horizon by the length of the applied sequence. This idea arose because of time-limitations similar to those tackled in this work. Ronco [15] shows great improvement in performance using the intermittent version compared to the original version of the Model Predictive Control (MPC) variant he used. The computational delay in the original version is solved by simply using zero-order hold to repeat the last actions until a new one is computed.

The improved performance is as expected, as the available sequence is optimized up to a certain time in the future. While the optimal sequence can change due to new information obtained by the receded horizon on the next optimization instant, it is very likely that the previous sequence will be closer to optimal than a sequence consisting of simply one repeated action. In addition, the intermittent algorithm is shown to be more robust to both modeling errors and measurement noise, for increasing optimization times.

In [16] an intermittent approach to predictive control is described again, where a predicted state – based on the current state and the current action sequence – is used to find the next action sequence. This approach is similar to the one that is used here, as described in the previous section. Gawthrop looks at the intermittent approach in more detail in [17]. In that work he describes the design of a state predictor, an open-loop control policy and an intermittent controller.

Various open-loop controllers are considered in Gawthrop [17], which will be left out of consideration here. We only note that the open-loop controller returns control actions over a certain horizon. The intermittent controller converts the open-loop policies into a closed-loop policy, by intermittently shifting the horizon for the open-loop optimizer. It is assumed that the computation of the open-loop control requires less time than the time available between two instances of closing the loop. The intermittent control predicts the state at the next instance the loop will be closed, to be used as initial state by the open-loop controller. Experiments show that the method performs well if the time instances at which the loop is closed are not too far apart. It is also shown that the intermittent control is more sensitive to disturbances.

Chen et al. [18] state that the computational complexity of nonlinear optimization leads to two issues: 1) it is not guaranteed that a new optimal solution is ready in each cycle and 2) a (large) delay is introduced, which cannot be ignored. Chen [18] notes that, for a computational delay of \( \delta \), there is no use in calculating the optimal control from the current time step \( k \), since the system time will have advanced to \( k + \delta \) by the time the control is available. Therefore, the controls actually used are those optimized for the time interval \([k + \delta, k + N]\), where \( N \) is the time horizon. That way, by the time the control will be applied, it will be the one corresponding to state estimate \( \hat{x}(k + \delta) \), which will be closer to the true state at that time than \( x(k) \). An additional advantage of optimizing controls only for time interval \([k + \delta, k + N]\), instead of \([k, k + N]\), is that the optimization problem is smaller and hence can be solved faster.

Of course, to have the optimizer calculate only those controls starting \( \delta \) steps in the future, \( \delta \) has to be known beforehand. Especially in MPC the optimization time at different iterations
can vary and it is not straightforward to determine $\delta$ beforehand.

Applying a control sequence in open-loop until the next recalculation time – or the time that the next control sequence is ready – is called sampled-data MPC by Findeisen and Allgöwer in [19]. As a way to consider the necessary calculation time, they propose something similar to [18]. Denoting $t_i$ the $i^{th}$ recalculation instant and assuming the maximum calculation time $\delta^c$ is known, the controls to be applied after recalculation instant $t_i$ are those belonging to the time interval $\tau \in [t_i + \delta^c, t_{i+1} + \delta^c]$. This is because, as $\delta^c$ is the maximum calculation time, it is guaranteed that the controls are ready at time $\tau = t_i + \delta^c$, and that the next, possibly better, controls will be available $\delta^c$ after the next recalculation instant, $t_{i+1}$. A prediction of $x(t_i + \delta^c)$ is needed for the optimizer to start the optimization from that time instant.

Both Chen and Findeiser prove that their approach still guarantees stability. However, this will not be discussed here, as it is rather specific to MPC. We do stress again that the methods discussed here proved to work quite well. This is made clear by an example given by Findeiser, where the MPC controller accounting for delay has a performance very similar to the theoretical ideal MPC controller, whereas the implemented MPC controller that neglects the delay, performs significantly worse.

While most of the research reviewed above assumes e.g. a certain maximum computation time taken by the algorithm, we will focus on deriving explicit guarantees on the computational time in relation to the length of the applied sequence. Conditions will be derived that, in certain settings, guarantee real-time applicability, while using as much as the available time for computation as possible.

### 4-3 Real-time guarantees

This section analyses the use of sequences of actions in different settings, deriving for each setting a set of conditions that, if met, guarantee real-time applicability of RTOPS.

A distinction is made between three different settings regarding the planning depth $d$, the computational budget\(^2\) $n$ and the length $d'$ of the applied sub-sequence:

**Fixed planning depth and fixed length of applied sub-sequence:** OPD is applied until depth $d$ is reached in the tree. As soon as this depth is reached the algorithm returns the action sequence of length $d$. The first $d'$ actions out of the returned sequence are sent to the system, while OPD starts planning again, starting from a $d'$-step-ahead prediction of the state.

**Fixed computational budget and variable length of applied sub-sequence:** OPD is used in its “classic” form: exactly $n$ node expansions are performed, before the algorithm returns an action sequence of length $d$, which can vary in each planning instance. The sub-sequence sent to the system consists of the first $d' = \lceil \alpha d \rceil$ actions of the entire sequence. In this setting the length of the applied sub-sequence can vary after each planning phase.

---

\(^2\)Recall that the computational budget $n$ is defined as the number of node expansions. That is, with each computational unit, all $K$ children of a node are created, where $K$ is the number of available actions in a state.
4-3 Real-time guarantees

Fixed computational budget and fixed length of applied sub-sequence: OPD again performs \( n \) node expansions, returning a sequence of length \( d \). In this setting, however, the length \( d' \) of the sub-sequence that is sent to the system is also fixed. The conditions derived in this chapter guarantee that \( d \geq d' \).

Although OPD typically works with a fixed computational budget, in [6] it is shown that OPD is easily adapted to compute sequences with a fixed planning depth (i.e. computing sequences of a fixed length). Note that the first two settings were used in [6], while the last setting, with fixed budget and fixed sequence length, is a novel setting introduced here.

Note that in the following sections, it is assumed that computation is dominated by node expansions (i.e. state-transition simulations) and that the rest of the OPD algorithm requires negligible computation time. All conditions for real-time applicability derived in this chapter, however, can be adapted to include non-negligible calculation time other than the time needed for node expansions. When, in addition to node expansions, tree operations are also considered, the complexity of OPD depends on the branching factor \( k \) of the near-optimal sub-tree [20]. This can be taken into account in initial measurements of the computational time per node expansion. This is clarified later in Section 5-1-1.

4-3-1 Fixed planning depth

An algorithm proposed in [6] has been adapted to Algorithm 4.2, to be used for real-time control using (sub)sequences of actions. Recall that \( k \) is the system time and \( m \) is the planning instance. The notation \( \text{OPD}(x, d) \) is used to indicate a planning instance of OPD is started from state \( x \), with fixed target depth \( d \).

\textbf{Algorithm 4.2} Pseudo-code for OPD with a fixed target depth \( d \) and applying sub-sequences of fixed length \( d' \)

\begin{verbatim}
Input: initial state \( x_0 \), target depth \( d \), sub-sequence length \( d' \)
1: apply OPD(\( x_0 \), \( d \)), obtaining \( u^0_d \)
2: send \( u^0_d \) to buffer \( \triangleright \) buffer sends actions to system, while planning continues
3: \( k \leftarrow 0 \)
4: \( m \leftarrow 0 \)
5: loop
6: measure current state \( x_k \)
7: simulate \( x_{k+d'} = f(x_k, u^m_{d'}) \)
8: apply OPD(\( x_{k+d'}, d \)), obtaining \( u^m_{d+1} \)
9: (if needed, wait until buffer is empty)
10: send \( u^m_{d+1} \) to buffer
11: \( k \leftarrow k + d' \)
12: \( m \leftarrow m + 1 \)
13: end loop
\end{verbatim}

For OPD to be applicable in real-time in this setting, there is one crucial condition that has to be met. Simply put, this condition is that a sequence of length \( d \geq d' \) needs to be computable within a time that is at most equal to the time it takes the system to apply \( d' \) actions. That...
30 Real-Time Optimistic Planning with Action Sequences

is, the computation time needed to guarantee a depth \(d\) being reached, needs to be less than or equal to \(d'\) times the sampling period.

From [6, 2] the properties of OPD using a fixed planning depth are repeated\(^3\):

**Theorem 4.1.** OP has the following properties with a fixed target depth \(d\):

1. OP returns a sequence \(u_d\) of length \(d\), which is \(\frac{\kappa^d}{1+\gamma}\)-optimal.
2. If OP is called from \(x\):
   - For \(\kappa(x) > 1\): \(n = \mathcal{O}(\kappa(x)^d)\) expansions are required.
   - For \(\kappa(x) = 1\): \(n = \mathcal{O}(d)\) expansions are required.

For the proof of the theorem the reader is referred to [2].

From Theorem 4.1, we need that from state \(x\) at least \(n = \mathcal{O}(\kappa(x)^d)\) expansions can be done within \(d' \leq d\) sampling times, if \(\kappa(x) > 1\).

We introduce the worst-case relative branching factor \(\kappa^* = \sup_{x \in X} \kappa(x)\) as the highest relative branching factor among all states. With this, real-time applicability can only be guaranteed if the computation time required to perform \(cn^*d\) expansions is less than or equal to \(d'T_s\), where \(c\) is some constant and \(T_s\) is the sampling time. This leads to Condition 4.1:

**Condition 4.1.** A viable real-time algorithm is obtained if the following always holds:

\[
(T_e \cdot n \leq) T_e \cdot c \cdot \kappa^{*d} \leq T_s \cdot d'
\]  

(4-1)

Here \(T_e\) is the computational time needed for one node expansion.

For the specific, simple class of problems where \(\kappa(x) = \kappa^* = 1\), for all \(x \in X\) – i.e. the class of problems where asymptotically there is only one (near-)optimal action choice in each state – the necessary number of node expansions is reduced to \(n \leq cd\). This significant reduction in necessary node expansions – linear instead of exponentially growing with depth – relaxes Condition 4.1, leading to Condition 4.2:

**Condition 4.2.** For the class of problems where \(\kappa^* = 1\), a viable real-time algorithm is obtained if the following always holds:

\[
(T_e \cdot n \leq) T_e \cdot c \cdot d \leq T_s \cdot d'
\]  

(4-2)

Although problems do exist with small \(\kappa^*\) or even \(\kappa^* = 1\), the best case Condition 4.2 only guarantees a viable algorithm if \(\kappa(x) = 1\) for all \(x \in X\).

While most real problems are likely to have a relative branching factor of \(1 < \kappa^* < K\), in practice both the constant \(c\), and the branching factor \(\kappa^*\) are often not known a priori. Therefore, the worst-case scenario needs to be considered to put a true upper bound on the allowed computation time for a problem to still be feasible. The worst case is when \(\kappa^* = K\), that is, when the relative branching factor is equal to the number of available actions \(K\), and

\(^3\)The notation \(g = \mathcal{O}(h)\) means that \(g\) grows asymptotically at most as fast as \(h\). \(g = \Omega(h)\) means that \(g\) asymptotically grows at least as fast as \(h\).
all nodes at each depth need to be expanded. In this case, the number of nodes expanded to reach a certain depth is the same as in uniform planning, or breadth-first search, where nodes are selected according to minimum depth \([7]\). Then, to reach a depth \(d\) in the tree at least \(n = 1 + K + K^2 + \cdots + K^{d-1} = \sum_{i=0}^{d-1} K^i = \frac{K^d - 1}{K - 1}\) node expansions are needed. This leads to Condition 4.3 as the only condition to guarantee that the real-time constraints are met for problems of which the exact structure is unknown:

**Condition 4.3.** When \(\kappa^*\) and/or \(c\) is unknown a priori, a viable real-time algorithm is guaranteed if the following holds:

\[
T_e \cdot \frac{K^d - 1}{K - 1} \leq T_s \cdot d' \tag{4-4}
\]

Real problems will need to be tackled by meeting Condition 4.3, which is the only safe condition with unknown \(\kappa^*\) or \(c\). The spectrum of possible branching factors and the location of most real problems is shown graphically in Figure 4-2.

Note that parallelization of the algorithm would at best lower the computational time by a constant factor and hence relax Conditions 4.1, 4.2 and 4.3 up to a constant, as shown in Condition 4.4. Here the assumption is made that a perfect parallelization decreases the computational time by a factor equal to the number of threads used.

**Condition 4.4.** For a perfect parallelization of the algorithm to be viable in real-time, the following must hold:

- for \(\kappa^* > 1\) and \(c\) known a priori

\[
\frac{1}{N} \cdot T_e \cdot c \cdot \kappa^* d \leq T_s \cdot d' \tag{4-5}
\]

- for \(\kappa^* = 1\)

\[
\frac{1}{N} \cdot T_e \cdot c \cdot d \leq T_s \cdot d' \tag{4-6}
\]

- for \(\kappa^*\) or \(c\) not known a priori

\[
\frac{1}{N} \cdot T_e \cdot \frac{K^d - 1}{K - 1} \leq T_s \cdot d' \tag{4-7}
\]
where $N$ is the number of threads.

In real applications, the above conditions can be used as follows. Preliminary experiments should be done to determine the computational time $T_e$. One way to do this, is to run one iteration of OPD and measuring the time that it takes. Dividing the measured time by the total number of expansions done in the iteration (either keep track of this number in a fixed planning depth setting, or set a fixed number of expansions for this preliminary experiment) gives (an approximation of) the calculation time per expansion $T_e$. Section 5-1-1 explains in more detail how to get the best (i.e. safest) measurement for $T_e$.

Using $T_e$, together with the known number of actions $K$ and the system’s sampling time $T_s$, the equation in condition 4.3 can be used to find all feasible combinations of $d$ and $d'$. More specifically, one should choose any $d$ and $d' \leq d$, for which the following holds:

$$d \leq \frac{\log[(K - 1) T_s d' + 1]}{\log K},$$

which follows from the safest, worst-case condition, by rewriting Equation (4-4). Preferably, $d$ is chosen as large as possible, to improve the near-optimality $\gamma_d$.

Note that the situation is different for the very first sequence to be computed. It is assumed that initially enough (but finite) time is available to compute the first sequence, as no previous sequence is available to be used to ensure the real-time constraints are met. An alternative would be to manually define the first action sequence, and an appropriate choice can often be to use a sequence of actions zero.

Also note that the safe condition in this setting actually reserves enough time for uniform expansion, while the planning stops as soon as depth $d$ is reached. In practice this setting would therefore often be inactive, because the planning is finished before new actions are required. This might be desired in practice if power consumption is an issue, but other than that this setting is mainly used to support the following settings.

### 4-3-2 Fixed computational budget

When the computational budget $n$ is fixed, the conditions for RTOPS to be viable in real-time are similar to the conditions for the setting of fixed planning depth, the difference being that the depth $d$ should be expressed in terms of computational budget $n$.

For a fixed $n$, a distinction is made between two settings for the length of the sub-sequence that is applied to the system. First, a variable length sub-sequence, depending on the depth reached in the tree, can be used. Second, one could fix both the budget $n$ and the length of the sub-sequence that will be applied. Both of these cases are discussed in this section.

#### Variable length sub-sequence

The pseudo-code for RTOPS in the fixed-$n$ case with variable length of the applied sub-sequences is shown in Algorithm 4.3. The notation $\text{OPD}(x, n)$ is used to indicate a planning instance of OPD is started from state $x$, with fixed computational budget $n$. Note that the
sequence length $d$ and the length of the applied sub-sequence $d'$ can vary after each re-planning phase. The length of the applied sub-sequence is determined by a factor $\alpha \in (0, 1]$, such that $d' = \lceil \alpha d \rceil$.

\begin{algorithm}
\textbf{Algorithm 4.3} Pseudo-code for OPD with fixed computational budget $n$ and applying sub-sequences of variable length $d'$

\begin{algorithmic}
\State \textbf{Input:} initial state $x_0$, budget $n$, sub-sequence fraction $\alpha$
\State 1: apply OPD($x_0$, $n$), obtaining $u_d^0$
\State 2: $d' \leftarrow \lceil \alpha d \rceil$
\State 3: send $u_d^0$ to buffer \Comment{buffer sends actions to system, while planning continues}
\State 4: $k \leftarrow 0$
\State 5: $m \leftarrow 0$
\State 6: \textbf{loop}
\State 7: measure current state $x_k$
\State 8: simulate $\hat{x}_{k+d'} = f(x_k, u_d^m)$
\State 9: apply OPD($\hat{x}_{k+d'}, n$), obtaining $u_d^{m+1}$
\State 10: $d' \leftarrow \lceil \alpha d \rceil$
\State 11: (if needed, wait until buffer is empty)
\State 12: send $u_d^{m+1}$ to buffer
\State 13: $k \leftarrow k + d'$
\State 14: $m \leftarrow m + 1$
\State 15: \textbf{end loop}
\end{algorithmic}

From [6, 2] the properties of OPD in a setting with fixed computational budget are repeated in Theorem 4.2. For the proof of the theorem, see e.g. [2, 6].

\begin{theorem}
OP has the following properties with a fixed computational budget $n$:
\begin{enumerate}
\item OP returns a sequence $u_d$ of length $d$, which is $\frac{\gamma}{\frac{n}{\log \kappa^*}}$-optimal.
\item If OP is called from $x$:
\begin{itemize}
\item For $\kappa(x) > 1$: a depth of $d = \Omega\left(\frac{\log n}{\log \kappa^*}\right)$ will be reached.
\item For $\kappa(x) = 1$: a depth of $d = \Omega(n)$ will be reached.
\end{itemize}
\end{enumerate}
\end{theorem}

For proof of this theorem, the reader is referred to [2].

To start with, the most common case of $1 < \kappa^* \leq K$ is considered. From Theorem 4.2, we know that a depth $d \geq c \frac{\log n}{\log \kappa^*}$ is reached when OPD is used with a fixed budget $n$, where $c$ is some constant. Inversely, choosing computational budget $n \geq c' \kappa^* d$ is needed to reach depth $d$ and to guarantee $\frac{\gamma}{\frac{n}{\log \kappa^*}}$-optimality. Given the restriction that the planning time should be at most equal to the minimum execution time, determined by the length of the applied sub-sequence, Condition 4.5 follows directly:

\begin{condition}
In a setting with fixed computational budget $n$, for problems with $1 < \kappa^* \leq K$, a viable real-time algorithm is obtained if the following is met:
\begin{equation}
T_e \cdot n \leq T_s \cdot \alpha \cdot c \cdot \frac{\log n}{\log \kappa^*},
\end{equation}
where $c$ is some constant.
\end{condition}
The best case is also considered. That is, the case where $\kappa^* = 1$. From Theorem 4.2, we know that a depth $d \geq cn$ is reached with a budget $n$, where $c$ is some (unknown) constant. That is, $n \leq \frac{d}{c}$ expansions are needed to reach depth $d$, and to guarantee $\frac{d}{1-\gamma}$-optimality. The condition for the algorithm to be viable when $\kappa^* = 1$ is the following:

**Condition 4.6.** In a setting with fixed computational budget $n$, for problems with $\kappa^* = 1$, a viable real-time algorithm is obtained if the following is met:

$$T_e \cdot n \leq T_s \cdot \alpha \cdot c \cdot n$$

This simplifies to

$$T_e \leq T_s \cdot \alpha \cdot c$$

Finally, the worst-case condition is derived, because it is the only safe assumption in most real problems. When no prior knowledge about $\kappa^*$ or the constant $c$ is available, we know (see also Section 4-3-1) that $n = \frac{K^{\kappa^* - 1}}{K - 1}$ node expansions are needed to be guaranteed that a depth $d$ is reached in the tree, and thus to guarantee $\frac{d}{1-\gamma}$-optimality. Now, if this relation is solved for $d$, we find that with budget $n$, a depth $d = \frac{\log(n(K-1) + 1)}{\log K}$ is reached. The resulting condition is the following:

**Condition 4.7.** In a setting with fixed computational budget $n$, a viable real-time algorithm is obtained for all problems if the following is met:

$$T_e \cdot n \leq T_s \cdot \alpha \cdot \frac{\log[n(K-1) + 1]}{\log K}$$

As said before, most real problems will have an a priori unknown $1 \leq \kappa^* \leq K$. Therefore, the safe approach is to design the algorithm to meet Condition 4.7, which will guarantee a minimum depth $d$ being reached.

In real applications, the above conditions can be used as follows. The computational time $T_e$ should be found through preliminary experiments, in the same way as described at the end of Section 4-3-1. Using $T_e$, the number of actions $K$ and the system’s sampling time $T_s$, one can find the viable combinations of $n$ and $\alpha$ easily by rewriting Equation (4-12) to give the minimum required value of $\alpha$ for any given $n$:

$$\alpha_{\min} = \frac{nT_e \log(K)}{T_s \log[n(K-1) + 1]}$$

The maximum value of $\alpha$ is of course $\alpha_{\max} = 1$, for all $n$.

**Fixed length sub-sequence**

The pseudo-code for this setting is very similar to the pseudo-code for the variable-length sequences, but there are some crucial differences in the way the sub-sequence length is defined. The pseudo-code for the setting with fixed-length sequences is shown in Algorithm 4.4.

The properties of OPD as shown in Theorem 4.2 are also applicable in this setting. Again, three different conditions are derived: firstly, for problems where $1 < \kappa^* \leq K$ and the constant
Algorithm 4.4 Pseudo-code for OPD with fixed computational budget $n$ and applying subsequences of fixed length $d'$

**Input:** initial state $x_0$, budget $n$, sub-sequence length $d'$

1: apply OPD($x_0, n$), obtaining $u^0_d$
2: send $u^0_d$ to buffer \(\triangleright\) buffer sends actions to system, while planning continues
3: $k \leftarrow 0$
4: $m \leftarrow 0$
5: loop
6: measure current state $x_k$
7: simulate $\hat{x}_{k+d'} = f(x_k, u^m_d)$
8: apply OPD($\hat{x}_{k+d'}, n$), obtaining $u^{m+1}_d$
9: (if needed, wait until buffer is empty)
10: send $u^{m+1}_d$ to buffer
11: $k \leftarrow k + d'$
12: $m \leftarrow m + 1$
13: end loop

$c$ are both known a priori; secondly, for problems with the best-case $\kappa^* = 1$; and, finally, the safe condition which will mostly be used to tackle real problems.

All three of the above mentioned cases share a condition on the maximum allowed computational budget together with a minimum required length of the applied sub-sequence. This condition is that the computational time required to use the entire computational budget must not exceed the time that the system uses to apply $d'$ actions. This is summarized in Condition 4.8:

**Condition 4.8.** In a setting with fixed computational budget $n$ and fixed sub-sequence length $d'$, the following is a bound on the maximum computational budget and the minimum length of applied sequence, needed for real-time applicability:

$$T_e \cdot n \leq T_s \cdot d' \tag{4-14}$$

In addition to Condition 4.8, which ensures real-time applicability, another constraint on $d'$ is that it must be at most equal to the depth reached in the tree. Because the computational budget is fixed, the depth reached during planning could vary at each planning instance, depending on the structure of the problem. For the general case where $1 < \kappa^* \leq K$ and $c$ are known a priori, $d'$ must be chosen less than or equal to the minimal reached depth $d \geq c \cdot \frac{\log n}{\log \kappa^*}$:

**Condition 4.9.** In a setting with fixed computational budget $n$ and fixed length of applied sub-sequence $d'$, where $1 < \kappa^* \leq K$, to ensure enough actions are always available, the following must hold:

$$d' \leq c \cdot \frac{\log n}{\log \kappa^*} \tag{4-15}$$

In the best case – where $\kappa^* = 1$, the minimal reached depth is $d_{min} = cn$, giving the following constraint on the choice of $d'$:
Condition 4.10. In a setting with fixed computational budget \( n \) and fixed length of applied sub-sequence \( d' \), where \( \kappa^* = 1 \), to ensure enough actions are always available, the following must hold:

\[
d' \leq cn
\]  

Finally, the following upper bound on the length of the applied sub-sequence must be met for most real problems – for which \( \kappa^* \) and/or \( c \) are unknown – to guarantee having at least as many actions available as one wants to apply:

Condition 4.11. In a setting with fixed computational budget \( n \) and fixed length of applied sub-sequence \( d' \), where \( \kappa^* \) or \( c \) are unknown, to ensure enough actions are always available, the following must hold:

\[
d' \leq \frac{\log[n(K - 1) + 1]}{\log K}
\]  

Note that all above conditions are conservative, due to the fact that a worst-case branching factor \( \kappa^* \) is used and in practice possibly less expansions are needed to reach the same depth. Therefore, in practice, a depth \( d'' \geq d \) could be reached, if a fixed budget \( n \) is used, improving the performance of the sequences.

To use this setting in a real application, one can find the feasible values for \( d' \) for any budget \( n \) by looking at its lower bound of \( d'_\text{min} = \frac{T}{T_1} n \) – which follows directly from Equation (4-14) – and its upper bound as given in Equations (4-15) – (4-17).

Comparison Intuitively, one could say that the latter setting – where both the computational budget and the length of the applied sub-sequence are fixed – is potentially the most promising of the three. First of all, the advantage of fixing the computational budget over fixing the target planning depth, is that in the former setting all available computational time is used. In a fixed \( d \) setting it is very likely that on certain planning phases the target depth is reached well before the available computational time – imposed by the choice of \( d' \) – has passed. This means that (a significant) part of the available time is wasted.

In a fixed \( n \) and fixed \( d' \) setting, however, the algorithm guarantees using all available time, if one chooses the maximum allowed computational budget for a given \( d' \).

The guarantee of using all available time is not given in the fixed \( n \) case with variable sub-sequence lengths. In this setting, when a large depth is reached in the tree, a larger sub-sequence will be applied to the system. Consequently, more time is available before a new action sequence is required, but the computational budget remains the same. Therefore, in this setting too, time might be wasted.

4-4 Performance analysis

This section discusses the effects of using sequences of actions instead of single actions on the performance of OPD. A distinction is made in the analyses between the types of model of the system. Firstly, a perfect model will be assumed to be available in Section 4-4-1, before considering the effects of model mismatches in Section 4-4-2.
Figure 4-3: This shows a trajectory of a system, resulting from applying action sequences found by RTOPS with the use of a perfect model. The dots mark the states at the planning instances, and the paths in between are denoted by the action sequence that was used to obtain the path. With a perfect model, the predicted states are equal to the true states at any time $k$. The path resulting from applying the computed action sequences does therefore not diverge from the desired path.

4-4-1 Perfect model

Here, the assumption is made that the model of the dynamics $x_{i+1} = f(x_i, u_i)$ perfectly matches the real system’s dynamics. This assumption means that the predicted future state $\hat{x}_{k+d_m}$, based on an action sequence $u^m_{d_m}$ of length $d_m$, is equal to the real state $x_{k+d_m}$, resulting from applying the same action sequence starting from state $x_k$. This simplifies the problem, since any errors resulting from imperfect state predictions are avoided.

Figure 4-3 shows a path that a system could traverse using sequences of actions as found by RTOPS with the use of a perfect model. It should be noted that the state predictions done before each planning instance result in predicted states $\hat{x}_{k+d_m} = x_{k+d_m}$ equal to the actual states at the next planning instance.

The remainder of this section first gives a general guarantee on the performance of RTOPS with a perfect model, after which the influence of the length of the applied sequence will be discussed in more detail.

General performance guarantee

The authors of [6] show an important property of an OP algorithm that applies sequences of actions, namely that the near-optimality guarantee is not affected by the sub-sequence length. That is, the solution found by RTOPS is guaranteed to be $\gamma d_0$-optimal, where $d_0$ is the depth reached on the first planning instance. This is because of the perfect model, and the fact that upon re-planning a path can be found that obtains at most $\gamma d_0$ more reward after depth $d_0$. This general performance guarantee is formalized in Theorem 4.3.

Theorem 4.3. In case a perfect model is available, the near-optimality of RTOPS is:

$$v^* - v([u^0_{d_0}, u^1_{d_1}, \ldots]) \leq \frac{\gamma d_0}{1 - \gamma},$$

where $[u^0_{d_0}, u^1_{d_1}, \ldots]$ is the infinite sequence consisting of the sequences sent to the system after each planning instance and $d_0$ is the depth reached on the first planning instance.
Shorter versus longer sequences

This section gives more insight into the influence of the length of the applied sequence on the performance of RTOPS. Although the near-optimality guarantee is as good as when only one action is applied, this does not mean that the performance is the same regardless of the length of the applied sequence. While no definite answer can be given to the question whether longer or shorter applied sequences perform better, bounds can be given on the performance loss in both cases.

For the following we define \([u_{d_m}, u_{d_{m+1}}]\) as the concatenation of the two action sequences \(u_{d_m}\) and \(u_{d_{m+1}}\). As before, the length of the applied sequence is denoted by \(d'\), for which \(d' \leq d\). Additionally \(d'' > d'\) is used to denote for comparison a longer applied sub-sequence.

First, we give a bound on the performance loss when using shorter sequences compared to longer sequences [4]:

\[
v([u_{d''}, u_{d_{2}}]) - v([u_{d'}, u_{d_{1}}]) \leq \frac{\gamma^{d' + d_1}}{1 - \gamma}
\]  

(4-19)

This means that the maximum loss from applying a shorter sub-sequence is bounded by the maximum discounted rewards obtained after the combined length of the shorter sub-sequence and the sequence after re-planning.

As seen in [6], an example can be constructed that proves that the bound in Equation (4-19) is tight. Such an example is illustrated in Figure 4-4.

However, shorter applied sequences could also perform better than longer sub-sequences. A bound is therefore also needed on the maximum performance loss of using longer sequences instead of shorter ones. Intuitively, one would even expect shorter applied sequences to outperform longer sequences, as the earlier recalculation instances could provide new information, allowing for a better action sequence. The example in Figure 4-4, however, proves that this is not always the case.
We know that
\[ v^* - v(u_d) \leq \frac{\gamma^d}{1 - \gamma} \]  
(4-20)
and, because \( v(u_{d'}) \geq v(u_d) \), that also
\[ v^* - v(u_{d'}) \leq \frac{\gamma^d}{1 - \gamma} \]  
(4-21)

Then, the following bound on the loss from using longer sequences follows:
\[ (v^* - v(u_d)) - (v^* - v(u_{d'})) = v(u_{d'}) - v(u_d) \leq \frac{\gamma^d}{1 - \gamma} \]  
(4-22)

An example is constructed to prove that this bound is also tight, as shown in Figure 4-5.

Given that \( v([u_{d''}, u_{d_1}]) \geq v(u_d) \), we can now put a general bound on the performance as shown in Theorem 4.4.

**Theorem 4.4.** In case a perfect model is available, the following general bound can be put on the performance of a sequence of actions as compared to other sequences of different length:
\[ v(u_{d'}) - \frac{\gamma^d}{1 - \gamma} \leq v(u_d) \leq v([u_{d'}, u_{d_1}]) + \frac{\gamma^{d'+d_1}}{1 - \gamma} \]  
(4-23)

### 4-4-2 Model mismatches

This section analyses RTOPS under the assumption that the model \( \hat{x}_{k+1} = g(x_k, u_k) \) is not perfect and therefore errors in predicted states will occur. Figure 4-6 shows a visual representation of the situation that arises when the model used by RTOPS is not perfect.
Figure 4-6: This shows a path of a system, resulting from applying action sequences found by RTOPS when the used model is not perfect. The solid dots mark the states resulting from applying the sequences returned by RTOPS. The circles are predicted states, which differ from the true states. As initial states for planning, the predicted states are used.

Although it is assumed here that a perfect model is not available, some constraints are put on the model mismatch. Namely, it is assumed that there exists an $\varepsilon \geq 0$ such that:

$$
||g(x,u) - f(x,u)|| \leq \varepsilon, \forall (x,u) \tag{4-24}
$$

where $g$ is the model and $f$ is the real system. Additionally, assume that the model dynamics $g$ are Lipschitz continuous:

$$
||g(x,u) - g(x',u)|| \leq L||x - x'||, \tag{4-25}
$$

where $L$ is the Lipschitz constant.

Finally, it is assumed that the reward function is also Lipschitz continuous:

$$
||\rho(x,u) - \rho(x',u)|| \leq G||x - x'||, \tag{4-26}
$$

where $G$ is the Lipschitz constant.

The following two sections will derive bounds on the error in the value of an action sequences and on the closed-loop performance loss, respectively. All values depending on the system’s model $g$ will be denoted with a hat (\(^\hat{\cdot}\)).

**Bounds on value errors**

To analyze the effect of model mismatches on the value of an action sequence when applied to the real system compared to the same sequence being simulated using the available model, a bound on the error in the predicted state at given depth is needed. Such a bound is given below in Equation (4-27). Note that the notation $g(x_0, u_d)$ is used to indicate that actions $u_0 \ldots u_{d-1}$ from sequence $u_d$ are applied sequentially to states $x_0 \ldots x_{d-1}$ where $x_k = g(x_{k-1}, u_{k-1})$, using model $g$.

$$
e_d = ||\hat{x}_d - x_d|| = ||g(x_0, u_d) - f(x_0, u_d)|| \leq \sum_{i=0}^{d-1} L^i \varepsilon = \frac{L^d - 1}{L - 1} \tag{4-27}
$$
The proof of Equation (4-27) is given as an induction. For $d = 1$ the bound follows directly from Equation (4-24):

$$||\hat{x}_1 - x_1|| = ||g(x_0, u_0) - f(x_0, u_0)|| \leq \varepsilon = L^0 \varepsilon = \frac{L^1 - 1}{L - 1}$$

Assume the bound holds for $d$:

$$||\hat{x}_d - x_d|| \leq \sum_{i=0}^{d-1} L^i \varepsilon,$$

then, for $d + 1$:

$$||\hat{x}_{d+1} - x_{d+1}|| = ||g(\hat{x}_d, u_d) - f(x_d, u_d)||$$

$$= ||g(\hat{x}_d, u_d) - g(x_d, u_d) + g(x_d, u_d) - f(x_d, u_d)||$$

$$= ||g(\hat{x}_d, u_d) - g(x_d, u_d)|| + ||g(x_d, u_d) - f(x_d, u_d)||$$

$$\leq L||\hat{x}_d - x_d|| + \varepsilon = L \sum_{i=0}^{d-1} L^i \varepsilon + \varepsilon = \sum_{i=0}^{d-1} L^{i+1} \varepsilon + \varepsilon$$

$$= \sum_{i=0}^{d} L^i \varepsilon = \frac{L^{d+1} - 1}{L - 1}$$

which concludes the proof.

Note that Equation (4-27) holds for any $d$-step-ahead prediction, regardless of the initial step. That is, using $x_k$ instead of $x_0$ and $x_{k+d}/\hat{x}_{k+d}$ instead of $x_d/\hat{x}_d$ in Equation (4-27), does not change the bound on the error.

Now the effects of the model mismatch on the value obtained by a computed sequence should be analyzed. The following can be said about the lower bound on the value of a length $d$ action sequence:

$$\nu_{x_0}(u_d) = \sum_{k=0}^{d} \gamma^k r_{k+1}$$

$$= \sum_{k=0}^{d} \gamma^k \rho(x_k, u_k)$$

$$= \sum_{k=0}^{d} \gamma^k [\rho(x_k, u_k) - \rho(\hat{x}_k, u_k) + \rho(\hat{x}_k, u_k)]$$

$$= \sum_{k=0}^{d} \gamma^k [\rho(x_k, u_k) - \rho(\hat{x}_k, u_k)] + \sum_{k=0}^{d} \gamma^k \rho(\hat{x}_k, u_k)$$

(4-28)

$$= \hat{\nu}_{x_0}(u_d) + \sum_{k=0}^{d} \gamma^k [\rho(x_k, u_k) - \rho(\hat{x}_k, u_k)]$$

$$\leq \hat{\nu}_{x_0}(u_d) + \sum_{k=0}^{d} \gamma^k G||x_k - \hat{x}_k||$$

$$\leq \hat{\nu}_{x_0}(u_d) + \sum_{k=0}^{d} \gamma^k G \varepsilon L^k - \frac{1}{L - 1}$$
From Equation (4-28), a bound on the error in the lower bound on the value of a sequence starting with $u_d$ can be defined:

$$|\nu_x(u_d) - \hat{\nu}_x(u_d)| \leq \sum_{k=0}^{d} \gamma^k G \varepsilon \frac{L^k - 1}{L - 1} \overset{\text{def}}{=} \Delta(d)$$

(4-29)

The absolute value on the left-hand side can easily be shown to hold, because what is shown in Equation (4-28) can be shown in the same way for $\hat{\nu}_x(u_0)$. When $d$ goes to infinity, the latter term on the right-hand side of Equation (4-28) – $\sum_{k=0}^{d} \gamma^k G \varepsilon \frac{L^k - 1}{L - 1}$ – only converges if $L < \frac{1}{\gamma}$. If this condition is met, then from the above equation, we conclude that the difference between the value as obtained using the model and using the real system, with infinite length sequences, is bounded by:

$$|v_x(u_0) - \hat{v}_x(u_0)| \leq \lim_{d \to \infty} \sum_{k=0}^{d} \gamma^k G \varepsilon \frac{L^k - 1}{L - 1} = \frac{G \varepsilon \gamma}{(\gamma - 1)(\gamma L - 1)}$$

(4-30)

Note that by definition $v_x(u_\infty) = v_x(u_0)$ for infinite sequences. Similarly, we can define the bound on the error in the optimal value of the infinite-length solution:

$$|V^*(x) - \hat{V}^*(x)| \leq \frac{G \varepsilon \gamma}{(\gamma - 1)(\gamma L - 1)} \overset{\text{def}}{=} \Delta(\infty)$$

(4-31)

Closed-loop performance guarantees

Denoting by $[u_{d_0}^0, u_{d_1}^1, \ldots]$ the concatenation of sequences of length $[d_1, d_2, \ldots]$ applied to the system at re-planning instances $[0, 1, \ldots]$, we can now give Theorem 4.5 on the closed-loop performance of RTOPS, taking into account the model mismatch:

**Theorem 4.5.** The following is the bound on the closed-loop performance loss of RTOPS in the presence of model errors:

$$V^*(x_0) - \nu([u_{d_0}^0, u_{d_1}^1, \ldots]) \leq \frac{\gamma^d}{1 - \gamma} + \Delta(\infty) + \Delta(d)$$

(4-32)

**Proof.**

$$V^*(x_0) - \nu([u_{d_0}^0, u_{d_1}^1, \ldots]) \leq V^*(x_0) - \nu(u_{d_0}^0)$$

$$= \hat{V}^*(x_0) - \hat{V}^*(x_0) + V^*(x_0) - \nu(u_{d_0}^0)$$

$$\leq \hat{V}^*(x_0) + \Delta(\infty) - \nu(u_{d_0}^0)$$

$$= \hat{V}^*(x_0) + \Delta(\infty) - \hat{\nu}(u_{d_0}^0) + \hat{\nu}(u_d) - \nu(u_d)$$

$$\leq \hat{V}^*(x_0) + \Delta(\infty) - \hat{\nu}(u_{d_0}^0) + \Delta(d)$$

$$\leq \frac{\gamma^d}{1 - \gamma} + \Delta(\infty) + \Delta(d)$$

Equation 4-32 gives a conservative, but guaranteed bound on the maximum loss in value of applying the actions to the real system as compared to the value that would be obtained if a perfect model were available.
4-5 Summary and concluding remarks

An adaptation of OPD, called RTOPS, has been proposed that applies multiple actions instead of one action, making more time available for planning. When a sequence of length $d'$ is used, the time available for planning is increased by a factor $d'$, compared to typical OPD.

Three different settings regarding number of expansions, planning depth and sequence length were discussed. The first setting is to impose a fixed planning depth $d$ and a fixed length for the applied sub-sequence $d' \leq d$. The other two settings both impose a fixed computational budget $n$, as is typically done in OPD, but differ in the length of the applied sub-sequence. One setting uses a variables sub-sequence length proportional to the depth $d$ reached, and the other setting uses a fixed length $d'$.

For the three settings, conditions on $n$, $d$ and/or $d'$ have been derived that guarantee real-time applicability. This is done separately for the general case where $1 < \kappa \leq K$, the best case where $\kappa = 1$ and the worst-case where $\kappa = K$. Although most real problems will belong to the first mentioned general case, in practice $\kappa$ will rarely be known a priori. Therefore, the worst-case conditions are the only safe conditions in practice, as they guarantee real-time applicability regardless of the exact structure of the problem.

The effect of using sequences of actions on the performance of OPD has also been examined. Bounds have been derived on the maximum performance loss from using longer or shorter sequences when a perfect model is used, shown in Theorem 4.4. In addition, it is shown that with a perfect model, the near-optimality guarantee of OPD holds for RTOPS as well.

Since in practice model mismatches are always present, this has also been taken into account in the performance analysis. Under the assumptions that the error in the model for any state-action pair is bounded and that both the model and the reward function are Lipschitz continuous, a bound is derived on the loss of the value of the closed-loop control compared to the optimal value. Note that this bound (see Theorem 4.5) only holds if the Lipschitz constant for the model is less than the reciprocal of the discount factor.
This chapter discusses various experiments that have been conducted to analyze the practical performance of Real-Time Optimistic Planning with Action Sequences (RTOPS). Three different systems are considered: a cart-pole simulation, an acrobot simulation and a real inverted pendulum. For the experiments with each of these systems, the experiment design – including a description of the setup and selection of the parameters of RTOPS – along with the experimental results will be discussed. This is done in Sections 5-1, 5-2 and 5-3 for the cart-pole simulation, acrobot simulation and real inverted pendulum, respectively. In case of the experiments on the simulated systems, part of the experiments have been done with the addition of simulated model mismatches. In real nonlinear systems, it is of course inevitable that model errors are present.

The C code used for the implementation of RTOPS is based for a large part on the code written by Jean-Francois Hren [21]. His code is also used for the simulators.

5-1 Cart-pole simulation

The cart-pole system – or, the inverted pendulum on a cart – is a frequently used system to analyse the performance of control methods. To get an impression of how well RTOPS performs, a simulated version of the inverted pendulum on a cart is used.

Section 5-1-1 describes the experimental setup and how to find feasible settings for the target depth, computational budget and sub-sequence length for the cart-pole problem. Section 5-1-2 shows the results of some experiments conducted with these settings.
5-1-1 Experiment design

Experimental setup

The cart-pole system setup is shown in Figure 5-1. The system has a four-dimensional state space, where the states $x$ are defined as follows:

$$x = \begin{bmatrix} \theta \\ \dot{\theta} \\ x \\ \dot{x} \end{bmatrix},$$

(5-1)

where $\theta$ is the angle in radians of the rod with respect to the vertical axis, $x$ is the position of the cart in meters and $\dot{\theta}$ and $\dot{x}$ are the angular velocity and velocity of the cart, respectively.

The dynamics of the system are described by the following set of equations [22]:

$$\begin{bmatrix} \dot{\theta} \\ \dot{x} \end{bmatrix} = \begin{bmatrix} \frac{4l}{3} & -\cos(\theta) \\ lm_p \cos(\theta) & -\left(m_c + m_p\right) \end{bmatrix}^{-1} \begin{bmatrix} g \sin(\theta) - \frac{\mu_c \dot{\theta}}{m_p} \\ lm_p \dot{\theta}^2 \sin(\theta) - u + \mu_c \text{sign}(\dot{x}) \end{bmatrix}$$

(5-2)

Here, $l = 0.5$ m is the half-length of the pole, $m_p = 0.1$ kg and $m_c = 1$ kg are the mass of the pole and the cart, respectively, $g = 9.81$ m/s$^2$ is the gravity acceleration, $\mu_c = 0.0005$ is the friction coefficient of the cart on the track and $\mu_p = 0.000002$ is the rotational friction coefficient of the pole.

The goal in the experiments with this system is to achieve both swing-up of the pendulum and to position the cart to which the pendulum is attached in the middle of the line, where the red dot is shown in Figure 5-1. More precisely, the reward function to be maximized is

$$r_k = \frac{3}{8} \left( \cos(\theta_k) - \frac{2}{3} \frac{|x_k|}{L} + \frac{5}{3} \right),$$

(5-3)

where $L = 2.4$ m is the half-length of the track. The constants are chosen such that more weight is put on the swing-up than on the cart’s position and such that all rewards are in the interval $[0, 1]$.

The control action is a force applied on the cart and the action set used of size $K = 3$ is $u \in \{-5, 0, 5\}$. In addition, note that the sampling time in these experiments was set to $T_s = 0.1$ seconds.

Finding computational complexity

As mentioned before in Chapter 4, the first step to finding the parameters for RTOPS that allow for real-time application, is to find the computational time $T_e$ necessary for a node expansion. This can be done by conducting a simple preliminary experiment where a number of arbitrary node expansions is done. It is recommended to simply run one iteration of Optimistic Planning for Deterministic systems (OPD) with a fixed number of node expansions and to measure the total time this iteration takes. Dividing this total time by the set number of expansions gives a good estimate of $T_e$.
To be safe, one could add a certain safety margin of for example 10% to the measured $T_e$. Another safety precaution could be to run the preliminary experiment multiple times – perhaps for different values of the fixed number of expansions – and take the worst (i.e. the highest) $T_e$ found.

Although in theory all computation other than node expansions is assumed to be negligible, in practice, with the above method, all computation that is needed from the start of finding one sequence, until the algorithm can start finding the next, is taken into account and averaged over the number of node expansions. That way, the conditions remain usable, while possible non-negligible computation doesn’t have to be dealt with separately.

**Measuring $T_e$ to include tree operations** The time complexity of tree operations in OPD depends on the branching factor $\kappa$ of the near-optimal subtree [20]. This complexity is highest for $\kappa = 1$, when the algorithm has to navigate through the tree along a path of depth $O(n)$, giving overall time complexity of $O(n^2)$. For $\kappa = K$ the time complexity is only $O(n \log n)$.

To avoid computation taking longer in practice than expected, in the preliminary experiment to find $T_e$, the structure of the tree could be forced to be such that $\kappa = 1$. By simply using a reward function that always gives rewards equal to one for one action and rewards equal to zero for all other actions, the highest complexity of tree operations is guaranteed.

If this scenario is used to measure the total computational time, which is then divided by the total number of expansions, a worst-case $T_e$ is found. That way, the conditions from Chapter 4 remain valid, while taking into account computation other than node expansions.

Some additional tests have also been done to investigate for example the effect of the fixed planning depth or fixed computational budget on the value of $T_e$ that results from the method described above. OPD is run for several values of fixed depth $d$ and fixed computational budget $n$ and the time of the planning and simulation phase is measured. That is, the computation time for all tree navigation, node expansions, reward back-propagation and simulation of a sequence of length $d'$ is measured and divided by the total number of node expansions done – just as is the case in the simple method described above. The real reward function is used in this case.
Figure 5-2: Measured computation time $T_e$ for various target depths. The minimum $T_e$ that needs to be used to find feasible $d/d'$ combinations is chosen as the highest measured $T_e$ among the target depths for which OPD could control the pendulum to the upright position.

The results of the experiments to measure $T_e$ are shown for a range of fixed $d$ and fixed $n$ in Figures 5-2 and 5-3, respectively. The difference between the measured $T_e$ for small $d$ or $n$ and large $d$ or $n$ can be explained by the fact that in practice computation other than node expansions only becomes negligible when many node expansions are done. Because, however, the target depths and budgets that are of interest are high enough, all conditions in Chapter 4 that assume all computation to be due to node expansions, remain valid.

The fact that, at the higher end of the range, the measured expansion times $T_e$ are significantly lower for fixed $n$ than for fixed $d$ is most likely due to the varying number of expansions performed in the fixed $d$ case. More specifically, from certain states a depth $d$ could be reached with a relatively low number of node expansions. When this occurs, the influence of all computation beside the node expansions becomes much more apparent, just like it does for low fixed budgets $n$.

This analysis clearly shows that the measured $T_e$ can vary depending on the specific setting chosen. Because the number of expansions done at each depth in the tree also varies depending on the setting, the choice of a reward function that forces only one path in the tree to be expanded is motivated further. The best practice is to use such an alternative reward function and a computational budget in the same order that will be used for the true experiments, to find a good measure of $T_e$.

**Feasible settings for $d$, $n$ and $d'$**

The measured computation times $T_e$ can be used in the equations of Conditions 4.3, 4.7 and 4.11 to find all possible combinations of either $d$ and $d'$, $n$ and $\alpha$ or $n$ and $d'$ that allow for real-time application of RTOPS.
Figure 5-3: Measured computation time $T_e$ for various computational budgets. The minimum $T_e$ that needs to be used to find feasible $n/\alpha$ combinations is chosen as the highest measured $T_e$ among the budgets for which OPD could control the pendulum to the upright position.

Recall the upper bound on the target depth in the fixed $d$ setting from Equation (4-8). One can use any combination of $d$ and $d'$ for which $d \geq d'$ and

$$d \leq \frac{\log((K-1)\frac{T_s}{T_e}d' + 1)}{\log K},$$

and be guaranteed that the real-time constraints will not be broken. Of course, neither the length of the applied sub-sequence nor the target depth can be chosen as a fractional number. Therefore, $d$ must be no larger than the largest integer number that meets the upper bound in the above equation. A plot of all viable pairs $d$ and $d'$ for the cart-pole simulation is shown in Figure 5-4. The figure is in line with the intuition that to reach a larger depth $d$ with planning, more time is needed and hence the length of the applied sub-sequence should be larger.

For the fixed-$\alpha$ case with variable length of applied sub-sequence, the minimum value of $\alpha$ follows directly from Condition 4.7. Recall Equation (4-13), repeated here:

$$\alpha_{min} = \frac{nT_e \log(K)}{T_s \log[n(K-1) + 1]}$$

This minimum value of $\alpha$ and its maximum value $\alpha_{max} = 1$ are plotted to find the feasible combinations of $n$ and $\alpha$. This was done for the cart-pole simulation with $K = 3$, $T_s = 0.1$ and $T_e$ as taken from Figure 5-3. Figure 5-5 shows this plot.

Finally, for the fixed budget and fixed sub-sequence length setting, we find the feasible combinations of $n$ and $d'$, using Conditions 4.8 and 4.11. From these conditions it follows that for any given budget $n$, the length of the applied sequence must be chosen as:

$$\left\lfloor \frac{T_e}{T_s} n \right\rfloor \leq d' \leq \left\lfloor \frac{\log[n(K-1) + 1]}{\log K} \right\rfloor,$$

(5-4)
Figure 5-4: The gray area shows all possible combinations of $d$ and $d'$ for which the worst-case condition for real-time applicability of RTOPS holds (see Equation (4-4)). All intersections of the dotted lines represent valid choices for $d/d'$.

Figure 5-5: The gray area shows all possible combinations of $n$ and $\alpha$ for which the worst-case condition for real-time applicability of RTOPS holds (see Equation (4-12)).
Figure 5-6: The gray area shows all possible combinations of $n$ and $d'$ for which the worst-case condition for real-time applicability of RTOPS holds (see Equation (4-17)). Note that $d'$ can only be chosen on one of the dotted lines within the gray area. This is because fractional sequence lengths have no meaning in reality.

where $\lceil \bullet \rceil$ is the ceiling operator and $\lfloor \bullet \rfloor$ is the floor operator. A plot of all viable pairs $n$ and $d'$ is shown in Figure 5-6. In this setting, the best practice is to use the largest allowed budget $n$ for a given sub-sequence length $d'$. Choosing a lower $n$ than the maximum allowed means that time will be wasted that could be used for computation.

5-1-2 Experimental results

Fixed target depth $d$

Perfect model For a fixed-$d$ setting, the highest reward obtained in the experiments was with target depth $d = 8$, and its lowest allowed sub-sequence length $d' = 2$, as can be seen in Table 5-1. Graphs of the reward, position and angular position for this setting, are shown in Figure 5-7.

The other well-performing settings were mostly those with large target depth $d$ and/or short applied sub-sequences. In fact, the performance of all experiments where the applied sub-sequence length $d'$ was larger than 5, was very poor, and for target depth $d < 8$, a sub-sequence length of $d' > 2$ already led to bad performance.

Higher performance when a larger depth was reached is exactly as expected, because the optimality $\frac{-d}{1-\gamma}$ is directly related to the planning depth. Intuitively, one could also expect that applying longer sub-sequences of actions would not perform as well as shorter sub-sequences, because earlier re-planning (closing the loop earlier) means that new information
Table 5-1: Discounted sum of rewards (over 100 steps) obtained in the experiments with fixed $d$. Note that the fields marked with '-' indicate combinations of $d$ and $d'$ that do not meet the real-time applicability condition. The three best results are marked with $^1$, $^2$ and $^3$.

<table>
<thead>
<tr>
<th>$d'$ \ $d$</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>47.970</td>
<td>48.706 $^3$</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>34.715</td>
<td>48.166</td>
<td>48.709 $^1$</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>32.794</td>
<td>17.332</td>
<td>48.708 $^2$</td>
<td>48.367</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>35.058</td>
<td>17.332</td>
<td>44.507</td>
<td>48.558</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>28.796</td>
<td>29.044</td>
<td>38.991</td>
<td>48.558</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>2.891</td>
<td>37.079</td>
<td>34.792</td>
<td>35.207</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>24.199</td>
<td>41.263</td>
<td>39.754</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>27.171</td>
<td>43.434</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>23.336</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>31.134</td>
</tr>
</tbody>
</table>

Figure 5-7: Simulation with perfect model, for $d = 8$ and $d' = 2$. The number of actions was $K = 3$ and the discounted reward obtained over 100 steps is 48.709.

can be used earlier. Although this is proven to not always be the case, as seen in Section 4-4-1, it can still be the case in practice, as indicated by the experiments.

Model mismatch Simulations have been with the controller using different model parameters than the system. That is, some model mismatches are introduced. The goal of these experiments is to show that RTOPS still works even if a perfect model is not available. The best-performing setting for the case where a perfect model is available, is used here ($d = 8$ and $d' = 2$). Since this setting uses a relatively low length for the applied sub-sequence, it is expected to also perform best in the presence of parameter errors in the model.

For the pendulum swing-up problem, the parameters that are changed are the cart’s mass, the pole’s mass and the pole’s length. Several combinations of parameter errors are used.

Figures 5-8 and 5-9 show the results of two of these simulations. In both cases, with different parameter errors, the control task is still fulfilled.
Fixed computational budget \( n \), variable length applied sub-sequence

**Perfect model**  Also for a fixed computational budget, several experiments have been done on the pendulum, using various combinations of \( n \) and \( \alpha \), as found from the graph in Figure 5-5.

An interesting result here is one that confirms that sometimes longer applied sub-sequences can be better than shorter sub-sequences, while in other situations the relation is reversed.

For \( n = 10000 \), several experiments were conducted, among which \( \alpha = 0.25, \alpha = 0.75 \) and \( \alpha = 1.00 \) were used. The interesting result was that the worst performance was that of the middle setting for sub-sequence length, with \( \alpha = 0.75 \). While \( \alpha = 0.25 \) showed the best performance, \( \alpha = 1.00 \) also significantly outperformed \( \alpha = 0.75 \). The position, angular position and rewards are shown in Figures 5-10 to 5-12, while Figure 5-13 shows the lengths of the sub-sequences that were applied.

In Table 5-2 the discounted sum of rewards for the experiments conducted with fixed \( n \) and \( \alpha \) are shown. The combinations of \( n \) and \( \alpha \) that do not meet the real-time applicability conditions are marked in the table by ‘\( -\)’. The table shows that the best performance was achieved using \( n = 25000 \) with \( \alpha = 0.50 \).

From Table 5-2, one can see that most lower rewards occur for high values of \( \alpha \) – that is, for (relatively) long sub-sequence lengths – and/or for low values of \( n \) – that is, for lower reached depths in the tree search. Table 5-3 shows the depths that were reached during the planning in the experiments with fixed \( n \) and variable length sub-sequences. Both the depth reached on the first planning iteration and the average depth over the duration of the experiment are shown. Comparing Tables 5-2 and 5-3 shows a clear correlation between the performance and the planning depth. The settings that performed best are also the settings where the highest depth was reached. And, the lowest reached depths correspond to the lowest performance.
Figure 5-10: Results of experiments with \( n = 10000 \) and \( \alpha = 1.00 \). The number of actions was \( K = 3 \) and the discounted rewards obtained over 100 steps is 47.128.

Figure 5-11: Results of experiments with \( n = 10000 \) and \( \alpha = 0.75 \). The number of actions was \( K = 3 \) and the discounted rewards obtained over 100 steps is 44.953.

Figure 5-12: Results of experiments with \( n = 10000 \) and \( \alpha = 0.25 \). The number of actions was \( K = 3 \) and the discounted rewards obtained over 100 steps is 48.983.

Table 5-2: Discounted sum of rewards (over 100 steps) obtained in the experiments with fixed \( n \) and variable length of applied sequence. Note that the fields marked with ‘\(-\)’ indicate combinations of \( n \) and \( \alpha \) that do not meet the real-time condition. The three best results are marked with 1 and 2 (second place is shared).

<table>
<thead>
<tr>
<th>( \alpha ) ( \backslash ) ( n )</th>
<th>5000</th>
<th>10000</th>
<th>25000</th>
<th>54000</th>
<th>84000</th>
<th>115628</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.06</td>
<td>48.933</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>0.11</td>
<td>48.981</td>
<td>48.981</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>0.25</td>
<td>48.931</td>
<td>48.982</td>
<td>48.968</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>0.50</td>
<td>48.981</td>
<td>48.941</td>
<td>48.985</td>
<td>48.982</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>0.75</td>
<td>48.923</td>
<td>44.952</td>
<td>48.621</td>
<td>47.234</td>
<td>48.879</td>
<td>–</td>
</tr>
<tr>
<td>1.00</td>
<td>42.727</td>
<td>47.128</td>
<td>40.999</td>
<td>35.936</td>
<td>48.869</td>
<td>48.824</td>
</tr>
</tbody>
</table>
Figure 5-13: Applied sub-sequence length of various experiments with $n = 10000$.

Table 5-3: Planning depths in the experiments with fixed $n$ for the pendulum on a cart simulation. $d_0$ is the depth reached in the first planning iteration, $d_{avg}$ is the average depth reached over 100 steps.

<table>
<thead>
<tr>
<th>$n$</th>
<th>5000</th>
<th>10000</th>
<th>25000</th>
<th>54000</th>
<th>84000</th>
<th>115628</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td>$d_0$</td>
<td>$d_{avg}$</td>
<td>$d_0$</td>
<td>$d_{avg}$</td>
<td>$d_0$</td>
<td>$d_{avg}$</td>
</tr>
<tr>
<td>0.06</td>
<td>9</td>
<td>27.07</td>
<td>9</td>
<td>31.60</td>
<td>10</td>
<td>38.38</td>
</tr>
<tr>
<td>0.11</td>
<td>9</td>
<td>27.39</td>
<td>9</td>
<td>31.46</td>
<td>10</td>
<td>38.38</td>
</tr>
<tr>
<td>0.25</td>
<td>9</td>
<td>25.85</td>
<td>9</td>
<td>29.87</td>
<td>10</td>
<td>45.39</td>
</tr>
<tr>
<td>0.50</td>
<td>9</td>
<td>25.64</td>
<td>9</td>
<td>29.87</td>
<td>10</td>
<td>45.39</td>
</tr>
<tr>
<td>0.75</td>
<td>9</td>
<td>24.38</td>
<td>9</td>
<td>18.51</td>
<td>10</td>
<td>37.31</td>
</tr>
<tr>
<td>1.00</td>
<td>9</td>
<td>11.88</td>
<td>9</td>
<td>29.42</td>
<td>10</td>
<td>15.18</td>
</tr>
</tbody>
</table>
Table 5-4: Discounted sum of rewards (over 100 steps) obtained in the experiments with fixed \( n \) and fixed sub-sequence length \( d' \). Note that the table is sorted by highest rewards.

<table>
<thead>
<tr>
<th>( n )</th>
<th>( d' )</th>
<th>Discounted sum of rewards</th>
</tr>
</thead>
<tbody>
<tr>
<td>19900</td>
<td>2</td>
<td>48.9852</td>
</tr>
<tr>
<td>29850</td>
<td>3</td>
<td>48.9852</td>
</tr>
<tr>
<td>39800</td>
<td>4</td>
<td>48.9852</td>
</tr>
<tr>
<td>49750</td>
<td>5</td>
<td>48.9852</td>
</tr>
<tr>
<td>59700</td>
<td>6</td>
<td>48.9852</td>
</tr>
<tr>
<td>9950</td>
<td>1</td>
<td>48.9825</td>
</tr>
<tr>
<td>69650</td>
<td>7</td>
<td>48.9819</td>
</tr>
<tr>
<td>109500</td>
<td>11</td>
<td>48.8851</td>
</tr>
<tr>
<td>79600</td>
<td>8</td>
<td>48.8851</td>
</tr>
<tr>
<td>89570</td>
<td>9</td>
<td>48.8851</td>
</tr>
<tr>
<td>99520</td>
<td>10</td>
<td>48.8851</td>
</tr>
</tbody>
</table>

Table 5-3 also shows why this setting with fixed budget \( n \) performs better than the fixed \( d \) setting: the depths that are reached in the fixed \( n \) setting are much larger.

**Model mismatch** Experiments indicate that the setting of fixed budget \( n \), with variable length of applied sub-sequences, is not very suitable for use when a perfect model is not available. The fact that a larger reached depth in the tree automatically leads to a larger length of the applied sub-sequence, means that a larger planning depth can also lead to a larger effect of the model mismatch. This is due to the maximal error between true states and states according to the model, that can grow with the length of the applied sequence, as seen in Equation (4-27) in Section 4-4-2.

**Fixed computational budget \( n \), fixed length applied sub-sequence \( d' \)**

**Perfect model** Table 5-4 shows the results of all viable sub-sequence lengths \( d' \), in combination with their maximum budget \( n \). Interestingly, the best performance is shared by five different settings, namely those with \( 2 \leq d' \leq 6 \). Taking a closer look at those results shows that all states and thus rewards are identical for these five settings, while the depth reached in the tree (and of course the length of the applied sequence) differs. See also Figure 5-14, where the reached depths \( d \) are shown for two of the settings that led to the exact same action sequences. This result could very well indicate that this is in fact the optimal action sequence, as reaching a larger depth – that is, looking over a longer horizon – does not improve the solution. While the remaining settings in Table 5-4 to not share this possibly maximum achievable return, the difference is very small (in fact, it is negligible when comparing the state (and reward) trajectories) and all settings for the fixed \( n/d' \) case perform very well.

**Model mismatch** A good alternative to the setting with fixed \( n \) and variable applied sequence length turns out to be this setting, where both \( n \) and \( d' \) are fixed. While for the perfect model, there were various settings that gave the same performance, out of these only
the setting with the lowest $d'$ is selected for experiments with model mismatch, in order to minimize the error caused by the imperfect model.

Figures 5-15 to 5-17 show the results of RTOPS with fixed $n = 19900$ and fixed $d' = 2$ for various model parameter errors. The first of these figures shows that RTOPS performed very well when the cart’s mass was modeled too large, while the model parameters for the pole’s mass and length were both too low. Figure 5-16, however, shows that the performance wasn’t nearly as well when the pole’s mass and length were modeled too large. Looking at Figure 5-17 – which shows very good performance with a model mismatch where the pole’s mass is lower in the system than in the model, while its vice versa for the pole’s length – one can assume that the system is especially sensitive to the pole being shorter than in the model. This could be due to the fact that a longer pole could require more swings if the maximum force applied to the system is not very large. Then, the action sequence returned by RTOPS might be based on needing several swings, while in the real system the pole swings up more easily. The result would be that the behavior is not as desired.

**Comparison of the different settings**

From Tables 5-1 and 5-2 we see that the twelve best results with fixed $n$ and variable subsequence length, are all better than the best result with fixed $d$. When along with the computational budget also the length of the applied sequence is fixed, all tested settings outperform the fixed-$d$ case.

Intuitively, one would expect a fixed budget $n$ setting to outperform a fixed $d$ setting, because in the former case, more of the available time is used for computation, while in the latter case, computation may stop while more time is available. In case $n$ and applied sequence length $d'$ are both fixed, it is even the case that all available computational time can be used, if the budget is chosen as $n = \frac{T_s}{T_e} d'$.

We know that $\frac{K_d - 1}{N - 1}$ node expansions are needed to guarantee depth $d$ to be reached. We also know, however, that this is the worst-case bound, and depth $d$ may very well be reached
**Figure 5-15:** Simulation with model mismatch, with $n = 19900$ and $d' = 2$. Parameter errors: cart's mass $\times 1.16$, pole's mass $\times 0.4$, pole's length $\times 0.88$.

**Figure 5-16:** Simulation with model mismatch, with $n = 19900$ and $d' = 2$. Parameter errors: cart's mass $\times 1.05$, pole's mass $\times 1.2$, pole's length $\times 1.05$.

**Figure 5-17:** Simulation with model mismatch, with $n = 19900$ and $d' = 2$. Parameter errors: cart's mass $\times 1.08$, pole's mass $\times 0.41$, pole's length $\times 1.03$. 
with \( n < \frac{Kd-1}{K-1} \) expansions, or that a larger depth may be reached with the same number of expansions.

This fact can, however, be a disadvantage in practice, in the fixed-\( n \) with fixed \( \alpha \) case. This is because, when a larger depth is reached in the tree, the fixed \( \alpha \) will cause a sub-sequence of larger length \( d' = \lceil \alpha d \rceil \) to be applied to the system. In the presence of model errors, applying longer sequences will allow for the cumulative error to grow, which will in turn deteriorate performance.

This disadvantage is no longer an issue when both the budget \( n \) and the length of the applied sub-sequences are fixed. Intuitively, one would expect that this setting allows for the advantage of a larger depth to be reached to increase performance, while avoiding a model error to accumulate over more than \( d' \) time steps.

Experiments show, however, that reaching a larger depth, while keeping the same length of the applied sub-sequence, does not guarantee the performance to be better, when model mismatches are present. When comparing for example Figures 5-8 and 5-16, the setting of fixed \( d = 8 \) clearly performs better than fixed \( n = 19900 \), with both settings using a fixed \( d' = 2 \). Using a fixed \( n = 19900 \), however, guarantees a depth \( d \geq \lceil \frac{\log(n(K-1)+1)}{\log K} \rceil = 9 > 8 \) being reached. But despite the minimum depth being reached in the fixed-\( n \) case being larger than the constant depth being reached in the fixed-\( d \) case, the latter setting still outperforms the former, for this specific model mismatch. The cause for this could be that the effect of the model mismatch becomes too large as the depth increases during the planning. While the solution found with this larger depth might be better with respect to the planning-model, it could be that the action sequence became worse for the actual system.

5-2 Acrobot simulation

The acrobot is a two-link robot arm, where one joint is fixed and only the middle joint is actuated. It gets its name from the similarity to a gymnast (acrobat) on the horizontal bar. It is also a common control problem, where the aim is to swing up both links to the upright position.

Section 5-2-1 describes the experimental setup and shows which settings allow for real-time application of RTOPS. Section 5-2-2 shows the results of some experiments conducted with these settings.

5-2-1 Experiment design

Experimental setup

A schematic of the acrobot setup is shown in Figure 5-18. The acrobot has a four-dimensional state-space, where the following states are used:

\[
\mathbf{x} = \begin{bmatrix} \theta_1 \\ \dot{\theta}_1 \\ \theta_2 \\ \dot{\theta}_2 \end{bmatrix},
\]  

\text{(5-5)}
where $\theta_1$ is the angle in radians of the first link (the “body”) with respect to the vertical axis, $\theta_2$ is the angle in radians of the second link (the “legs”) also with respect to the vertical axis. $\dot{\theta}_1$ and $\dot{\theta}_2$ are the angular velocity of the first and second link, respectively.

The dynamics of the system are described by the following set of equations [22]:

$$
\begin{bmatrix}
\dot{\theta}_1 \\
\dot{\theta}_2 
\end{bmatrix} = \begin{bmatrix}
a_{11} & a_{12} \\
a_{21} & a_{22}
\end{bmatrix}^{-1} \begin{bmatrix}
b_1 \\
b_2
\end{bmatrix},
$$

(5-6)

where

$$
a_{11} = \left(\frac{4}{3}m_1 + 4m_2\right)l_1^2 \\
a_{22} = \frac{4}{3}m_2l_2^2 \\
a_{12} = a_{21} = 2m_2l_1 l_2 \cos(\theta_1 - \theta_2) \\
b_1 = 2m_2l_1 l_2 \dot{\theta}_2^2 \sin(\theta_2 - \theta_1) + (m_1 + 2m_2)l_1 g \sin(\theta_1) - \mu_1 \dot{\theta}_1 - u \\
b_2 = 2m_2 l_2 l_1 \dot{\theta}_1^2 \sin(\theta_1 - \theta_2) + m_2 l_2 g \sin(\theta_2) - \mu_2 \dot{\theta}_2 + u
$$

Here, $l_1 = 0.5$ m is the half-length of the first link, $l_2 = 0.5$ m is the half-length of the second link, $m_1 = 1$ kg and $m_2 = 1$ kg are the mass of the first and second link, respectively, $g = 9.81$ m/s$^2$ is the gravity acceleration and $\mu_1 = 0.05$ and $\mu_2 = 0.05$ are the friction coefficients of the links.

The goal in the experiments with the acrobot simulation is to swing up both links. The reward function used to express this is [21]:

$$
r_k = 1 - \frac{\sqrt{y_k - (l_1 + l_2)^2 + x_k^2}}{2(l_1 + l_2)},
$$

(5-7)
with

\[ x_k = l_1 \sin(\theta_1) + l_2 \sin(\theta_2) \]
\[ y_k = l_1 \cos(\theta_1) + l_2 \cos(\theta_2), \]

(5-8)

This reward function is constructed such that all rewards are in the interval \([0, 1]\).

The control action is a torque applied to the middle joint and the action set used of size \(K = 3\) is \(u \in \{-2, 0, 2\}\). In addition, note that the sampling time in these experiments was set to \(T_s = 0.1\) seconds.

**Finding computational complexity**

The same method as described for the cart-pole simulation is used here to find the computational time per node expansion \(T_c\).

**Feasible settings for \(n\) and \(d'\)**

For the acrobot simulation, we only do experiments in the setting with fixed budget \(n\) and fixed length \(d'\) of the applied sub-sequence, because this setting proved to clearly outperform the other settings in the cart-pole simulations.

Again, the worst-case condition derived in Chapter 4 (see Equation (4-17)) is used to find the combinations of \(n\) and \(d'\) that allow for real-time application of RTOPS. The possible combinations are shown in Figure 5-19.

**5-2-2 Experimental results**

First, the results of the scenario where a perfect model is available are shown. Secondly, results are also shown for experiments where a model mismatch was introduced.

**Perfect model**

All feasible lengths for the applied sub-sequence, as taken from Figure 5-19 have been used, in combination with the maximum allowed budget \(n\) for that choice of \(d'\). The results of all these experiments are shown in Table 5-5, where the columns are sorted by descending discounted sum of rewards. The relation between the length of the applied sub-sequence and the performance is not as clear here as it was for the (simpler) cart-pole system. One thing that is noticeable is that the three settings giving the worst performance, are the three settings where the applied sub-sequence is the longest. That means that also for this system, even when a perfect model is available, applying relatively many actions before re-planning does not seem favorable.

For comparison, the reward- and state-trajectory resulting from the experiments with the best performing setting \((d' = 3)\) and one of the worst performing settings \((d' = 4)\) are shown in Figures 5-20 and 5-21, respectively. We choose \(d' = 4\) instead of for example the worse performing \(d' = 11\), because it shows that applying just one action more can already have
The gray area shows all possible combinations of $n$ and $d'$ for which the worst-case condition for real-time applicability of RTOPS holds. Note that $d'$ can only be chosen on one of the dotted lines within the gray area. This is because fractional sequence lengths have no meaning in reality.

quite a significant impact on the performance. In Figure 5-21 it can actually be seen that the goal of keeping both links upright is not met.

Because the experiments on the cart-pole system confirmed the intuition that in the presence of model mismatches, the shorter the applied sub-sequence, the better the performance, the results for $d' = 2$ are also included (see Figure 5-22) to show that this setting performed quite well.

**Model mismatch**

Several experiments for different kinds of model parameters errors have been conducted. Figures 5-23 and 5-24 show the results of RTOPS using $d' = 2$ and $d' = 3$, respectively, where the mass and length of both links were modeled too low. In this case $d' = 2$ performed slightly better, but $d' = 3$ also shows good performance.

In Table 5-6 the results for different model parameter errors are shown for both $d' = 2$ and $d' = 3$. The interesting thing is that neither $d' = 2$ nor $d' = 3$ performs consistently better. The model mismatch seems to determine which of the settings outperforms the other, with one particular model mismatch even giving the exact same performance in both settings. The results of experiments with the model mismatch for which the performance differs most between the two settings are shown in Figures 5-25 and 5-26. In these figures it can be seen that even though the performance with $d' = 2$ is not as good at first, it is able to keep both links positioned upright after one “miss”.

M.H. Wensveen

Master of Science Thesis
Table 5-5: Discounted sum of rewards (over 100 steps) obtained in the experiments with fixed \( n \) and fixed sub-sequence length \( d' \) on the acrobot simulation. Note that the table is sorted by highest rewards, and the best performing settings are in bold.

<table>
<thead>
<tr>
<th>( n )</th>
<th>( d' )</th>
<th>Discounted sum of rewards</th>
</tr>
</thead>
<tbody>
<tr>
<td>82950</td>
<td>3</td>
<td>46.34372</td>
</tr>
<tr>
<td>138300</td>
<td>5</td>
<td>46.34372</td>
</tr>
<tr>
<td>193600</td>
<td>7</td>
<td>46.30856</td>
</tr>
<tr>
<td>55300</td>
<td>2</td>
<td>46.26664</td>
</tr>
<tr>
<td>221200</td>
<td>8</td>
<td>46.25057</td>
</tr>
<tr>
<td>165900</td>
<td>6</td>
<td>46.14592</td>
</tr>
<tr>
<td>248900</td>
<td>9</td>
<td>45.28947</td>
</tr>
<tr>
<td>110600</td>
<td>4</td>
<td>40.41503</td>
</tr>
<tr>
<td>276600</td>
<td>10</td>
<td>40.22135</td>
</tr>
<tr>
<td>338477</td>
<td>12</td>
<td>39.38478</td>
</tr>
<tr>
<td>304200</td>
<td>11</td>
<td>39.12224</td>
</tr>
</tbody>
</table>

Table 5-6: Comparison of performance between \( d' = 2 \) and \( d' = 3 \) for different model parameter errors. The parameter errors are introduced on the mass and length of the poles. Four different settings are used for the parameters of the first link/second link: modeled too low/low, too low/high, too high/low, too high/high. For the first link “low” means \( m_1 \times 0.95 \) and \( l_1 \times 0.99 \) and “high” means \( m_1 \times 1.02 \) and \( l_1 \times 1.02 \). For the second link “low” is \( m_2 \times 0.97 \) and \( l_2 \times 0.6 \) and “high” \( m_2 \times 1.03 \) and \( l_2 \times 1.01 \).

<table>
<thead>
<tr>
<th>( d' \setminus (m_1, l_1/m_2, l_2) )</th>
<th>low/low</th>
<th>low/high</th>
<th>high/low</th>
<th>high/high</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 (( n = 55310 ))</td>
<td>48.2293</td>
<td>46.25057</td>
<td>44.8735</td>
<td>45.9687</td>
</tr>
<tr>
<td>3 (( n = 82950 ))</td>
<td>48.0277</td>
<td>46.2731</td>
<td>47.4749</td>
<td>45.9687</td>
</tr>
</tbody>
</table>
Figure 5-20: Results of experiments on the acrobot simulation with $n = 82950$ and $d' = 3$. The number of actions was $K = 3$ and the discounted rewards obtained over 100 steps is 46.334.

Figure 5-21: Results of experiments on the acrobot simulation with $n = 110600$ and $d' = 4$. The number of actions was $K = 3$ and the discounted rewards obtained over 100 steps is 40.415.

Figure 5-22: Results of experiments on the acrobot simulation with $n = 55300$ and $d' = 2$. The number of actions was $K = 3$ and the discounted rewards obtained over 100 steps is 46.267.
Figure 5-23: Results of experiments on the acrobot simulation with $n = 55310$ and $d' = 2$. The model mismatch was as follows: length first link $\times 0.99$, mass first link $\times 0.95$, length second link $\times 0.97$, mass second link $\times 0.6$.

Figure 5-24: Results of experiments on the acrobot simulation with $n = 82950$ and $d' = 3$. The model mismatch was as follows: length first link $\times 0.99$, mass first link $\times 0.95$, length second link $\times 0.97$, mass second link $\times 0.6$.

Figure 5-25: Results of experiments on the acrobot simulation with $n = 55310$ and $d' = 2$. The model mismatch was as follows: length first link $\times 1.02$, mass first link $\times 1.02$, length second link $\times 0.97$, mass second link $\times 0.6$.

Figure 5-26: Results of experiments on the acrobot simulation with $n = 82950$ and $d' = 3$. The model mismatch was as follows: length first link $\times 1.02$, mass first link $\times 1.02$, length second link $\times 0.97$, mass second link $\times 0.6$. 
Figure 5-27: Experimental setup: DC motor as inverted pendulum. The aim is to position the weight at the top (i.e. 180 degrees rotated from the position in the picture.)

5-3 Real inverted pendulum

5-3-1 Experiment design

Experimental setup

A DC motor setup of TU Delft is used to create an inverted pendulum, by attaching an additional mass at the edge of the rotating disc. A picture of the setup is shown in Figure 5-27, while a schematic of the DC motor is shown in Figure 5-28. A two-dimensional state-space is used:

$$\mathbf{x} = \begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix},$$

(5-9)

where $\theta$ is the angle in radians of the virtual pole with respect to the vertical axis, and $\dot{\theta}$ is its angular velocity. Both $\theta$ and $\dot{\theta}$ can be measured directly from the system.

The control action is the armature voltage and is limited to a maximum voltage $u_{\text{max}}$. Several experiments have been done with different maximum voltages, with the action sets of size $K = 3$ being $u \in \{-u_{\text{max}}, 0.0, u_{\text{max}}\}$. 
The dynamics of this inverted pendulum are described by the following equation of motion:

\[ \ddot{\theta} = \frac{mg \sin(\theta) - (b + \frac{K^2}{R}) \dot{\theta} + K_m u}{J} , \]  

(5-10)

where \( m = 0.03 \) kg is the mass of the attached weight, \( g = 9.81 \) m/s\(^2\) is the gravity acceleration, \( l = 0.042 \) m is the distance between the center of the disc and the center of the weight, \( b = 3.0 \times 10^{-6} \) Nms/rad is the damping coefficient, \( K = 53.6 \times 10^{-3} \) Nm/A is the torque constant, \( R = 9.50 \) Ω is the rotor resistance, \( K_m = \frac{K}{R} \) and \( J = 10 \times 10^{-5} \) kg·m\(^2\) is the moment of inertia of the rotor. The sampling time in the experiments with this system was set to \( T_s = 0.05 \) seconds.

The numerical integration method used to simulate the state transitions is Runge-Kutta 4\(^{th}\) order. This method proved to result in simulation results close to the actual system.

The goal of the experiments on this system is to swing up the weight and keep it at the top. The reward function used to achieve this goal is the following:

\[ r_k = (\cos(\theta) + 1.0) \times 0.5 , \]  

(5-11)

which restricts all rewards to the interval \([0, 1]\).

**Real-time implementation**

When using RTOPS on a real system, some practical issues need to be tackled. The biggest issues that arise in practice is the real-time synchronization and the fact that the planning and applying of actions needs to be done concurrently.

The approach taken here is to run RTOPS and the I/O-interface with the system in two separate processes and have them communicate through a UDP socket. Table 5-7 shows the activities that are done by both the RTOPS process and the system’s I/O process. Note that activities in the same row are done in parallel. A visual representation of the time-line is also shown in Figure 5-29.

**Finding computational complexity**

Several iterations of OPD – without sending any actions to the system – have been done to find the time per expansions \( T_e \). A reward function is used here to force \( \kappa = 1 \), to get
Table 5-7: Sequence of activities done by the RTOPS process and I/O interface process.

<table>
<thead>
<tr>
<th>step</th>
<th>RTOPS</th>
<th>I/O system</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>initialization</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(includes setting first $d'$ actions to zero)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>start RTOPS process</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>initialization</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>go into ‘receiving’ state</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>receive state measurements</td>
<td>send state measurements</td>
</tr>
<tr>
<td>6</td>
<td>use OPD to find action sequence</td>
<td>apply $d'$ actions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(one every $T_s$ seconds)</td>
</tr>
<tr>
<td>7</td>
<td>go into ‘receiving’ state</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>receive ‘OK’ signal</td>
<td>send ‘OK’ signal (to inform controller actions can be sent)</td>
</tr>
<tr>
<td>9</td>
<td>go into ‘receiving’ state</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>send action sequence</td>
<td>receive action sequence</td>
</tr>
<tr>
<td>11</td>
<td>go back to step 4</td>
<td>go back to step 4</td>
</tr>
</tbody>
</table>

Figure 5-29: Time-line of real-time application of RTOPS.
Figure 5-30: The gray area shows all possible combinations of $n$ and $d'$ for which the worst-case condition for real-time applicability of RTOPS holds. Note that $d'$ can only be chosen on one of the dotted lines within the gray area. This is because fractional sequence lengths have no meaning in reality.

Feasible settings $n$ and $d'$

In the same way as done for the other experiments discussed above, all combinations of the budget $n$ and the applied sequence length $d'$ that allow for real-time application of RTOPS are found. Figure 5-30 shows these combinations.

5-3-2 Experimental results

Since this is a real-life system, a perfect model does not exist and the exact model mismatch is not known. We do know, however, that the quality of the model will influence the performance of RTOPS. The model available for this system is known to be relatively accurate.

Several experiments have been conducted with $n = 1666$ and $d' = 2$, for different values of $u_{\text{max}}$. All of these experiments showed very decent performance, without violating real-time constraints. The results of two experiments, for $u_{\text{max}} = 1.5$ and $u_{\text{max}} = 0.9$, are shown in

\footnote{The experiments are done using an HP 8510w laptop computer from 2008. The CPU is an Intel Core 2 Duo T9300, operating at 2.50GHz.}
Figure 5-31: The angle- and reward-trajectory resulting from applying RTOPS with $n = 1666$ and $d' = 2$ on the real inverted pendulum system, with the maximum applied voltage $u_{\text{max}} = 1.5$.

Figure 5-31 and 5-32, respectively, where it is seen that RTOPS can indeed control the weight to the upright position.

The interesting difference between the two experiments, is that for the lower $u_{\text{max}}$ the control is still successful, even though as much as five swings are require to get the weight in the desired position. This tells us that RTOPS is able to search over a long enough horizon to plan the swings.

In addition, experiments have been done with a faster sampling frequency of 50 Hz (i.e. $T_s = 0.02$ seconds). The faster sampling means that less time is available for planning. To compensate for this, a larger length is used for the applied sub-sequence, namely $d' = 5$. The maximum budget $n$ in this case was the same as in the previous experiments, $n = 1666$. The experiments showed that RTOPS is not able to reach a large enough depth in this case to be able to swing up the pendulum with a small maximum input $u_{\text{max}}$. With a small $u_{\text{max}}$, several swings are needed and therefore, planning over a longer horizon is needed. While the chosen budget is the same as in the previous experiments, the shorter sampling period means that reaching a depth $d$ with planning corresponds to a shorter time horizon as with a longer sampling period. Therefore, even if the same depth is reached as in the experiments with higher $T_s$, this does not guarantee that the time horizon that the depth corresponds to, is long enough to plan the necessary swings. The result is that RTOPS always returns the maximum control input, keeping the pendulum at a constant angle. Manually giving the pendulum one push at this point (in the opposite direction) does result in RTOPS finding actions that keep the pendulum upright, as shown in Figure 5-33.

With the faster sampling frequency, RTOPS does manage to control the pendulum to the upright position when the maximum input voltage is high enough to swing up the pendulum in one go. The relatively high input voltage, in combination with the longer applied sequence, does result in more oscillation around the desired position. See Figure 5-34 for these results.
Figure 5-32: The angle- and reward-trajectory resulting from applying RTOPS with $n = 1666$ and $d' = 2$ on the real inverted pendulum system, with the maximum applied voltage $u_{\text{max}} = 0.9$. Because of the low maximum voltage, several swing are needed to achieve the goal.

Figure 5-33: The angle- and reward-trajectory resulting from applying RTOPS with $n = 1666$ and $d' = 5$ on the real inverted pendulum system with $T_s = 0.02$ and the maximum applied voltage $u_{\text{max}} = 1.5$. When RTOPS fails to swing up the pendulum, a push is given, after which RTOPS does manage to keep the pendulum upright.
Implementation and Experiments for RTOPS

Figure 5-34: The angle- and reward-trajectory resulting from applying RTOPS with $n = 1666$ and $d' = 5$ on the real inverted pendulum system with $T_s = 0.02$ and the maximum applied voltage $u_{max} = 2.5$. While RTOPS is able is swing up the pendulum, there is a low of oscillation around the setpoint.

Near-optimality analysis To really assess the performance of RTOPS on this inverted pendulum system, the solution must be compared to a solution that is known to be (close to) optimal. Fuzzy Q-iteration (fuzzy QI) [24] is used to find a near-optimal policy and the policy is then used to control the system and obtain the (discounted) rewards. The rewards obtained by using the near-optimal policy are then compared to the rewards obtained by using our RTOPS algorithm. This is done for both $u_{max} = 0.9$ and $u_{max} = 1.5$, with $T_s = 0.05$, and the compared results can be seen in Figures 5-35 and 5-36, respectively.

We also take a closer look at the discounted rewards obtained by both RTOPS and fuzzy QI. Although the infinite-horizon discounted rewards can not be found in practice, an approximation is shown in Table 5-8.

The infinite-horizon value is approximated by running the experiments long enough such that the possible additional rewards are negligible. In this case the experiments were run for 60 seconds, which is an action sequence of length $d = 1200$. Therefore, the future rewards are at most $\gamma^d = 0.006$, which is considered negligible. Denoting the value obtained with the fuzzy QI policy by $\bar{\nu}_x(\bar{u}_d)$, we find the sub-optimality of RTOPS compared to fuzzy QI to be $\bar{\nu}_x(\bar{u}_d) - \nu_x(u_d) = 0.5 < \frac{7.37}{1-\gamma}$.

Note that for these calculations only actions found by OPD were used, i.e. the first $d'$ actions, that were set to zero by default, are ignored.

The same is done for $u_{max} = 1.5$, the results of which are also shown in Table 5-8. In this case the sub-optimality of RTOPS compared to fuzzy QI is $\bar{\nu}_x(\bar{u}_d) - \nu_x(u_d) = 0.0612 < \frac{7.37}{1-\gamma}$.

M.H. Wensveen Master of Science Thesis
Figure 5-35: Comparison of the results from using fuzzy QI and RTOPS for $u_{\text{max}} = 0.9$. Note that the angle measurements shown in the upper row are mirrored. This is not strange, however, because there is no difference between swinging clock-wise or counter clock-wise.

Figure 5-36: Comparison of the results from using fuzzy QI and RTOPS for $u_{\text{max}} = 1.5$.

Table 5-8: Comparison between approximations of the infinite-horizon discounted rewards obtained with fuzzy QI and with RTOPS.

<table>
<thead>
<tr>
<th>$u_{\text{max}}$</th>
<th>Fuzzy QI</th>
<th>RTOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9</td>
<td>68.8578</td>
<td>68.3578</td>
</tr>
<tr>
<td>1.5</td>
<td>88.5934</td>
<td>88.5322</td>
</tr>
</tbody>
</table>
5-4 Summary and concluding remarks

Experiments with RTOPS have been conducted on three different systems: a simulated cart-pole system, a simulated acrobot system and an inverted pendulum system.

The first step in each experiment is to measure the computational time per node expansion $T_e$, which can be used to find all settings of RTOPS that meet the real-time applicability conditions as derived in Chapter 4. To find $T_e$, a preliminary experiment is needed. It is shown that, because of the dependency of OPD’s complexity on branching factor $\kappa$, the “safest”, worst-case measurement of $T_e$ is found by using in the preliminary experiment a reward function that always favors one action. Using such a reward function, one can find a good measurement of $T_e$ by running an iteration of OPD with a fixed computational budget and dividing the total measured time (excluding any one-time initializations) by the number of expansions.

The measured $T_e$ can be used to compute the combinations of $d/d'$, $n/\alpha$ and $n/d'$ for the three introduced settings of RTOPS, respectively. For the cart-pole simulation all three settings have been used to analyze the practical performance of RTOPS. It is shown that, with a perfect model, for the fixed $d/d'$ case the best performing settings were those with large target depth $d$ and/or short applied sub-sequences. Large length of the applied sub-sequences performed relatively poorly. In the fixed $n/\alpha$ setting a result from [6] – stating that in some cases longer applied sequences perform better, while in other cases shorter sequences can give higher performance – is confirmed in the experiments. It is also shown that this setting in general performs better than the fixed $d$ setting, because the depths reached with a fixed budget $n$ are on average much larger than the fixed $d$. The third setting, where along with the budget $n$, also the length of the applied sequence is fixed, is shown to always outperform the fixed $d$ setting and to mostly outperform the fixed $n/\alpha$ setting.

For the remaining two experiments, only the fixed $n/d'$ setting is used, because it proved to be the best performing setting in the cart-pole experiments. In the acrobot simulation it is shown that the longest applied sequences gave the worst performance, even when a perfect model was used. It is also shown that the performance can be very different for action sequences of length $d'$ than for sequences of length $d' + 1$.

For the two simulated experiments, the effects of different model mismatches have also been examined. In these experiments it is shown e.g. that neither longer nor shorter applied sequences perform consistently better, with instead the better performing setting being determined by the exact model mismatch. In addition, in the cart-pole simulation the fixed $n/\alpha$ setting was shown not be be a suitable setting when model mismatches are present.

On the inverted pendulum system RTOPS is proven to indeed be applicable in real-time and to give near-optimal performance. This is shown by comparing the results obtained with RTOPS (used in the fixed $n/d'$ setting) to the results obtained with a near-optimal policy. RTOPS is shown to also be able to control the pendulum even when the maximum control action is low, such that several swings are needed to position the pendulum upright. Note that since this is a real system, model mismatches are always present.

To conclude, various experiments have been conducted to show that RTOPS is applicable in real-time and that, depending on the settings, near-optimality is maintained.
Conclusion and Future Work

Two approaches have been investigated to make Optimistic Planning for Deterministic systems (OPD) applicable in real-time for the control of general nonlinear systems. The first approach is to decrease the computational time of OPD by parallelization of the algorithm, and the second method is to increase time available for planning by using sequences of actions instead of single actions.

For parallelization, the focus was mainly on one method, the N-Best-First approach, as introduced for the classical search algorithm A*. An implementation of this approach for OPD, called N-Best-First Optimistic Planning (NBFOP), did not show any improvement in the computational speed. It is expected, however, that a more efficient implementation should be possible, which might lead to a decrease in computational time per node expansion.

The structure of the problem can also have an impact on the potential benefit from using the developed algorithm NBFOP. OPD targets mainly problems that feature a small proportion of near-optimal paths, i.e. that have a small relative branching factor $\kappa$. For problems with $\kappa = 1$, however, NBFOP is not expected to be of any use, because only one thread would be expanding interesting nodes, while the other expansions are useless. With the more common branching factor of $\kappa > 1$, on the other hand, enough interesting nodes would be available for all threads to perform useful expansions. Then, NBFOP could in theory be an improvement over serial OPD, in terms of computational time.

The second method for making OPD applicable in real-time was to increase the time available for planning. To achieve this, a sequence of actions is applied instead of just one action, as is typically done with OPD. When a sequence of length $d'$ is used, the time available for planning is increased by a factor $d'$. The resulting algorithm is called Real-Time Optimistic Planning with Action Sequences (RTOPS) and three different settings of the algorithm are introduced. One setting imposes a target planning depth and applies fixed length action sequences. The remaining two settings both impose a maximum number of node expansions, but differ in the length of the applied sub-sequence. One setting applies sequences with length proportional to the depth reached in the tree, the other setting applies sequences of fixed length.

For the three settings of RTOPS, conditions have been derived on the allowed choices of the target depth or computational budget and on the length of the applied sequences, that...
guarantee real-time applicability. A distinction is made between three cases regarding the relative branching factor $\kappa$. One of these cases, where $\kappa$ is equal to the number of available actions, provides worst-case conditions that are needed for most real problems, because $\kappa$ is usually unknown. These worst-case conditions guarantee real-time applicability of RTOPS for any optimal control problem where the action space is discrete and finite, and the state space can be infinite.

The effects of using sequences of actions instead of single actions on the performance of optimistic planning has been investigated. A bound has been derived on the performance loss from using longer or shorter sequences, when a perfect model is available. In addition, under certain system continuity and model accuracy assumptions, a bound on the sub-optimality of the algorithm when applied in closed-loop is given, when model mismatches are present.

The performance of RTOPS has also been analyzed in several experiments with three different problems: a simulation of a cart-pole system, a simulated acrobot and a real inverted pendulum. Experiments with the first system showed that a fixed computational budget outperforms a fixed target depth in most cases. It also shown, however, that in the presence of (simulated) model mismatches, the setting where the length of the applied sequence varies is not suitable. The setting with fixed budget and fixed length of the applied sequence does show good performance, also with model mismatches. A benefit of the latter setting is that it uses as much of the available time for planning as possible, while the fixed target depth and varying sequence length settings will waste time.

The expectation that a larger reached depth in the tree leads to better performance is confirmed in these experiments. Additionally, it is shown that in most cases shorter applied sequences give better performance than longer sequences.

The experiments on the real inverted pendulum prove that RTOPS can be applied in real-time on physical systems. The performance of RTOPS was compared to a known near-optimal solution, proving that RTOPS also finds near-optimal actions.

**Future work**

**More experiments**  To further analyze the performance and usefulness of RTOPS in practice, more extensive experiments should be conducted, and, more importantly, experiments on more complex systems are required. While there were some good experimental results on the real inverted pendulum and, in simulation, on the more complex acrobot, one can better judge the practical value of RTOPS when it is applied to more interesting real systems. Systems that could be used include for example a two-link robot arm, a 2-D or 3-D crane or navigation of a mobile robot.

Further experiments should also be done to better investigate the influence of the length of the applied sub-sequence on the performance of RTOPS with real systems. In addition, systems with faster sampling times could be used to see whether RTOPS is still able to control those systems, where the allowed settings are more restricted.

Because the computational time per node expansion is directly related to the processing power of the hardware used to run the controller, it will be beneficial to run any further experiments using a newer, faster PC. While the PC used for the experiments with the real inverted pendulum is not a low-end machine, it is over five years old and it is expected that
a newer high-end PC could introduce an increase in computational speed of at least a factor 2. For comparison the benchmark program SuperPI – which computes one million digits of \( \pi \) – was used [25]. The hardware used for the experiments on the real inverted pendulum computes the one million digits in 19.49 seconds, while it takes only around 7 seconds with a modern Core i7 Intel processor.

**Improving parallel implementations** Even though the results obtained with parallelization of OPD were not positive, it is believed that with more research and more efficient implementations, parallelization could in fact decrease the computational time per node expansion. Should an implementation of a parallel version of OPD be found that does lead to a lower time required to perform \( n \) node expansions, it could be beneficial to implement that parallelization on the RTOPS algorithm discussed in Chapter 4. It should then be noted that the parallelization effectively lowers the computational time per expansion \( T_e \), which would for example allow for a larger computational budget for a given length of the applied sub-sequence. That is, it relaxes the conditions derived in Chapter 4.

For example, if a parallel version of OPD would show a similar performance as Parallel Best-N Block-First (PBNF) in [10] – where a speedup of up to 6× was realized for 8 threads – the real-time applicability conditions could be relaxed by a factor six. As an example, consider Condition 4.8, the safe condition for fixed computational budget with fixed length of applied sub-sequence, which puts the following restriction on \( n \) and \( d' \):

\[
T_e \cdot n \leq T_s \cdot d'
\]

Any speed-up of \( N \) times would decrease the computational time per node expansion to \( T_{e,\text{par}} = \frac{1}{N} T_e \) and consequently increase the maximum allowed budget \( n \) for a given \( d' \) by a factor \( N \).

The fact that parallelization could potentially increase the allowed computational budget – and with that the near-optimality – combined with the generally good performance found with the use of sequences of action, indicates that a parallel version of RTOPS could make optimistic planning even more valuable in practice.

Because the parallel algorithm PBNF showed significant speed-up in [10], it would be worthwhile to continue the research on possible benefits from applying similar parallelizations to OPD and RTOPS. One of the issues that arose during the implementation of PBNF was the complexity of defining successor-predecessor relations of blocks beforehand. An idea that could solve such an issue, is to define these relationships adaptively, in an online fashion.

**Introduce additional setting for RTOPS** In Chapter 4 three different settings of RTOPS have been extensively analyzed:

- Fixed target depth \( d \)
- Fixed computational budget \( n \) with variable length of applied sequence \( d' = \lceil \alpha d \rceil \)
- Fixed computational budget \( n \) with fixed length of applied sequence \( d' \)
In addition to these settings there is one more setting that could be of interest. Instead of fixing the computational budget and varying the length of the applied sequence according to the depth reached in the tree, one could adapt the computational budget to the depth reached. That is, an initial budget $n$ could be chosen, together with an $\alpha$ that determines the length $d' = \lceil \alpha d \rceil$ of the applied sequence. However, instead of using the same $n$ on the next planning instance, $n$ could be adapted according to $d'$, where $n$ is set to the maximum allowable budget for that particular $d'$:

$$n \leq \frac{T}{T_e} d'$$

(6-1)

This means that, when a large depth is reached in the tree – and as a result a long subsequence is applied – the extra computational time that becomes available is all used by increasing the budget accordingly. In such a setting, a maximum budget $n$ (equivalently, maximum length $d'$) should probably be set, to avoid the budget and applied sequence length to “explode”. This setting would allow for varying length of the applied sub-sequences, while still guaranteeing that as much of the available time as possible is used during the planning instances.

**Different OP algorithms**  It could also be of interest to apply similar analyses as done in Chapter 4 to an Optimistic Planning (OP) algorithm for stochastic system, such as the ones described in [4] and [20]. The same approaches as used in RTOPS might be applicable to OP algorithms for stochastic settings, making those applicable in real-time.

Additionally, while the OP algorithm used throughout this work allowed only a set of discrete actions, OP algorithms exist that allow for continuous actions as well. One OP algorithm with continuous actions that showed good performance is SOOP [26]. Analyzing SOOP with a view to its real-time applicability as done for RTOPS could further increase the domain in which OP is of practical use.

Finally, a modification of OP that limits the number of action switches, called OSP, is proposed in [27]. Since the complexity of the proposed algorithm is reduced to a polynomial complexity, it will be easier to meet the real-time constraints. Applying a similar approach as used to develop RTOPS could allow OSP to work in real-time.


Glossary

List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MDP</td>
<td>Markov Decision Process</td>
</tr>
<tr>
<td>MPC</td>
<td>Model Predictive Control</td>
</tr>
<tr>
<td>NBFOP</td>
<td>N-Best-First Optimistic Planning</td>
</tr>
<tr>
<td>NBFS</td>
<td>N-Best-First Search</td>
</tr>
<tr>
<td>OP</td>
<td>Optimistic Planning</td>
</tr>
<tr>
<td>OPD</td>
<td>Optimistic Planning for Deterministic systems</td>
</tr>
<tr>
<td>PBNF</td>
<td>Parallel Best-NBlock-First</td>
</tr>
<tr>
<td>RTOPS</td>
<td>Real-Time Optimistic Planning with Action Sequences</td>
</tr>
</tbody>
</table>

List of Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_n$</td>
<td>Accumulated regret after $n$ node expansions</td>
</tr>
<tr>
<td>$U$</td>
<td>Action space</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>Bound on model mismatch</td>
</tr>
<tr>
<td>$\kappa(x)$</td>
<td>Branching factor of near-optimal subtree</td>
</tr>
<tr>
<td>$n$</td>
<td>Computational budget for node expansions</td>
</tr>
<tr>
<td>$T_e$</td>
<td>Computational time per node expansion</td>
</tr>
<tr>
<td>$c$</td>
<td>Constant</td>
</tr>
<tr>
<td>$u_k$</td>
<td>Control action at time instance $k$</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>Discount factor</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>Fraction of length of sequence to be applied</td>
</tr>
<tr>
<td>$d'$</td>
<td>Length of applied sub-sequence</td>
</tr>
</tbody>
</table>
\( L \)  
Lipschitz constant for model dynamics

\( G \)  
Lipschitz constant for reward function

\( \nu_{x}(u_d) \)  
Lower bound on value of action sequence starting with \( u_d \)

\( g \)  
Model dynamics

\( N \)  
Number of threads in parallel algorithm

\( d \)  
Planning depth/length of sequence returned by OP

\( \hat{x}_k \)  
Prediction of state at time instance \( k \)

\( r_k \)  
Reward at time instance \( k \)

\( \rho \)  
Reward function

\( T_s \)  
Sampling time

\( K \)  
Size of action space

\( X \)  
State space

\( x \)  
State vector

\( ^* \)  
Superscript to indicate optimal variable

\( f \)  
System dynamics

\( x_k \)  
System’s state at time instance \( k \)

\( v_{x}(u_d) \)  
True value of action sequence starting with \( u_d \) and continuing optimally

\( b_{x}(u_d) \)  
Upper bound on value of action sequence starting with \( u_d \)

\( V^\pi \)  
Value function w.r.t. policy \( \pi \)