Kinodynamic Steering using Supervised Learning in RRT

S. Moring

Master of Science Thesis

Delft Center for Systems and Control

Kinodynamic Steering using Supervised Learning in RRT

Master of Science Thesis

For the degree of Master of Science in Signals & Systems at Delft University of Technology

S. Moring

December 12, 2017

Faculty of Electrical Engineering, Mathematics & Computer Science · Delft University of Technology

Table of Contents

Chapter 1

Introduction

Since the beginning of robotics, robots and manipulators keep on becoming more common to relieve humans in many ways, with an expected 1.4 million new robots being installed in industry between 2016 and 2019[1]. An area that saw a rapid growth in the application of robotic manipulators is in manufacturing industries, leading to the industrial revolution [2]. Recently, with initiatives such as Industry 4.0, extensive effort is being devoted to collaborative robotic applications with humans and robots working together. As shown by the Robotics Industry Association (RIA), this trend is continuously increasing and it does not appear to stop for the time being[3].
The ultimate goal in robotics is having them operate alongside humans in a collaborative environment. For this the robot needs to be able to react to the people in its workspace, making each of the motion the robot has to make slightly different. The problem of finding these motions is called *motion planning.* Since each motion the robot makes could be different, the motion planning needs to be done by the robot itself, i.e. online. However, with current methods this would mean the robot would need computing for minutes, if not hours between each motion. A further complication is that these motions need be smooth and 'natural'-looking in order to create a comfortable work environment for the human. For safety, the velocities and forces or torques the robot exerts need to be limited as well. All these constraints lead to a difficult and thus computationally heavy problem. Speeding up the computation of this problem will be a step towards collaborative robotics, and will be the main subject of this thesis.

1-1 Motion Planning

As the name suggests, motion planning covers the planning of motion of robots and manipulators. When a robot wants to move from some initial pose x_i to some final pose x_f, a specific sequence of control inputs need to be applied to achieve this. On top of that the motion often has to comply with constraints, such as positions where it can not go due to items blocking its path or even configurations the robot simply cannot make. The problem of finding how to get from x_i to x_f while respecting these constraints is called *motion planning*. In practice planning is mostly done in a space called the *configuration space*. The configuration space $\mathcal{C}$ is a space which consists of parameters that describe the pose of a system. For example the pose of a 2-degrees-of-freedom manipulator can be described by the two angles that its joints make, $\{\vartheta_1, \vartheta_2\}$. The movement of a system is limited due to for example joint constraints and obstacles. This splits the configuration space into a free space $\mathcal{C}_{free}$ and an obstructed space $\mathcal{C}_{obs}$. When there are no constraints or obstacles, $\mathcal{C}_{free} = \mathcal{C}$, planning can easily be done by making use of for example path finding algorithms or potential fields. However a heavy limited space due to complex constraints and obstacles can make planning impossible using these methods[4], since they need an exact and full representation of $\mathcal{C}_{free}$ and $\mathcal{C}_{obs}$. In practice path finding algorithm or potential fields are not used because of this.

A principle which can deal with these difficult constraints and which is used most nowadays is called *sampling-based motion planning*, and in particular *Rapidly Exploring Random Trees* (RRT)[5][6]. In RRT, a tree is built through the planning space until a path exists from the starting state to the goal state. For every new node added to the tree, three steps are taken: *sampling*, *neighbor selection* and *steering*. First a random point x_r is sampled from the configuration space $\mathcal{C}$. The next step is to find a node in the tree which is nearest to the randomly sampled node, x_{near}, based on some distance metric, often the euclidean distance or optimal *cost to go*. A new node is created by steering the system from x_{near} towards x_r. After checking whether the reached node and the path towards it are within $\mathcal{C}_{free}$ it is added to the tree. Using this method planning in configuration space becomes a much easier problem to solve, since it does not require a full representation of $\mathcal{C}_{free}$ and $\mathcal{C}_{obs}$ beforehand. Planning in the configuration space $\mathcal{C}$ using RRT often leads to a solution, however these found motions are not necessarily executable by the robot, let alone that the motions are smooth as is wanted for collaborative robotics. Incorporating the velocities and dynamic constraints during planning could solve this. Doing so is called *kinodynamic* motion planning, and is achieved by planning in a different space $\mathcal{X}$ called the state space.

The state space is a space which contains both the positions and velocities of the system, and is subject to a differential constraint in the form of the equations of motion $\dot{x} = f(x, u)$ of the system. For example the state of earlier mentioned 2-DOF manipulator is now contains $\{\vartheta_1, \vartheta_2, \omega_1, \omega_2\}$, it also includes the angular velocities. Planning in state space could lead to more smooth and natural-looking motions since the robots dynamics are taken in account. Furthermore it provides an infrastructure where limiting the velocities and forces becomes relatively easy, leading to a safer workspace for humans. An additional possible advantage could be reduction of energy usage, since the dynamics of the robot can be taken advantage of. However, these dynamics also greatly complicate the building of a tree. Travelling through state space is much more difficult because of the differential constraints which often are highly non-linear, making kinodynamic planning much more time consuming than planning in configuration space[4]. If a robot needs to do the planning itself, necessary for collaborative robotics since the robot needs to work around moving humans, the motion planning needs to be done online. The kinodynamic planning problem is currently a too difficult problem to solve online, it takes too much time, and thus its benefits are not taken advantage of.

Many variants have been created since the introduction of RRT with the goal to speed up the computation. The state of the art algorithms that achieve large speedups all make use of supervised learning. The focus of this thesis will be the application of supervised learning in RRT, with the goal to speed up kinodynamic planning.

1-2 Supervised Learning

State of the art algorithms incorporate supervised learning in different steps of RRT, such as approximating the distance metric to speed up the neighbor selection[7][8] or to improve the quality of the random samples[9][10][11]. Supervised learning is a collection of methods which are used to approximate functions or mappings from a dataset[12]. It is often used to approximate very complex functions or mappings, reducing the computation time. This is done by creating a model using input and output data coming from the true relationship, called labelled training data. This data can be retrieved from measurements or generated using the true complex function. Given a dataset of input data X, which is a subset of the set of all possible inputs $\mathcal{X}$, and a corresponding dataset of output data Y, which as well is a subset of all possible outputs $\mathcal{Y}$, then supervised learning can be described as:

Using input data $X \in \mathcal{X}$ and corresponding output data $Y \in \mathcal{Y}$, find a mapping $\hat{Y} = \hat{f}(X)$ such that $e(\hat{f}(X), Y)$ is minimized, where $e(\hat{f}(X), Y)$ is some scoring function.

A lot of methods have been developed over the years, starting from simple linear regression models up to very complex neural networks[12][13], the latter becoming more popular after the development of the backpropagation algorithm in 1975, which greatly sped up the training of multilayer networks. A detailed description of the methods that have been succesfully applied in RRT and that are relevant to this study on speeding up kinodynamic planning are given in Chapter 2.

1-3 RRT CoLearn

The most recent development in kinodynamic planning and the starting point for this research is called RRT CoLearn and applies supervised learning to appproximate two of the three main steps in RRT. Both the steering function and the distance metric[14] are approximated using a method called k-Nearest Neighbor. In RRT CoLearn, a dataset is created with a lot of small trajectories using indirect optimal control. From this dataset the algorithm tries to learn how to steer the system from some state x_i to another state x_f, by learning the initial costates from the trajectories in the dataset. In terms of the above definition of supervised learning, X would contain the trajectories in the form of the initial and final state; $\{x_i, x_f\}$, and Y would contain the initial costates and the optimal cost to go; $\{\lambda_i, c\}$. Hence the supervised learning method learns the mapping $f : \{x_i, x_f\} \mapsto \{\lambda_i, c\}$. The initial costates are a property of indirect optimal control which dictate in what direction the system will move.
The algorithm was applied on a single pendulum swing up problem, which was solved on average within 2.4s, ten times faster than state of the art kinodynamic planning algorithms. It is however the only problem that it has been applied on so far, how the algorithm performs on more complex planning problems with more degrees of freedom is not yet known. Assessing the performance of RRT CoLearn on these systems is the starting point of this thesis.

1-4 Research

The main focus of this thesis is the application of supervised learning in RRT. RRT CoLearn is the state of the art algorithm applying supervised learning, so this will be the starting point. Since it has only been applied on a single pendulum yet, the first part of the research tries to extend the algorithm to 2-DOF system. RRT CoLearn is applied and tested on the swing up problem of a cart pole system in order to answer the following research question:

1. *What are the consequences for RRT CoLearn when it is applied to a 2-DOF system?*
   - 1.1. *What are the consequences for the data generation?*
   - 1.2. *What effect do more dimension have on the computational time?*

During this research many different configurations of the algorithm have been tested together with a lot of slight adjustments to the algorithm in order to make the algorithm converge to a solution. Since none of the adjustments on RRT CoLearn led to a solution for the swing up problem, the research broadened with a more general research goal:

> *Use supervised learning in RRT to solve a kinodynamic planning problem of a 2-DOF system.*

Pursuing this goal the research deviated from using indirect optimal control as was done in RRT CoLearn and a new steering function was created based on learning the inverse dynamics of a system based on artificial neural networks. Using this new steering function the swing up problem of the cart pole system was easily solved.

1-5 Thesis outline

This thesis starts in Chapter 2 with theoretical background on RRT and supervised learning. It handles multiple variants of RRT and the state of the art methods in supervised learning. Readers familiar with these subjects can continue to Chapter 3 which handles the investigation into RRT CoLearn. In Chapter 4 a new steering function is presented which is based on learning the inverse dynamics of the underlying system making using neural networks. The final chapter, Chapter 5, concludes the thesis with a conclusion and recommendations.

Chapter 2

Related Research

A lot of effort has been put in decreasing the computation time of kinodynamic planning, as already briefly introduced in the introduction. RRT is regarded as the state of the art in solving this problem relatively quickly. This chapter handles related research and is split up into three sections. Section 2-1 handles the RRT algorithm and many of its variants which speed up the algorithm by using supervised learning. Section 2-2 handles the RRT CoLearn algorithm, which is also the main focus of Chapter 3. Section 2-3 handles the supervised learning methods which are used in the research. Readers who are familiar with these subject can skip these sections.

2-1 Rapidly Expanding Random Tree

In the RRT algorithm the kinodynamic motion planning problem is solved by building a tree through the free state space $\mathcal{X}_{free}$[4][5][6]. The origin of the tree is based at the initial state x_i, from which it attempts to explore the free state space. The algorithm returns when a tree node is added which is inside the goal region $\mathcal{X}_{goal}$. The process of adding a new node to the tree consists of three main parts: *sampling*, *neighbor selection* and *steering*. When the environment the robot operates in is not free, i.e. $\mathcal{X}_{free} \neq \mathcal{X}$, an addition fourth step is added called *collision detection*. The vanilla RRT algorithm is outlined in Algorithm 1.

Sampling The simplest step of the RRT algorithm is the sampling. In this step a random state x_r is drawn from the free state space $\mathcal{X}_{free}$. The easiest way of sampling is drawing uniformly from $\mathcal{X}_{free}$. Usually a bias is introduced towards the goal state x_g by drawing a sample from $\mathcal{X}_{goal}$ a certain percentage of the time[5][6], which improves the convergence speed of the algorithm as it introduces greed into the algorithm.

Neighbor Selection After a random state x_r is sampled, it needs to be connected to the tree. The process of finding the optimal node to connect to is called neighbor selection, and is done

Algorithm 1: Basic RRT Outline

```
Input:
$x_i$: Initial state
$\mathcal{X}_{goal}$: Goal region
$\mathcal{T} = \{x_i\}$
while $\mathcal{T} \cap \mathcal{X}_{goal} = \emptyset$ do
    $x_r$ = sample($\mathcal{X}_{free}$)
    $x_{near}$ = findNeighbor($\mathcal{T}$,$x_r$)
    $x_{new}$ = steer($x_{near}$,$x_r$)
    if noCollision($x_{new}$,$E$) then
        if $x_{new} \in \mathcal{X}_{goal}$ then
            return $\{x_i, \cdots, x_{new}\}$
        else
            $\mathcal{T} = \mathcal{T} \cup x_{new}$
        end
    end
end
```

by minimizing some distance metric. A typical distance metric often used is the Euclidian distance as shown in Equation 2-1.

$$d(x, x_r) = \sqrt{(x - x_r)^T(x - x_r)} \tag{2-1}$$

In kinematic planning this makes sense, since only positions are regarded and the Euclidean distance then is the actual distance. However when planning in state space, the definition of distance is not that easy. For example what is the distance between two angular velocities ω_1 and ω_2? In Figure 2-1 a swing up trajectory for an underactuated single pendulum is shown, with two points on the trajectory markerd. Eventhough the two points are spacially near eachother, to move from x_1 to x_2 an entire extra swing is needed. Hence in kinodynamic sense these points are actually far apart. Therefore in [6] and [15] it is suggested to use the optimal *cost-to-go* as distance metric. However, the optimal cost-to-go is a heuristic that is difficult and thus time consuming to compute. Instead of exactly computing the cost-to-go, it can be approximated as well using supervised learning[16][7][8][14]. State of the art methods that do so will be handled in the remainder of this section. In Chapter 4 the distance metric is addressed as well.

Steering After a random state x_r and its nearest neighbor x_{near} are selected, the system needs be *steered* from x_{near} towards x_r. Steering means finding a control input such that when applied to the system with initial state x_{near}, the state evolves towards a new final state x_f, which is closer to the random sample x_r than its neighbor x_{near}. In the original RRT steering is done in a crude way. The allowable input set $\mathcal{U}$ is assumed to be discrete, for example $\{u_{min}, 0, u_{max}\}$, and all inputs are applied to the system[5][6]. The control input that minimizes the distance between the reached state x_f and the sampled state x_r is selected, and the corresponding x_f is added to the tree.

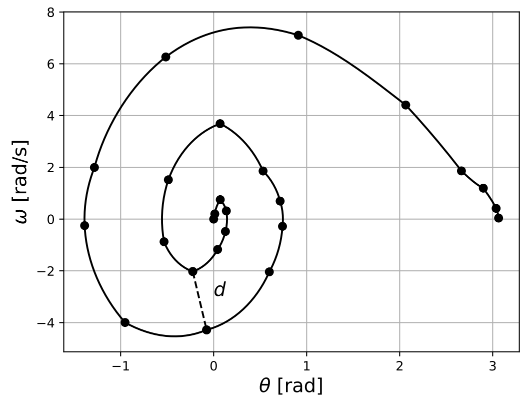


Figure 2-1: Swing up trajectory of a pendulum. The euclidean distance d is small, however since direct travel is not possible, the actual distance is far.

Collision Detection If the space is planned in is not free of possible collisions, i.e. $\mathcal{X}_{free} \neq \mathcal{X}$, an additional step is required called collision detection. Not only the reached state x_f, but also the trajectory from x_{near} to x_f need to be collision free. Even when a full geometric description is available of the environment and all the objects in the surrounding, finding collission is a computationally heavy process[17]. However, in this thesis the state space is considered to be entirely free, i.e. $\mathcal{X}_{free} = \mathcal{X}$ and the collision detection step will be further ignored.

2-1-1 Properties

Due to the random sampling the state space, the expansion of the tree is biased outwards, making it expand rapidly. This outward bias is called the *Voronoi bias.* Consider Figure 2-2. Every node in the tree has its own voronoi region. When sampling uniformly over the state space, the probability of sampling a node in a large voronoi region is larger than that of sampling in a small one. In practice this means that it is more likely to sample a new random state x_r in an area that has not been explored yet and thus the Voronoi bias stimulates the tree to extend outwards and explore the unexplored regions. This outward bias is clearly visible in Figure 2-3. Another nice property of RRTs is its *probablistic completeness.* In [6] it is proven that the basic RRT algorithm fulfills this property. Probabilistic completeness means that the probability of the algorithm finding a path approaches 1, as the number of randomly taken samples increases. This of course assuming that there exists a feasible path. This means that, if there exists a path from x_i to x_f, the RRT algorithm is guaranteed to find it. It is however a fragile property, minor changes to the algorithm can already take it away.

So far only the 'Vanilla' version of RRT has been handled. Many RRT variants have been developed, the fastest ones being RRT which makes use of supervised learning. The remainder of this section handles different approaches in speeding up the RRT algorithm.

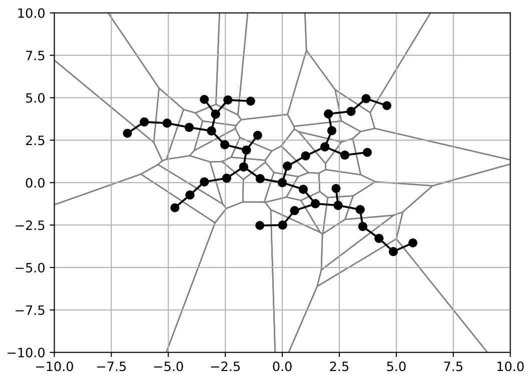


Figure 2-2: 40 nodes of an RRT with $x_i = (0, 0)$, and no goal state. Each node has its Voronoi region shown. Nodes on the outside of the tree have larger Voronoi regions.

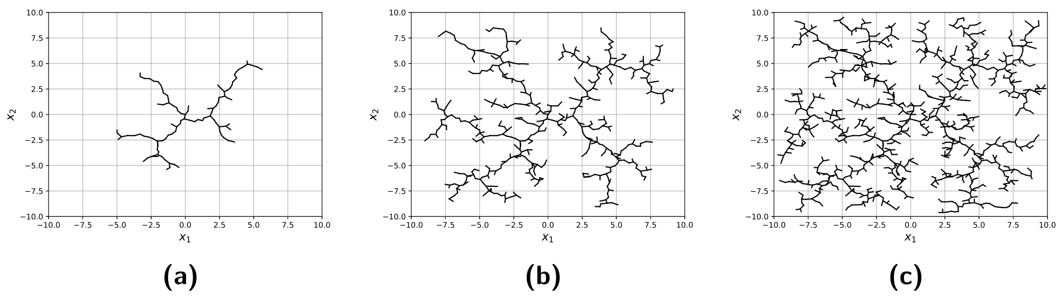


Figure 2-3: Expansion of an RRT for a system with dynamics $\dot{x_1} = u_1$ and $\dot{x_2} = u_2$. The tree starts at $x_i = (0, 0)$, and there is no goal state. Figure (a) has 100 nodes, (b) has 500 nodes and (c) has 1000 nodes. The outwards expansion towards unexplored regions, caused by the Voronoi bias, is clearly visible.

2-1-2 Parallel RRT

A popular method of speeding up the RRT is by breaking the problem up into smaller pieces such that the computations can be done in parallel.

RRT Connect

Instead of building a single tree from the initial state x_i, RRT Connect builds two trees: One tree starting from the initial state x_i and one tree starting from the goal state x_{goal}[18]. The two trees are built in the same way as the original RRT, the only difference is that the tree starting from the goal state x_f is connected backwards. When two nodes, one of each tree, are close enough to eachother, the trees are connected via these nodes. The idea of using two trees was already presented in [5], but the building of the tree was done completely random. In RRT-Connect the trees are steered towards eachother speeding up the algorithm even more.

Radial RRT

In Radial RRT there are not two, but many trees that are built at the same time[19]. The space that needs to be explored is split up into multiple sections. This subdividing of the space is done in a radial way, for example a 2D space is split up as a pie chart. In each of these seperate sections, a tree is built. Then after a given number of nodes have been added, a global update is done which tries to connects all the trees. This way the computation can be spread out over multiple processors, parallelizing the RRT problem.

Blind RRT

Blind RRT is an extension to the Radial RRT algorithm. Presented in [20], instead of doing the collision checking for every node, it first builds a large tree after which it removes the branches that are colliding with the environment. This way it solves some cases where a specific space is not reachable immediately.

A drawback of each of these parallelizations of RRTs is that they are all specifically designed for kinematic problems. Subdividing the state-space does not work as well, since some regions will be much more difficult to reach then others, if not impossible. Look for example at Figure 2-1, it is impossible to move outwards in the first and third quadrant of the statespace. Therefore parallelizing the kinodynamic planning problem as done in Radial or Blind RRT would not speedup as much as it would for kinemetic planning.
Instead of trying to speed up the entire RRT algorithm, seperate parts of it could be sped up as well. Eventually these could then be combined with parallel RRT to gain an ever larger speed up.

2-1-3 Sampling

Instead of simply taking uniform random samples from the state space, using deterministic sampling could largely speed up the RRT algorithm since it has a large influence on the subsequent steps[21]. The state-of-the-art sampling methods are called Dynamic Sampling Domain and Reachability Guided RRT.

Dynamic Sampling Domain

In [22] a method is presented which tries to speed up RRT by adjusting the domain of which the random samples are drawn. Each node in the tree gets assigned a radius R, which in the beginning is set to $R = \infty$. When a specific node is selected for expansion but it fails due to a collision, this radius is reduced to the distance between the random sample x_r and the selected tree node x_{near}. The same node will in a later iteration only be selected if the new random sample x_r lies within the radius R. By doing so, the *visibile Voronoi region* is approximated by circle with radius R. This sampling method is specifically useful when specific states are difficult to reach, for example when a lot of obstacles are near or with so-called narrow gap problems where a path needs to be found through a narrow gap. The authors claim an increase in speed of several order of magnitude in these difficult cases.

Reachability Guided RRT

Reachability Guided RRT (RG-RRT) was presented in presented in [9], with the goal to tackle the problem of random samples not being reachable. It does so by approximating the reachable set $\mathcal{R}$ for every node in the tree. The reachable set is defined as all states that can be reached from an initial state x_i, by applying inputs from the allowable input set $\mathcal{U}$ within a finite time t_f. The approximation in RG-RRT is done by storing the reached states when applying the minimum and maximum control inputs. The reachable set is then approximated by $\hat{\mathcal{R}}$ which is the triangle created between x_i, $x_{u_{min}}$ and $x_{u_{max}}$. Hence if the randomly taken sample falls within $\hat{\mathcal{R}}$, the point is considered reachable and the algorithm tries to connect it. If not, RG-RRT takes a new random sample. The idea behind RG-RRT is that the sampling and checking wether x_r falls within the approximated reachable set is much cheaper in computation time, compared to trying to connect the node and failing. Hence RG-RRT provides a speedup to the original RRT. A problem however is that the implementationas described earlier does not hold for systems with more than one degree of freedom.

2-1-4 Neighbor Selection

The RRT algorithm is very sensitive to the distance metric used when selecting the neighbor in the tree[15]. Already in the original RRT paper the optimal cost-to-go was defined as the best distance metric. However, since its computation is time consuming, they proposed use a custom metric which is very specific for the underlying system[6]. New algorithms found a way to speed up the computation of the cost-to-go, albeit an approximation.

LQR Based Heuristic

In [23] a distance metric is proposed based on Linear Quadradic regulation. In LQR, a closed loop controller is found by minimizing the quadratic cost function in Equation 2-2. The cost function can be tuned using the weight matrices Q and R.

$$C(x,u) = \int_0^{t_0} (x^T Q x + u^T R u) \mathrm{d}t \tag{2-2}$$

Minimizing this cost function can easily by achieved by solving the quadratic Riccati Equation 2-3 for S, where A_r and B_r are the state-space matrices found by linearizing the system around x_r. The optimal value of the cost-to-go to move from some state x_i to the random state x_r is then found by evaluating Equation 2-4.

$$A_r^T S + S A_r - S B_r R^{-1} B_r^T S + Q = 0 \tag{2-3}$$

$$V^*(x_i, x_r) = (x_i - x_f)^T S (x_i - x_f) \tag{2-4}$$

The nearest node in the tree x_{near} now is found by finding the node that minimizes $V^*(x_i, x_f)$. This technique leads to a distance metric which is an approximation of the optimal cost to go. Eventhough it is not the exact optimal cost to go, it is a much better distance metric then the euclidean distance, leading to a better coverage of the state space. The article does not state anything about the computation time, but it can be assumed that it is faster than computing the exact cost-to-go. They do claim that the advantage of using the LQR based heuristic over the euclidean distance fades as the number of dimensions rises, especially when the system is very non-linear.

Learning LQR

A large speed up was achieved by [7] which applied supervised learning to the neighbor selection. In this article, an approximation of the optimal cost to go that was generated using Iterative Linear Quadratic Regulation (iLQR) was learned by a function approximator called Locally Weighted Projection Regression (LWPR) which is handled in Section 2-3-5.
In iLQR a different cost function is defined as shown in Equation 2-5. Here $C(x, u)$ is the earlier defined cost function, and Q_f is a weighting matrix for the error between the goal state x_f and the reached state x_r.

$$C_{iLQR}(x, u) = C(x, u) + (x_f - x_r)^T Q_f (x_f - x_r) \tag{2-5}$$

This cost function is then minimized iteratively, until the cost function does not change anymore. Doing so leads to a tuple containing of an initial state, a final state and a cost to go: $\{x_i, x_f, c\}$. The relationship between the state pair $\{x_i, x_f\}$ and the cost c is then learned using LWPR. The authors claim a massive decrease of a factor 1000 in computation time.

2-1-5 Steering

One of the most difficult parts of RRT is the steering from the found neighbor x_{near} towards a random state x_r within reasonable time. The simplest way is discretizing the input set $\mathcal{U}$ and trying all possible control inputs. With a large input set $\mathcal{U}$ this of course becomes time consuming.

LQR-RRT

In [24] a method is presented which uses LQR for steering. Following the same steps as in [23], a cost function $C(x, u)$ is minimized by solving for S in the Riccati Equation in Equation 2-3. Using S, an optimal control policy can be defined which steers the system from x_{near} towards x_r.

$$u^* = -R^{-1} B_r^T S (x_r - x_{near}) \tag{2-6}$$

Hence when using the LQR based heuristic as presented in [23], the steering function is practically readily available. In the article results are shown for the swing up of a single pendulum and an acrobot system of which the average computation time equals 4.8 seconds and 109 seconds respectively.

A very recent development managed to compute a swing up for a single pendulum within an average computation time of 2.4 seconds. It does so by approximating both the distance metric and the steering function within a single function approximator. This algorithm is called RRT CoLearn.

2-2 RRT CoLearn

A state of the art attempt to speed up the kinodynamic planning problem, a new algorithm called RRT CoLearn was introduced[14]. The algorithm speeds up the kinodynamic planning problem by replacing both the distance metric and steering function by a function approximator. The function approximator is trained on a dataset of trajectories, which are generated by integrating optimal equations of motion, found using indirect optimal control. The RRT CoLearn algorithm tries to tackle the problem of steering a system from an initial state x_i to a final state x_f by applying supervised learning on a dataset of pregenerated trajectories. A model tries to learn how to connect two states by learning *initial costates*, which are a part of Indirect Optimal Control. The algorithm is currently the state of the art in RRT algorithms which use supervised learning and the starting point for the research in Chapter 3.

2-2-1 Indirect Optimal Control

In optimal control, a control input sequence u is found, such that the system is steered from x_i to x_f within a finite time t_f, by optimizing a cost $J(x, u)$, which is defined as:

$$J(x,u) = \int_0^{t_f} C(x,u)dt \tag{2-7}$$

In the RRT CoLearn algorithm, a method named *indirect optimal control* is used to generate trajectories, which is based on Pontryagin's Maximum Principle[25]. In indirect optimal control, the optimal control sequence is defined by differential equations additional to the equations of motion of the sytem, called the *costate*-equations. Each state x of the regarded system has its corresponding costate λ_x. These differential equations can found by minimizing the control Hamiltonian $\mathcal{H}$. Minimizing $\mathcal{H}$ is necessary condition for optimality in Pontryagin's Principle. Given the cost integrand $C(x, u)$ from Equation 2-7, the system's equations of motion $\dot{x}$ and their corresponding costates λ_x, the control Hamiltonan can be constructed as follows:

$$\mathcal{H}(x, \lambda_x, u) = C(x,u) + \lambda_x^T \dot{x} \tag{2-8}$$

As already mentioned, minimizing $\mathcal{H}$ is a necessary condition for optimality. Hence to find the optimal control input u^* we need to minimize $\mathcal{H}$ with respect to u.

$$u^*(x, \lambda_x) = \arg\min_u \mathcal{H}(x, \lambda_x, u) \tag{2-9}$$

Substituting the control input u with the optimal control input u^* in the control Hamiltonian $\mathcal{H}$ gives us the optimal control Hamiltonan $\mathcal{H}^*$.

$$\mathcal{H}^*(x, \lambda_x) = \mathcal{H}(x, \lambda_x, u^*) \tag{2-10}$$

Using this optimal control hamiltonan, the optimal state and costate trajectories can be derived. Note that u^* is a function of x and λ_x, hence when we have the trajectories we can compute the optimal control input. Since the system now is described in Hamiltonian formalism, the differential equations are found by taking the partial derivates to the states and costates as in Equation 2-11.

$$\begin{aligned}\frac{dx^*}{dt} &= \frac{\partial \mathcal{H}^*}{\partial \lambda_x} \\ \frac{d\lambda_x^*}{dt} &= -\frac{\partial \mathcal{H}^*}{\partial x}\end{aligned} \tag{2-11}$$

This set of ordinary differential equations describe the optimal evolution of the states over time. Say we integrate the differential equations from $t = 0$ to $t = t_f$ with an initial state x_i and initial costate λ_i, then at the time t_f a final state x_f is reached. In order to optimally steer the system from x_i to x_f, $u^*(x, \lambda_x)$ needs to be computed using the found trajectories and applied to the system. Hence, to solve the problem of steering a plant from x_i to x_f we need to find an initial costate λ_i such that $x(t_f) = x_f$.
As later becomes clear, the trajectories that are generated are generated with a free final time t_f. Following Pontryagin's Maximum Principle this imposes a constraint on the initial value of the control Hamiltonian, namely that the control Hamiltonian needs to equal zero at $t = 0$. This implies a constraint on the initial costates, since the control hamiltonian can now be solved for a single costate, such that it is a function of the initial state x_i and the remaining costates. A difficulty with this method of optimal control, and the main reason why different methods which using optimizers to find a control sequence are much more popular, is the fact that the costate equations most of the times are unstable, making the use of this method impractical for larger t_f since integrating the differential equations becomes impossible. For the RRT CoLearn algorithm it is ideal however.

2-2-2 Supervised Learning

The main improvement over normal RRT that RRT CoLearn introduces is the use of supervised learning. Instead of solving the complex TP-BVPs, both the distance metric in the connecting step and the steering function are approximated.

Data generation

The RRT CoLearn uses supervised learning to estimate the cost-to-go to travel between two states, as well as the optimal steering policy. This is done by training a k-Nearest Neighbor approximator on a large dataset of optimal trajectories, and the corresponding cost-to-go to travel over those trajectories. The k-Nearest Neighbor classifier is handled in detail in Section 2-3-1. The trajectories that the approximator is trained on, are generated by integrating the differential equations in 2-11 using for example Runge Kutta integration. Using these trajectories, a dataset is created. A single datapoint consists of an initial state x_i, a final state x_f, the initial costates λ_x and the corresponding cost-to-go J. From a single trajectory, the dataset is structured as follows:

$$\mathcal{D} = \begin{bmatrix} x_0 & x_0 & \lambda_0 & J_0 \\ x_0 & x_1 & \lambda_0 & J_1 \\ x_0 & x_2 & \lambda_0 & J_2 \\ \vdots & \vdots & \vdots & \vdots \\ x_0 & x_{n-1} & \lambda_0 & J_{n-1} \\ x_0 & x_n & \lambda_0 & J_n \end{bmatrix} \tag{2-12}$$

So from this dataset it reads that for example to get from x_0 to x_2, as initial costates λ_0 needs to be selected and the differential equations need to be integrated until the cost-to-go equals J_2.

k-Nearest Neighbors

This dataset will now be used to train a k-NN approximator to create a function approximator which maps the initial and final state x_i and x_f to the initial costates λ_x and cost-to-go J. However, before simply feeding the dataset into the function approximator, the dataset needs to be cleaned. The main reason for this is that multiple trajectory-endpoints can exist which are very close together or even overlap, but while following a different trajectory and thus with different initial costates and cost-to-go. When a 1-NN approximator is used this wouldn't be a problem, but this would also lead to a high variance estimator. Therefore a 3-NN approximator is used and thus these trajectories need to be removed from the dataset, because averaging over different initial costates leads to a completely different trajectory. Cleaning the dataset is done as follows: A random sample is taken from the dataset, together with its nearest neighbor. If the distance between these neighbors is smaller then a boundary value ε, the sample with the highest cost is removed. This is done until k_{max} times no samples have been removed. The cleaning algorithm is depicted in Algorithm 2. The performance

Algorithm 2: RRT CoLearn cleaning

```
$k = 0$
while $k < k_{max}$ do
    $p_r = \texttt{randomSample}(\mathcal{D})$
    $p_n = \texttt{nearestNeighbor}(\mathcal{D}, p_r)$
    if $\|p_r - p_n\| < \varepsilon$ then
        $p_{high} = \arg\max_{p \in \{p_r, p_n\}} \texttt{cost}(p)$
        $\mathcal{D} = \mathcal{D} \setminus p_{high}$
    end
    $k = k + 1$
end
```

of the cleaning algorithm is highly dependant on the boundary value ε. Choosing ε too large would lead to removing too many datapoints, while choosing ε too small still leaves us with a noisy dataset. This parameter has to be selected empirically by testing the approximators model quality. Once the dataset is cleaned the k-NN approximator can be trained and it is ready to be applied in RRT.

2-2-3 RRT

As described in Section 2-1, the RRT algorithm consists of three main parts: *sampling*, *neighbor selection* and *steering*. The sampling part is done the same as in the original RRT, simply uniformly sampling the state space $\mathcal{X}$. The connecting and steering part however are done differently.

Neighbor Selection

Selecting the tree node that will be expanded towards the random sample x_r is done in two steps: first the nodes are selected that are 'near enough'. This is done by computing the euclidian distance to the dataset $\mathcal{D}$ from a query point $x_q = [x_{near}, x_r]$. Here x_{near} is a node in the tree and x_r is the random sample. The distance to the dataset can be seen as a *confidence measure*. If the query point is too far away, the approximator will not able to make a good estimation and thus x_{near} cannot be properly expanded towards x_r. The query points that are within a certain distance d_{max} are stored in a set N. If the set of near enough nodes is empty, a new random sample is taken. Now the only thing left to do is finding the nearest neighbor. This is done selecting the node in the set of close enough nodes which has the lowest cost-to-go. The full algorithm for finding the nearest neighbor is outlined in Algorithm 3.

Algorithm 3: RRT CoLearn: Finding the nearest neighbor

```
$N \leftarrow \{\}$
$\mathcal{T} \leftarrow$ Current Tree
$\mathcal{D} \leftarrow$ Approximator Training Set
while $N = \{\}$ do
    $x_r \leftarrow$ randomSample($\mathcal{X}$)
    foreach $x_t \in \mathcal{T}$ do
        $x_q \leftarrow [x_{near}, x_r]$
        $d \leftarrow$computeDistanceToDataset($\mathcal{D}$,$x_q$)
        if $d < d_{max}$ then
            $N \leftarrow \{N, x_t\}$
        end
    end
end
$x_{near} \leftarrow \arg\min_{x_t \in N}$ cost($x_t, x_r$)
```

Steering

Steering is done fairly simple in RRT CoLearn. The steering input that is needed to steer the system from x_{near} to x_r is found using the k-NN function approximator. The approximator returns an initial costate λ_i and a corresponding cost-to-go J. Using these, the equations of motion are integrate until the cost-to-go equals J. The state that is reached, x_f, will then be added to the tree.

2-2-4 Single Pendulum Example

As an example to demonstrate how the RRT CoLearn algorithm works, we derive the equations for a single pendulum. The single pendulum is modeled as a massless rod of length l, with a mass m attached to the end of it. The other end of the rod is fixed at an axis around which it can rotate and where a torque u can be applied. A graphical representation of the

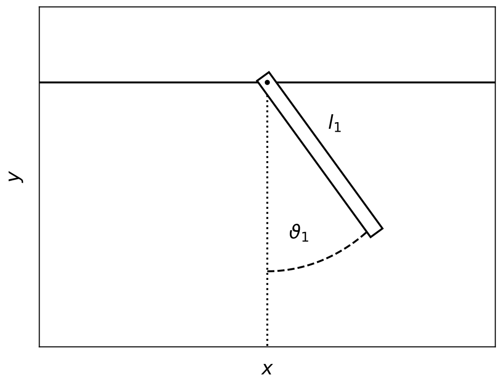


Figure 2-4: Graphical representation of the Single Pendulum model. The rod with length l is fixed at the origin around which it can rotate.

model is shown in Figure 2-4. Using Langrangian mechanics, the equations of motion are derived for the pendulums angle ϑ and its angular velocity ω. For simplicity of computations all constants, m,l and the gravity g are set to 1. As a cost function $C(x, u) = w + \frac{u^2}{2}$ is chosen, where w is some constant which introduces a cost on time. Now the control Hamiltionan can be found as in 2-14.

$$\dot{x} = \begin{bmatrix} \dot{\vartheta} \\ \dot{\omega} \end{bmatrix} = \begin{bmatrix} \omega \\ \frac{1}{ml^2}u - \frac{g}{l}\sin\vartheta \end{bmatrix} = \begin{bmatrix} \omega \\ u - \sin\vartheta \end{bmatrix} \tag{2-13}$$

$$\mathcal{H}(x, \lambda, u) = C(x, u) + \lambda_x^T \dot{x} \tag{2-14}$$

$$\mathcal{H}(x, \lambda, u) = w + \frac{u^2}{2} + \lambda_\vartheta \omega + \lambda_\omega (u - \sin\vartheta) \tag{2-15}$$

The optimal control law u^* can now be found by taking the derivative of H with respect to u and setting it to zero.

$$\frac{\partial \mathcal{H}}{\partial u} = u + \lambda_\omega = 0 \Rightarrow u^* = -\lambda_\omega \tag{2-16}$$

Substituting u with u^* in Equation 2-14 gives us the optimal Hamiltonian $\mathcal{H}^*$ from which we can derive the differential equations describing the optimal state evolutions.

$$\mathcal{H}^*(x, \lambda) = w + \lambda_\vartheta \omega - \lambda_\omega \sin\vartheta - \frac{\lambda_\omega^2}{2} \tag{2-17}$$

The optimal differential equations are now found to be:

$$\begin{aligned} \frac{d\vartheta^*}{dt} &= \frac{\partial \mathcal{H}}{\partial \lambda_\vartheta} = \omega \\ \frac{d\omega^*}{dt} &= \frac{\partial \mathcal{H}}{\partial \lambda_\omega} = -\sin\vartheta - \lambda_\omega \\ \frac{d\lambda_\vartheta^*}{dt} &= -\frac{\partial \mathcal{H}}{\partial \vartheta} = \lambda_\omega \cos\vartheta \\ \frac{d\lambda_\omega^*}{dt} &= -\frac{\partial \mathcal{H}}{\partial \omega} = -\lambda_\vartheta \end{aligned} \tag{2-18}$$

Range of φ	sign(λ_ϑ)	sign(root(λ_ω))
$-\frac{\pi}{2} < \varphi < 0$	$-$	$+$
$0 < \varphi < \frac{\pi}{2}$	$+$	$+$
$\frac{\pi}{2} < \varphi < \pi$	$+$	$-$
$\pi < \varphi < \frac{3\pi}{2}$	$-$	$-$

Table 2-1: The signs of the costates as caused by the parameterization in Equation 2-20. Because of this parameterization, all possible combinations of signs of the costates are used.

Using these ordinary differential equations, a dataset of trajectories can be created that later on can be used for learning. The dataset is created by integrating the equations, using random initial states and costates. Remember that optimal control Hamiltonian has to equal zero at $t = 0$, leading to a constraint on the costates. Because of this constraint, λ_ω can be written in terms of ϑ, ω and λ_ϑ. Note that Equation 2-17 is a quadratic function in terms of λ_ω. Finding the roots of this equations leads to:

$$\lambda_\omega = -\sin\vartheta \pm \sqrt{\sin^2\vartheta + 2w + \lambda_\vartheta \omega} \tag{2-19}$$

Now the initial costates are characterized by only one value, λ_ϑ. However, still one of the two roots needs to be chosen. To solve this, λ_ϑ is parameterized by the parameter $\varphi \in (-\frac{\pi}{2}, \frac{3\pi}{2})$, as shown in Equation 2-20.

$$\begin{aligned} \lambda_\vartheta &= \tan\varphi \\ \lambda_\omega &= -\sin\vartheta + \text{sign}(\cos\varphi)\sqrt{\sin^2\vartheta + 2w + \tan(\varphi)\omega} \end{aligned} \tag{2-20}$$

The reason for this parameterization has to do with the signs of the costates, for every λ_ϑ there exist two values of λ_ω that are valid, namely both roots. The parameterization makes sure that all combinations of signs exist, as is made clear in Table 2-1. Note that when the roots in Equation 2-20 become complex, the initial point is discarded and a new φ is sampled.

Now the dataset $\mathcal{D}$ containing the optimal trajectories can be generated, by first random sampling an initial state $[\vartheta, \omega] \in \mathcal{X}$ and an initial costate parameter $\varphi \in (-\frac{\pi}{2}, -\frac{3\pi}{2})$. Integrating the equations of motion from this initial state creates the trajectories that then are stored in the dataset.
Using the generated dataset, the kNN function approximator is trained with which the RRT algorithm is ran. The goal of the planner is to swing the pendulum up from its downwards equilibrium $[0, 0]$ to its upright equilibrium $[\pi, 0]$. An examplary trajectory that was found using RRT CoLearn is shown in Figure 2-5.

In the article where are RRT CoLearn was presented, the average computation time of finding a swing up trajectory was 2.4 seconds, much faster then other variants. However, this system is currently the only system on which the RRT CoLearn algorithm works. As later turns out in the research, the algorithm *as is* fails on a more complex cart-pole system. In attempts to improve the performance, different supervised learning techniques were investigated.

2-3 Supervised Learning

Supervised learning is a method used to create an approximated function $\hat{f}(\cdot)$, based on a true function $f(\cdot)$. For example the distance metric in the neighbor selection in RRT is often

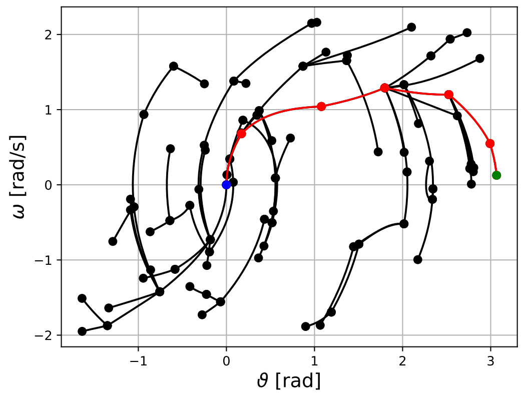


Figure 2-5: Swingup trajectory of a single pendulum found by the RRT CoLearn algorithm.

computed using some function $d(x_i, x_f)$, which is a mapping from an initial and final state pair $\{x_i, x_f\}$ to a some distance value d. In [7] and [8] the optimal cost-to-go is approximated, leading to an approximated function $\hat{d}(x_i, x_f)$. Creating such a function is done using *supervised learning*. First, a large set of examples is created, in case of the approximated distance metric $\hat{d}(x_i, x_f)$ this would be done by computing the true optimal cost-to-go for a lot of different initial and final state pairs $\{x_i, x_f\}$. Doing so can be a time consuming process, but since it is done offline this is not a problem. This large dataset is then shown to the approximation method, which tries to learn its model parameters based on the examples.
This section describes several methods that are already used in RRT such as k-Nearest Neighbors and Locally Weighted Projection Regression, and state of the art methods such as Gaussian Process Regression and Artificial Neural Networks. The state of art methods will be considered in the research for this thesis.

2-3-1 k-Nearest Neighbours

One of the simplest yet a very effective way of supervised learning is k-Nearest Neighbours (kNN)[26]. The mechanics of it are simple: Find the k nearest neighbours in the training set, find the corresponding outputs and then estimate the output by averaging the corresponding outputs in the training set. The full algorithm is outlined in Algorithm 4. Due to this mechanics, any mapping, even highly nonlinear, can be approximated as long as enough datapoints are available. A large dataset does mean a increase in computational time.
The kNN algorithm does have some drawbacks. For a starter its complexity: using the basic algorithm described above the complexity is $\mathcal{O}(N)$, dependant on the size of the training set. For a good approximation N needs to be large, and of course this data has to be stored in memory. The complexity can be reduced by using an efficient data structure, such as kd-trees[27] which reduces it to $\mathcal{O}(\log N)$. A kd-tree partitions the space of the input data, improving the searching speed. In higher dimensions this however loses its effectivity, making the computational complexity approach $\mathcal{O}(N)$. Another problem is the distance metric used[28]. For low input spaces the euclidean distance makes sense, however in higher dimensional spaces this does not have to be. As was already shown in Section 2-1, in state space the euclidean distance does not make sense at all.

Algorithm 4: k-Nearest Neighbors

```
Input: $\mathbf{x}_q$,$X$,$Y$
$\mathbf{y}_q = 0$
for $i = 1$ to $k$ do
    find $\mathbf{x} \in X$ which minimizes $\|\mathbf{x}_q - \mathbf{x}\|^2$
    add corresponding $\mathbf{y} \in Y$ to $\mathbf{y}_q$
    remove $\mathbf{x}$ from $\mathcal{X}$
end
return $\frac{1}{k}\mathbf{y}_q$
```

2-3-2 Locally Weighted Regression

Locally Weighted Regression (LWR) is a method used for function approximation, where a linear model is fitted locally, on a selection of the training data[29]. The total set of training data must be stored in memory, and is used for every single estimation $\hat{y} = \hat{f}(x_q)$. First a diagonal weight matrix $\mathbf{W}$ is computed. The diagonal values w_{ii} are computed using a distance metric D and the values from the dataset x_i.

$$w_{ii} = \exp\left(-\frac{1}{2}(x_i - x_q)^T D(x_i - x_q)\right) \tag{2-21}$$

The distance metric D is the only open parameter, which describes which datapoints of x_i are used in the computation of the regression coefficients. Usually the D matrix is chosen with the variances on the diagonal as in Equation 2-22, where h now is the only free parameter to tune.

$$D = h \cdot diag\left(\frac{1}{\sigma_1^2}, \frac{1}{\sigma_2^2}, \frac{1}{\sigma_3^2}, ..., \frac{1}{\sigma_N^2}\right) \tag{2-22}$$

When enough data is available in the training set, leave-one-out cross validation can be used to optimize the scaling parameter h. Of course this also can be done for the entire D-matrix, using for example gradient-descent methods. When the weight matrix is computed, a linear model is fitted using weighted least squares:

$$\beta = (\mathbf{X}^T\mathbf{W}\mathbf{X})^{-1}\mathbf{X}^T\mathbf{W}\mathbf{y} \tag{2-23}$$

With:

$$\mathbf{X} = \begin{bmatrix}\mathbf{x} & \mathbf{1}\end{bmatrix} \text{ with } \mathbf{x} = \begin{bmatrix}x_1 & x_2 & \cdots & y_N\end{bmatrix}^T \tag{2-24}$$
$$\mathbf{y} = \begin{bmatrix}y_1 & y_2 & \cdots & y_N\end{bmatrix}^T$$

The value $\hat{y}$ can now be estimated using:

$$\hat{y} = \hat{f}(x_q) = \begin{bmatrix}x_q & 1\end{bmatrix} \cdot \beta \tag{2-25}$$

LWR does have some disadvantages, for example the computational complexity is $O(N^2)$, it grows quadratically with the number of training points, since it has to consider the entire training set for every queried datapoint x_q. Another disadvantage of LWR is that the input matrix $\mathbf{X}$ must have full rank. Due to redundant input availability Equation 2-23 could become singular. Furthermore, regular least squares has difficulties with collinearities in the input data. In case of the randomly generated datasets, both redundant data an collinearity could occur. A way to deal with these issues is by using partial least squares.

2-3-3 Locally Weighted Partial Least Squares

Instead of regressing on all data such as LWR does, one can also regress on projections. Locally Weighted Partial Least Squares (LWPLS) recursively projects the training data on a orthogonal basis, after which it does a univariate regression on this projection [30][31]. The total algorithm for LWPLS is listed below is shown in Algorithm 5.

Algorithm 5: LWPLS Algorithm

```
Initialize the data:
    $\bar{x} = \frac{\sum_{n=0}^{N-1} Wx}{\text{trace}(W)}, \bar{y} = \frac{\sum_{n=0}^{N-1} Wy}{\text{trace}(W)}$
 $X_{res} = X - \bar{x}$, $y_{res} = y - \bar{y}$

 for $i = 1, \dots, k$ do
   1. Project the input data on an orthogonal basis:
        $\mathbf{u}_i = \mathbf{X}_{res}^T \mathbf{W}$
        $\mathbf{s}_i = \mathbf{X}_{res} \mathbf{u}_i$
   2. Regress in this direction:
        $\beta_i = \mathbf{s}_i^T \mathbf{W} \mathbf{y}_{res} / (\mathbf{s}_i^T \mathbf{W} \mathbf{s}_i)$
        $\mathbf{p}_i = \mathbf{s}_i^T \mathbf{W} \mathbf{X}_{res} / (\mathbf{s}_i^T \mathbf{W} \mathbf{s}_i)$
   3. Compute the residuals:
        $\mathbf{y}_{res} = \mathbf{y}_{res} + \mathbf{s}_i \beta_i$
        $\mathbf{X}_{res} = \mathbf{X}_{res} + \mathbf{s}_i \mathbf{p}_i^T$
end
Prepare the output computation:
$x_q = x_q - \bar{x}$, $y_q = \bar{y}$
for $i = 1, \dots, k$ do
   1. Project the queried value $x_q$ on projection $i$
        $s = x_q^T u_i$
   2. Add this to the output value $y_q$
        $y_q = y_q + s\beta_i$
   3. Compute the residual of $x_q$
        $x_q = x_q - sp_i$
end
```

Principle Component Regression

Whereas with LWPLS the data was projected on a orthogonal basis, with Principal Component Regression (PCR) the data is projected on its principal components instead of an orthogonal basis. Hence step 1 in the first for-loop in algorithm 5 becomes:

$$u_i = \max(\text{eigv}(\mathbf{X}_{res}^T \mathbf{W} \mathbf{X}_{res})) \tag{2-26}$$

Choosing these projections will maximize the confidence in the univariate regression. It however is known to choose bad projections and cannont ignore irrelevant dimensions[31].

2-3-4 Receptive Field Weighted Regression

Presented in [30], the Receptive Field Weighted Regression (RFWR) predicts the function output $\hat{y} = \hat{f}(x)$ by taking a normalized weighted sum of the outputs of multiple local linear models, as shown in equation 2-27.

$$\hat{\mathbf{y}} = \frac{\sum\limits_{i=1}^{n} w_i \hat{\mathbf{y}}_i}{\sum\limits_{i=1}^{n} w_i} \tag{2-27}$$

$$\hat{\mathbf{y}}_i = \beta_i^T \tilde{\mathbf{x}} \text{ where } \tilde{\mathbf{x}} = \begin{bmatrix}(\mathbf{x}-\mathbf{c}_i)^T & 1\end{bmatrix}^T \tag{2-28}$$

Here w_i is an activation strength computed by an activation centered around c_i. The activation function is characterized by a kernel function, usually the gaussian distribution shown in equation 2-29. The distance metric D_i is constructed from an upper triangular matrix M_i, such that it renders D_i positive definite.

$$w_i = \exp\left(-\frac{1}{2}(\mathbf{x}-\mathbf{c}_i)^T \mathbf{D}_i (\mathbf{x}-\mathbf{c}_i)\right), \text{ with } \mathbf{D}_i = \mathbf{M}_i^T \mathbf{M}_i \tag{2-29}$$

Computation of β

β, the parameter vector for the local fit is computed using a weighted least squares.

$$\beta = (\mathbf{X}^T \mathbf{W} \mathbf{X})^{-1} \mathbf{X}^T \mathbf{W} \mathbf{Y} = \mathbf{P} \mathbf{X}^T \mathbf{W} \mathbf{Y} \tag{2-30}$$

The update rule for β when a new datapoint $\tilde{\mathbf{x}}$ has arrived is shown in equation 2-31. A forgetting factor λ is incorporated, to cancel out data from when the weights w were not properly learned yet.

$$\begin{aligned}
\mathbf{e}_{cv} &= \mathbf{y} - \beta_n^T \tilde{\mathbf{x}} \\
\mathbf{P}^{n+1} &= \frac{1}{\lambda}\left(\mathbf{P}^n - \frac{\mathbf{P}^n \tilde{\mathbf{x}} \tilde{\mathbf{x}}^T \mathbf{P}^n}{\frac{\lambda}{w} + \tilde{\mathbf{x}}^T \mathbf{P}^n \tilde{\mathbf{x}}}\right) \\
\beta^{n+1} &= \beta^n + w \mathbf{P}^{n+1} \tilde{\mathbf{x}} \mathbf{e}_{cv}^T
\end{aligned} \tag{2-31}$$

Now the total algorithm is as follows: The only parameter left to tune now is the distance metric $\mathbf{D}$. This can be done by hand, or as proposed in [30] minimizing a cost function J using gradient descent. The used cost function is shown in 2-32. In general it consists of two parts: A weighted cost of the mean squared error of the model, as if it was computed using leave-on-out cross validation, and a cost for the number of receptive fields used, weighed by a factor γ.

$$J = \frac{1}{W} \sum_{i=1}^{p} \frac{w_i \|\mathbf{y}_i - \hat{\mathbf{y}}_i\|^2}{(1 - w_i \tilde{\mathbf{x}}_i^T \mathbf{P} \tilde{\mathbf{x}}_i)^2} + \gamma \sum_{i,j=1}^{p} D_{ij}^2 \tag{2-32}$$

One of the biggest advantages of RFWR is that it is truely incremental, it does not need to store the training data for its regression. Some issues however arise: The computational complexity, rises cubic in higher dimensions,i.e. $\mathcal{O}(d^2)$ [29][32]. Hence it is not suitable for high

Algorithm 6: RFWR Update Algorithm

```
Input: New datapoint $(\mathbf{x}, y)$
for All models do
    Compute activation according to 2-29.
    Update local model according to 2-31.
end
if $\boldsymbol{w} < w_{gen} \forall w_i$ then
    Create new model:
    $c_i = x$
    $\beta_i = \begin{bmatrix} \mathbf{0} & y \end{bmatrix}^T$
end
```

dimensional systems. Another problem is that, eventhough the distance metric $\mathbf{D}$ is updated, still an initial value has to be chosen. This initial value highly dictates the performance of the algorithm since it is highly data dependant. Another problem that can occur which is highly data-dependant, is the number of needed underlying linear models. When the data used for training is highly non-linear, a lot of local models will be needed to achieve good performance since the model assumes local linearity[33][34].

2-3-5 Locally Weighted Projection Regression

Created in 2000 by S. Schaal and S. Vijayakumar, Locally Weighted Projection Regression (LWPR) is a variant on RFWR, where instead of regressing on all available data, it projects the data on an orthogonal basis, on which the regression is performed recursively (much like LWPLS)[31]. LWPR is currently considered as one of the most state of the art function approximators. Since LWPR uses projections instead of the entire dataset, it has a complexity of $\mathcal{O}(d)$, linear in the number of dimensions[32][33]. This makes it one of, if not the fastest algorithm around. Choosing the initial distance metric $\mathbf{D}$ however still remains problematic, since it still is highly dependant on the underlying data and thus to apply this method still a lot of tuning is needed beforehand.

2-3-6 Gaussian Process Regression

Gaussian Process Regression (GPR) is a different kind of approach than the regressions already mentioned. It assumes the function to be approximated is a function with additive zero-mean gaussian noise with variance σ_n^2, i.e. $\mathbf{y} = f(\mathbf{x}) + \epsilon$[35]. Hence the output space $\mathcal{Y}$ can be modelled as a Gaussian process:

$$\mathcal{Y} \sim \mathcal{N}(0, \mathbf{K}(\mathcal{X}, \mathcal{X}) + \sigma_n^2 \mathbf{I}) \tag{2-33}$$

Here $\mathbf{K}(\mathcal{X}, \mathcal{X})$ is a covariance matrix, computed using some covariance function $k(\mathbf{x}_1, \mathbf{x}_2)$, which most often are selected to be gaussian kernels [29][35][36]:

$$\mathbf{k}(\mathbf{x}_1, \mathbf{x}_2) = \sigma^2 \exp\left(-\frac{1}{2}(\mathbf{x}_1 - \mathbf{x}_2)^T \mathbf{W} (\mathbf{x}_1 - \mathbf{x}_2)\right) \tag{2-34}$$

Other covariance functions could of course also be used. Inserting a query point $\mathbf{x}_q$ now leads to a joint distribution of $\mathcal{Y}$ with $\mathbf{y}_q = \hat{f}(\mathbf{x}_q)$:

$$\begin{bmatrix} \mathcal{Y} \\ \mathbf{y}_q \end{bmatrix} \sim \mathcal{N}\left(0, \begin{bmatrix} \mathbf{K}(\mathcal{X},\mathcal{X}) + \sigma_n^2\mathbf{I} & \mathbf{k}(\mathcal{X},x_q) \\ \mathbf{k}(\mathcal{X},x_q) & \mathbf{k}(x_q,x_q) \end{bmatrix}\right) \tag{2-35}$$

Conditioning the distribution in Equation 2-35 leads to a new distribution with all available data $\mathbf{p}(\mathbf{y}_q|\mathbf{x}_q,\mathcal{X},\mathcal{Y})$, of which thus the value $\mathbf{y}_p$ can be computed as the mean of the distribution:

$$\mathbf{y}_q = \mathbf{E}[\mathbf{p}(\mathbf{y}_q|\mathbf{x}_q,\mathcal{X},\mathcal{Y})] = \mathbf{k}(\mathbf{x}_q,\mathcal{X})(\mathbf{K}(\mathcal{X},\mathcal{X}) + \sigma_n^2\mathbf{I})^{-1}\mathcal{Y} \tag{2-36}$$

An exact derivation can be found in [35]. Eventhough the algorithm is slower than LWPR, it has a complexity of $\mathcal{O}(d^3)$, its predictions are often more accurate. It still has some initial metaparameters to set (σ^2,σ_n^2 and $\mathbf{W}$), however these could be optimized by maximizing the log marginal likelihood [35].

In [34] GPR is applied to learn the model of a 7-DOF manipulator, to use it in CTC control. The model found was actually more accurate than an analytical model found using rigid body dynamics. Due to its complexity however it was too slow to use as an online feedback law. For motion planning purpuses however it could be fast enough.

2-3-7 Artificial Neural Networks

A different type of approximator than the above mentioned ones is called Artificial Neural Networks (ANNs). ANNs themselves have been around for a while, but fastly grew in popularity together with the rise of computing power in the 1990s. A big advantage of neural networks is the simplicity in use, for basic applications only a few parameters need to be set. In case of function approximation, the most used network type is a multilayer perceptron network.

Perceptron

In a mulilayer perceptron network, a lot of *perceptrons* are linked together to create a network[37]. These perceptrons themselves are very simple building blocks, consisting of a weight vector w, a bias b and some activation function $\phi(x)$[38]. Using these variables, the output of the perceptron is computed as in Equation 2-37. Hence, the perceptron does nothing more than take an affine combination of the inputs and than compute an activation using some, usually non-linear, function.

$$y(x) = \phi(w^T x + b) \tag{2-37}$$

This function is also one of the most important parameters of the perceptron which can be selected. Most of the times this function is chosen to be a non-linear function such as the sigmoidal function in Equation 2-38 or the hyperbolic tangent function. In 2000 the rectified linear unit (ReLU), Equation 2-39, was presented based on how actual neurons in brains are actived[39]. Since then the ReLU function has been the most widely used activation function in neural networks. The main advantage of the ReLU unit are its better learning properties.

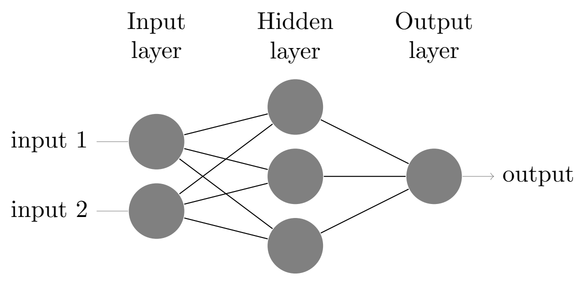


Figure 2-6: Schematic representation of an Artifical Neural Network

In case of for example the sigmoidal function, when the error is very large, the derivative of it is very small leading to a slow learning. The ReLU function does not have this problem.

$$\phi(x) = \frac{1}{1+e^{-x}} \tag{2-38}$$

$$\phi(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{2-39}$$

Another type of activation funtion that has been successfully used in approximating functions, are wavelet functions. A wavelet function is a function that starts and ends at 0 but oscillates in between. In [40] some highly non-linear and even discontinuous functions have been approximated very accurately using Gaussian wavelet functions.

Multilayer Perceptron Network

A multilayer perceptron network is created by combining a lot of these perceptrons, as shown in Figure 2-6. In this figure each circle represents a perceptron with its own weight vector w and bias b, and the connections mean that the output of the perceptron on the left is used as input by the perceptron on the right. Another thing that can be seen from this figure is that there are multiple layers containing perceptrons. There are three types of layers: *Input Layers*, *Hidden Layers* and *Output Layers*. The number of perceptrons in the Input layer is always equal to the number of inputs the mapping has that is being approximated, and the same holds for the Output Layer. The number of layers in the hidden layer however is subject to change. Choosing a lot of perceptrons in a hidden layer leads to a *wide* neural network, whereas adding multiple hidden layers behind eachother leads to a *deep* neural network. No exact solution exists for the optimal number of layers and perceptrons to choose, but in [41] it is proven that any two-layer network can learn a dataset of N samples with m outputs, using $\sqrt{(m+1)N}$ perceptrons per layer, however usually much less are needed. In [42] a method is presented which optimizes the network topology using a genetic algorithm.

Training

Now that a network is constructed, the total output of the network can be computed and the network can be trained. The parameters that are adapted in the training are thus the weight vectors w_i and the biases b_i of every node. Before the training can be done, an error function needs to be defined. In the field of neural networks, this error function is often called the *loss*-function. Many options are available for the loss function, but the one that is used most in function approximation is called the L_2-Norm given in equation 2-40.

$$L_2 = \sum_i \|y_i - \hat{y}_i\|^2 \tag{2-40}$$

Initially the training was often done using the Gradient Descent algorithm[43], where the weights and biases are updated by the partial derivatives of the error L with respect to the weights and biases as in Equation 2-41. Nowadays however more complex optimization algorithms are used, which converge must faster and with better results. The most widely used one is called Adam[44].

$$w_{i+1} = w_i - \alpha \frac{\partial L}{\partial w_i} \tag{2-41}$$

Classification

However, neural networks are not only used for function approximation. Actually most applications nowadays are classification problems. In classification, the input vector is called a vector of *features*, and certain combinations of these input features will then be classified into a certain class. To achieve this, the output layer is changed to have as many neurons as classes, and the activation of the output layer is changed such that it outputs the probability that it is the given class. Candidates for the output function could be a simple *max* function, which outputs a 1 at the most probable class, or for example the *softmax* function in Equation 2-42 which computes the probability that a set of features is in class j, where N is the number of classes.

$$\phi(x)_j = \frac{e^{x_j}}{\sum_{i=1}^{N} e^{x_i}} \tag{2-42}$$

Training a classification network can be done using the L_2 norm, however this often does not lead to optimal performance[45]. A better loss function for this problem is called the *log-loss* function which is .

$$L_{ce} = -\frac{1}{n}\sum_x y \ln \hat{y} + (1-y)\ln(1-\hat{y}) \tag{2-43}$$

Using the L_2-Norm as loss function has the problem that when the network is far off, learning goes really slow. Only when the estimated output $\hat{y}$ closes in on y the learning speed improves. When the cross entropy is used as loss function this problem is much smaller.

Recurrent and Convolutive Networks

Two types of neural networks that are subject to a lot of research nowadays, and which are applied broadly are Recurrent and Convolutive networks. In recurrent networks, the output of the neurons in the output layer are fed back into the neurons in the hidden layers[46]. This way a directed cycle is created, which makes the network particularly profound in detection of sequences, since the output at some time is a function of the previous outputs as well. Recurrent networks are therefore used a lot in recognition of speech and writing[47]. Convolutive networks are actually multiple networks on top of each other, that are interconnected.

Method	Complexity	Dimenionality	Non-linearity
kNN	$\mathcal{O}(N)^1$	-	++
LWPR	$\mathcal{O}(d)$	++	-
GPR	$\mathcal{O}(d^3)$	+	+
ANN	$\mathcal{O}(p^3)$	++	+

Table 2-2: Comparison of the most promising function approximators. [1]The complexity of kNN using kd-trees is $\mathcal{O}(\log N)$, but reduces to $\mathcal{O}(N)$ in higher dimensional spaces. Therefore $\mathcal{O}(N)$ is used in the comparison.

Because of this instead of a vector as input, these networks take a matrix as input[48], which is very useful when applying classification on images or videos. Since the main applications of Recurrent and Convolutive Networks are not in the scope of this thesis, they will not be handled more indepth.

A drawback of using artificial neural networks is called catastrophic interference[49].Interference happens when a trained model is offered new data to learn. Learning this new data causes it to forget the previously learned data, especially when the new training data and the previously used training data do not overlap. A nice example of this is given in [30]. Because of the interference neural networks can not be used as an incremental model such as the LWPR algorithm, and they must be trained only once. Neural networks do have the property of being universal approximators[50][51]. This means that any non-linear mapping can be approximated by a neural network, as long as enough neurons are used. In practice this of course could lead to very large networks making them impractical to use, since the complexity of a neural network is approximately $\mathcal{O}(p^3)$ where p is the number of neurons[52].

2-3-8 Overview

In Table 2-2 an overview is given of the most promising function approximators handled in this section. Note that in the complexity column, N stands for the number of datapoints, d the number of dimensions and p is the number of neurons. Some algorithms, such as LWR or RFWR are not considered, since very much improved versions of them are available. The algorithms are scored on computational complexity, dimensionality and the ability to handle non-linearity. Based on these qualities a selection is made which will be used later on in this thesis. According to the literature, LWPR should be the state of the art in function approximation with by far the lowest complexity, it is however very sensitive to tuning its initial hyperparameters. On top of that, when the non-linearity of the mapping rises and more local models are needed the complexity rises as well. Neural networks also have a low complexity, depending on how wide and deep they are made. An advantage of kNN is its ability of handling highly non-linear data, at the cost of computational time. GPR performs less on dimensionality but is supposed to create more accurate models. Since computational time is the main problem with motion planning, ANN and LWPR will be considered. kNN is considered as well since it is used in the RRT CoLearn algorithm.

Chapter 3

RRT CoLearn on Multi-DOF Systems

In order to answer the first research question '*What are the consequences for RRT CoLearn when it is applied to a multi-DOF system?*', the algorithm is applied on a Cart Pole system and its results are compared to the same experiment on a single pendulum swing up. After this an analysis is performed on the steering function and data generation. Improvements have been implemented on both steps, which after analysis showed to not benefit the algorithm as was expected. Furthermore an investigation is done in applying different function approximators than K-Nearest Neighbor (kNN) in order to speed up the algorithm.

3-1 Baseline Analysis of RRT CoLearn

To make a fair comparison between results, a baseline analysis is done of the RRT CoLearn algorithm. The reason for doing this and not comparing the new results with those in the original publication[14], is to rule out performance differences caused by the hardware it is run on or for example implementation differences. All software is written in Python and run on a Hewlet-Packard 8560w workstation with an Intel i7-2670QM processor and 16 Gb of memory. For the kNN function approximator the SciKit implementation is used from the SciKit-Learn package[53].

3-1-1 Experiment

Algorithm 7 outlines the framework that is used for all experiments performed in this chapter. First a dataset (D) is created containing the optimal trajectories and their corresponding initial costate parameterization. In case of the single pendulum system from Section 2-2-4 this means the optimal equations of motion are integrated for 3000 different initial states and costates for 2 seconds with a timestep of $\Delta t = 0.01$s. This leads to $600k$ datapoints of the form $\{x_i, x_f, \varphi, c\}$, which can be read as '*To get from x_i to x_f compute the initial costates using φ and integrate until the cost equals c*'. Hence each datapoint contains a trajectory. For more

detailed information refer to Section 2-2. This dataset is then cleaned using `cleanData(`$\mathcal{D}$`)` after which it is used to create a model $f(\cdot)$ which approximates both the distance metric D and the steering input parameterization φ. This model is passed to the `doRRT(`$f(\cdot)$`)` routine which is outlined in Algorithm 8. This routine implements the RRT algorithm with approximated distance metric and steering function. Finally, all steps are nested in loops such that they can be executed multiple times. The variable *NUM_RUNS* states the number of times a new dataset $\mathcal{D}$ is created on which a new model is trained. The variable *NUM_TRAJS* states the number of planning attempts that are done using these models. Hence, a full run leads to *NUM_RUNS*×*NUM_TRAJS* trajectories. The RRT framework and the separate functions outlined by Algorithm 7 and 8 will be the used framework throughout this thesis. This way the basic RRT functions such as sampling and keeping track of the tree are all kept equal. When functions are tested separately, such as for example the steering function `steer(`x_{near}`,`x_r`,`φ`)`, these functions will be implemented such that they fit straight into this RRT framework.
For RRT CoLearn, the model $f(\cdot)$ computes the cost-to-go between two states x_i and x_f and the input parameterization is based on the initial costates λ_i. *NUM_RUNS* is set to 10 and *NUM_TRAJS* is set to 100, leading to a total of 1000 trajectories. The first system that is tested is the single pendulum system of section 2-2-4. The `doRRT(·)` function is timed, such that eventually the average computation time can be computed.

Algorithm 7: RRT Baseline Experiment

```
1 for $k \leftarrow 1$ to $NUM\_RUNS$ do
2     $\mathcal{D} \leftarrow$ generateData()
3     $\mathcal{D} \leftarrow$ cleanData($\mathcal{D}$)
4     $f(x_i, x_f) \leftarrow$ createModel($\mathcal{D}$)
5     for $l \leftarrow 1$ to $NUM\_TRAJS$ do
6         doRRT($f(\cdot)$)
7     end
8 end
```

Algorithm 8: `doRRT(`x_i`,`x_f`,`$f(x_i, x_f)$`)`

```
1 $\mathcal{T} \leftarrow \{x_i\}$
2 while $\mathcal{T} \cap \mathcal{X}_{goal} = \emptyset$ do
3     $x_r \leftarrow$ sample($\mathcal{X}_{free}$)
4     $D \leftarrow$ approximateDistanceMetric($\mathcal{T}$,$x_r$,$f(\cdot)$)
5     $x_{near}, c \leftarrow$ findNeighbor($\mathcal{T}$,$D$)
6     $\varphi \leftarrow$ approximateSteeringInput($x_{near}$,$x_r$,$f(\cdot)$)
7     $x_{new} \leftarrow$ steer($x_{near}$,$x_r$,$\varphi$,$c$)
8     $\mathcal{T} \leftarrow \mathcal{T} \cup x_{new}$
9 end
```

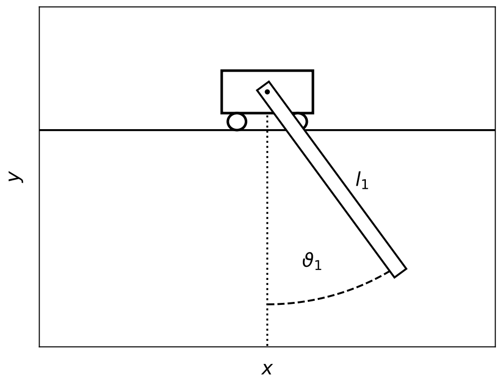


Figure 3-1: Graphical representation of the cart pole system. The cart with mass m_1 can move along the x-axis. The rod with mass m_2 is connected to the cart and can rotate around its end. Both masses m_1 and m_2 are modelled as point masses.

3-1-2 Results

The baseline experiment for the RRT CoLearn algorithm on a single pendulum swingup resulted in an average computation time of 3.323 ± 5.471 seconds. Note that this is about 1.5 times the computation time from the original paper[14], indicating the difference in hardware used since the implementation is exactly equal.

3-1-3 Cart Pole System

The RRT CoLearn algorithm works very fast for the single pendulum system. However, it has not yet been tested on any other system. The next step now is to repeat the experiment outlined in Algorithm 7, but now applied on a cart pole system. The cart pole system is a highly non-linear system with 2 degrees of freedom. It consists of two moving parts: A cart with mass m_1 which can move frictionless in the direction of x on the line $y = 0$, and a weightless rod of length l with a point mass m_2 attached to it at one end. The other end of the rod is connected to the cart, such that it can rotate around this axis. A graphical representation of the system is shown in figure 3-1. The state space of the cart is defined by the position and velocity of the cart, x and v, and the angle and rotational velocity of the rod ϑ and ω. The equations of motion are derived using Lagrangian mechanics. The Lagrangian is found by the difference between the kinetic energy and the potential energy, which is shown in Equation 3-1.

$$\begin{aligned} \mathcal{L} &= T - V \\ T &= \frac{1}{2}(m_1 + m_2)v^2 + m_2 v\omega\cos\vartheta + \frac{1}{2}m_2 l^2\omega^2 \\ V &= -m_2 g l\cos\vartheta \end{aligned} \tag{3-1}$$

Using the Lagrangian $\mathcal{L}$ the equations of motion can be derived by applying Equation 3-2 with q substituted for ϑ and x. Doing so leads to the equations of motion given in Equation

3-3.

$$\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial \mathcal{L}}{\partial \dot{q}} - \frac{\partial \mathcal{L}}{\partial q} = u \tag{3-2}$$

$$\begin{aligned}
\dot{x} &= v \\
\dot{\vartheta} &= \omega \\
\dot{v} &= \frac{m_2 l \omega^2 \sin\vartheta + u + m_2 g \cos\vartheta \sin\vartheta}{m_1 + m_2 - m_2 \cos^2\theta} \\
\dot{\omega} &= -\frac{m_2 l \omega^2 \cos\vartheta \sin\vartheta + (m_1 + m_2) g \sin\vartheta + u \cos\vartheta}{l(m_1 + m_2 - m_2 \cos^2\vartheta)}
\end{aligned} \tag{3-3}$$

Using these equations of motion the differential equations describing the optimal state evolution are derived as shown in Section 2-2-4, and the same cost function $C(x, u) = w + \frac{u^2}{2}$ is used. The expectation was that the algorithm would converge within reasonable time, however as opposed to the single pendulum system, the algorithm did not converge at all. Many different settings were tried, for example shorter trajectories with higher resolution, since the equations of motion are unstable this might have improved the performance. Multiple resolutions were tried ranging between $\Delta t = 0.001$s and $\Delta t = 0.1$s. The function approximator had been changed to a 1-NN classifier to remove the influence of cleaning. This way only the trajectories were to be stitched together in order to build the tree. Even huge datasets were made, up to 600 million datapoints, to rule out that the problem was the curse of dimensionality. None of these adjustments led to convergence.
The remainder of this chapter is an investigation into other possible reasons for the algorithm not converging, and how to address these problems.

3-2 Steering Function

The approximation of the cost was found to be within reasonable bounds, as shown in Table 3-2, and multiple publications have also shown that the distance metric can successfully be approximated[7][8]. Therefore it was expected that the cause of the algorithm not converging lies with the approximation of the steering input parameterization. This section will investigate the approximation method used and the performance of the approximated steering function seperately. First the steering function is investigated as it is applied on the Single Pendulum, since this system is shown to work. The same is done for the cart pole system and the differences will be discussed.

3-2-1 Analysis of Steering Function

In order to find out if bad performance of the steering function is the cause of the algorithm not converging, an experiment is set up as follows: Two datasets are created using the `generateData()` function, a training set $\mathcal{D}_{train}$ and a test set $\mathcal{D}_{test}$. Using the training set a model $f(x_i, x_f)$ is created which will then be applied on the test set. This test set contains initial and final state pairs $\{x_i, x_f\}$ and the corresponding true cost-to-go and input parameterization, $\{c, \varphi\}$. For each point in the dataset their approximated values $\hat{c}$ and $\hat{\varphi}$

are computed, which are then used to steer the system from x_i towards x_f. For each of these three values the Mean Squared Error (MSE) shown in Equation 3-4 is computed. The expirement is outlined in Algorithm 9. By doing this experiment the quality of the function approximator is tested, as well as its application in the steering function. Note that the way the functions in lines $2 - 8$ are implemented are exactly the same way as in Algorithm 7.
The experiment is done for the single pendulum system as well as the cart pole system. The training set contains 3000 random trajectories, generated by integrating the equations of motion for 2 seconds using Runge Kutta integration with a Δt of $0.01s$. The results are shown in Table 3-1.

$$MSE(x, y) = \frac{1}{N} \sum_{i=0}^{N-1} (x_i - y_i)^2 \tag{3-4}$$

Algorithm 9: Analysis of Steering Function

```
1  for $k \leftarrow 1$ to $NUM\_RUNS$ do
2      $\mathcal{D}_{train} \leftarrow$ generateData()
3      $\mathcal{D}_{test} \leftarrow$ generateData()
4      $f(x_i, x_f) \leftarrow$ createModel($\mathcal{D}_{train}$)
5      foreach $\{x_i, x_f, c, \varphi\} \in \mathcal{D}_{test}$ do
6          $\hat{c} \leftarrow$ approximateDistanceMetric($x_i$,$x_f$,$f(\cdot)$)
7          $\hat{\varphi} \leftarrow$ approximateSteeringInput($x_i$,$x_f$,$f(\cdot)$)
8          $\hat{x}_f \leftarrow$ steer($x_i$,$x_f$,$\varphi$,$c$)
9          $e_c \leftarrow \{e_c$, MSE($c$,$\hat{c}$)$\}$
10         $e_\varphi \leftarrow \{e_\varphi$, MSE($\varphi$,$\hat{\varphi}$)$\}$
11         $e_x \leftarrow \{e_x$, MSE($x_f$,$\hat{x}_f$)$\}$
12     end
13 end
```

An interesting result is that the quality of the separate predictions is not very good. Considering that the range of φ_i is equal to $(-\frac{\pi}{2}, \frac{3\pi}{2})$, then an average estimation error of 3.928 is substantial. What is even more remarkable is that even though the approximation errors of the model are high, applying the steering function still results in a very samll error of 0.020. These results are further confirmed by Figure 3-2. In this figure, each estimated value is plotted against its true value. In case of a perfect estimation with zero error, this figure would show a straight line through the origin, which is shown in red in each figure. The closer the points are plotted near the red line, the higher the correlation is between the true and esti-

	Single Pendulum	Cart Pole
$MSE(\varphi, \hat{\varphi})$	3.928 ± 0.271	60.037 ± 2.030
$MSE(c, \hat{c})$	0.079 ± 0.010	0.625 ± 0.052
$MSE(x_f, x_r)$	0.020 ± 0.001	0.148 ± 0.012
t_{approx}	1.130 ± 0.114	1.258 ± 0.098

Table 3-1: Results for the RRT CoLearn steering quality experiment, outlined in Algorithm 9. The steering error $MSE(x_f, x_r)$ is higher for the Cart Pole system, but note that the dataset the approximtor is trained on is of equal size. Increasing the dataset size should lower the error.

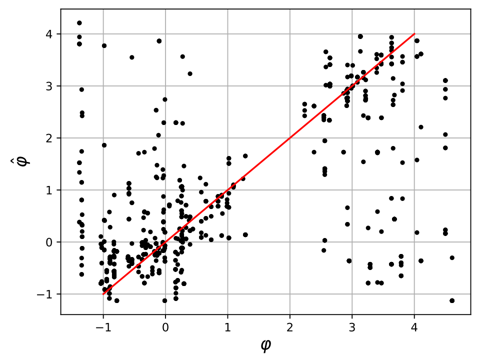
mated values. Figure 3-2a shows the plot for the input parameterization φ. As expected, the spread is large, leading to the high estimation error in Table 3-1. The spread of the approximation of the cost to go is less, as shown in Figure 3-2b. However, when looking at Figure 3-2c and 3-2d the spread around the red line is minimal, meaning that the steering function does work well even though the approximations are off. Figure 3-3 shows similar results for the cart pole system. The estimation of the input parameterization, shown in Figure 3-3a, appears to be completely random. The estimation of the cost in Figure 3-3b shows similar behavior. However, Figure 3-3c through 3-3f show better results for the steering function.
The fact that even though the approximations of the variables are very off, the steering function still works has two reasons: The first reason lies in the way the data that the model is trained on is generated. The trajectories are generated using indirect optimal control, and thus the dataset contains optimal trajectories. However, these trajectories are not guaranteed to be globally optimal. Cleaning of the dataset removes sub-optimal trajectories, making the remaining trajectories the optimal trajectories of the dataset. When the test contains a trajectory with similar $\{x_i, x_f\}$ to a trajectory of the training set, it is most likely to be from a different optimum. In that case, the cost to go and the input parameterization are different but the end point of the trajectory remains the same. Second, the x_f and $\hat{x}_f$ are both end-states from a trajectory starting at the same initial state x_i. Since the trajectories in both the training and test set are short, the reached state $\hat{x}_f$ is bound to lie in the vicinity of x_i and thus also of x_f.
Taking this in account it is clear that a good steering function based on short trajectories would lead to a a very small steering error $MSE(x_f, \hat{x}_f)$. It does so for the single pendulum, however it does not for the cart pole system with a MSE of 0.148. A possible reason for this is the number of trajectories that are used for training the kNN model.

3-2-2 Dataset Size

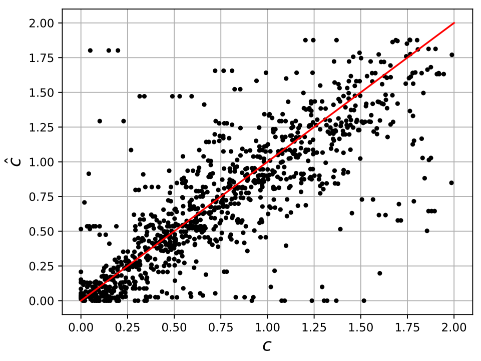
Recall that the function approximator used is a kNN approximator, which finds the most similar entries in a dataset and uses those to compute and estimate. A larger dataset would increase the probability of the dataset containing a similar $\{x_i, x_f\}$ state pair which would result in a lower steering error $MSE(x_f, \hat{x}_f)$. Since the space that is sampled from is doubled in number of dimensions, more trajectories are needed in order to properly cover the state space. Therefore the same experiment is repeated, but now with the kNN model trained on a larger dataset. The results for different dataset sizes are shown in Table 3-2.
As expected, the steering error decreases when larger training sets are used. With a dataset compiled of 100000 trajectories, the steering error $MSE(x_f, \hat{x}_f)$ is reduced to 0.076, which is still larger than for the single pendulum but half of what it was. However, repeating the experiment of Section 3-1 but now with the kNN model trained on 100000 trajectories did not lead to any improvements, the RRT CoLearn algorithm still did not converge. Therefore the problem is to be sought elsewhere. Since the algorithm does not converge, even when a large dataset is used, the cause might lie in the quality of the data itself.

3-2-3 Dataset Coverage

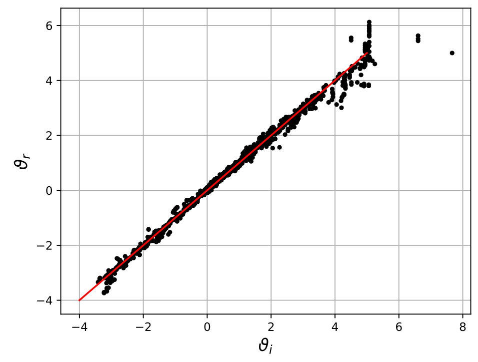
The trajectories in the dataset that is used for training are generated using indirect optimal control. The initial costates play an important role, since they define the trajectory that is



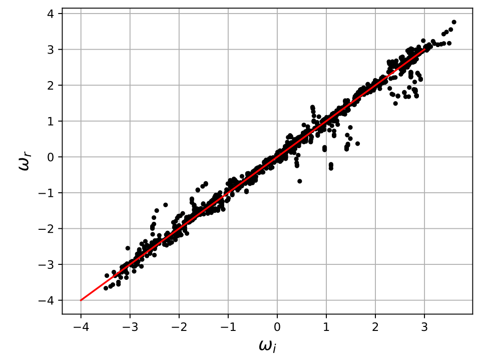
(a)



(b)



(c)



(d)

Figure 3-2: (a) and (b) show the estimated values of ϕ and c plotted against their true values. (c) and (d) show the goal states ϑ and ω plotted against their reached counterparts. The red line depicts perfect estimation. the estimations in (a) and (b) are really off, still a small steering error is achieved as is visibile in (c) and (d).

Trajectories	$MSE(\varphi, \hat{\varphi})$	$MSE(c, \hat{c})$	$MSE(x_f, \hat{x}_r)$
3000	60.037 ± 2.030	0.625 ± 0.052	0.148 ± 0.012
10000	58.988 ± 2.971	0.511 ± 0.035	0.116 ± 0.015
50000	56.064 ± 1.219	0.357 ± 0.025	0.089 ± 0.009
100000	56.522 ± 1.089	0.110 ± 0.030	0.076 ± 0.012

Table 3-2: Results for the steering function analysis when using different dataset sizes on the cart pole system. As was expected, a larger dataset decreases the steering error.

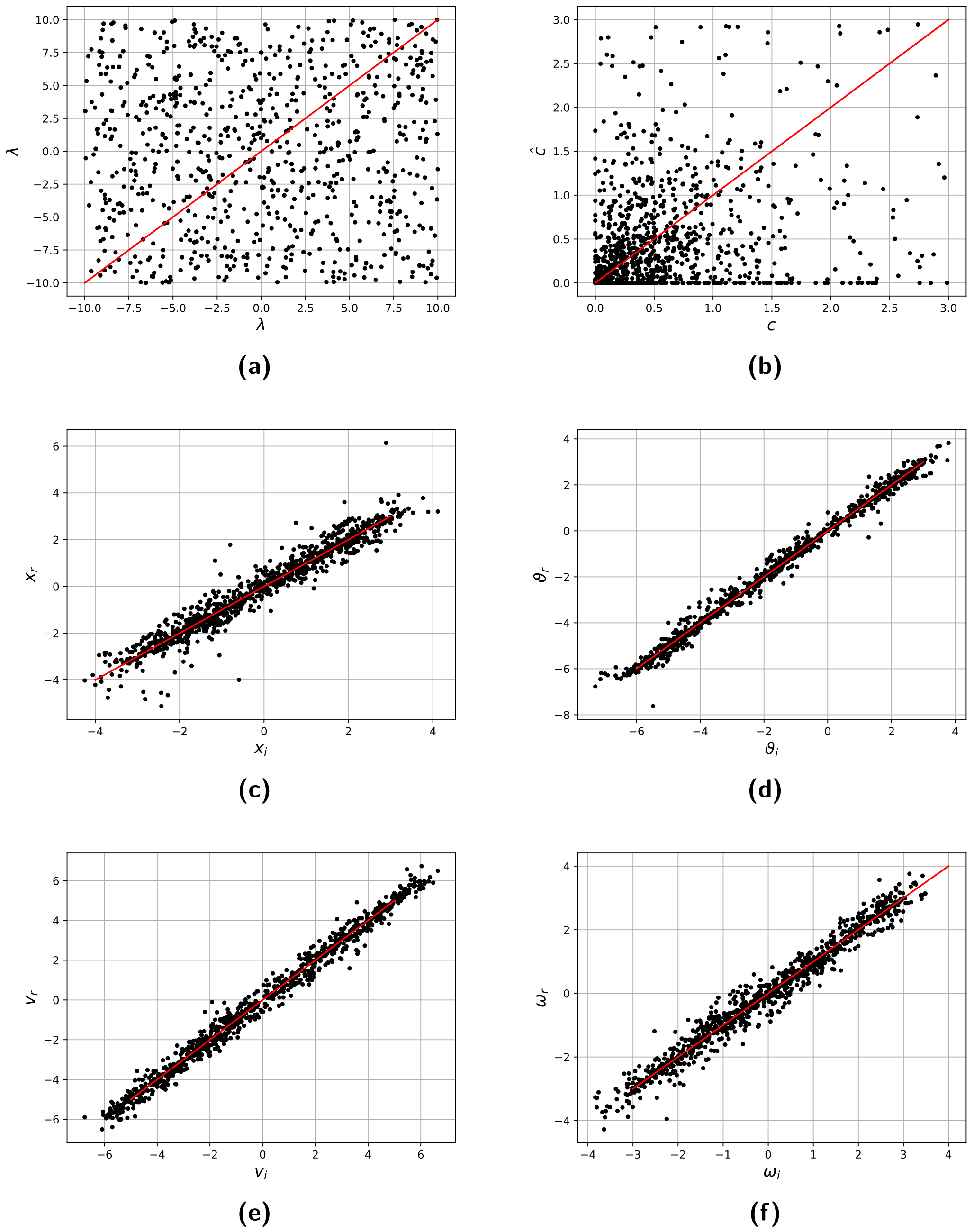


Figure 3-3: (a) and (b) show the estimated values of λ and c plotted against their true values. (c)-(f) show the goal states x, ϑ, v and ω plotted against their reached counterparts. The red line depicts perfect estimation. The large deviation from the red line indicates a high prediction error.

generated. Using different initial costates λ_i for the same initial state x_i will lead to a different state x_r. Recall from Section 2-2 how the control Hamiltonian is constructed when using a quadratic cost function $C(x, u) = w + \frac{u^2}{2}$:

$$\mathcal{H}(x, \lambda_x, u) = w + \frac{u^2}{2} + \lambda_x^T \dot{x} \tag{3-5}$$

Now when computing the optimal control Hamiltionan H^*, the input u is replaced by the optimal control input u^*, which is parameterized by λ_x and x. Hence, the optimal control Hamiltonian is a quadratic function in the terms of λ_x and x. Finally, recall the initial condition that needs to be fullfilled, $\mathcal{H}^*(x, \lambda_x) = 0$ at $t = 0$. This condition was met by solving the condition for the final costate. Since the condition is quadratic, there practically always will exist two valid solutions. Since the relationship that is learned by the kNN model needs to be deterministic, only one of the two valid solutions will be selected. This lead to the belief that the generated dataset D does not cover the full space containing all trajectories enough. The input data for the function approximator are tuples of initial and final states, $\{x_i, x_f\}$. The space these tuples are from can be seen as a *trajectory space* with a dimension two times the state space dimension. A point in the trajectory space corresponds with a trajectory from one point to another in state space. Now ideally, the dataset would contain trajectories from each point in state space to any other point in state space. This would mean the trajectory space is filled up uniformly. Due to the differential constains in the form of equations of motion and limited integration time, this is unlikely. For example, a trajectory from a point with very high positive velocities towards a point with very negative velocities is unlikely to be present. Hence a 100% dataset coverage is likely impossible, however we can see whether or not there is a big difference between the datasets for the single pendulum and the cart pole system.

The dataset coverage can be estimated by doing a fairly simple experiment. The estimation is done by splitting up the trajectory space into multiple bins, and counting the number of datapoints in each of these bins for a generated dataset. When a datapoint point is present in a specific bin, than it means that there is a trajectory starting at a specific part in state space and ends in another specific part of state space. Each dimension in the trajectory space is divided in to four bins. This way a division is made between values that are negative large, negative small, positive small and positive large. The dataset coverage is then defined as the percentage of non-empty bins. The experiment is repeated for different dataset sizes, such that the effect caused by the dataset size is also visible.

The results for this experiment are shown in Figure 3-4. Clearly the coverage for the single pendulum is much better, when using 3000 trajectories in the dataset the coverage is already at 28%, whereas the coverage for the cart pole just reaches 5% even when using 50000 trajectories. Interesting to see as well is that the dataset coverage for the single pendulum saturates at 28%. To estimate the number of trajectories needed for the cart pole system to equal the coverage of the single pendulum system, an optimistic linear extrapolation is done. The estimation is optimistic since the curve in Figure 3-4 is clearly saturating and not linearly increasing. Extrapolating leads to a required dataset size of over 500000 trajectories for a coverage of 25%, but this value is likely to be much higher, assuming that it would ever reach such a coverage.

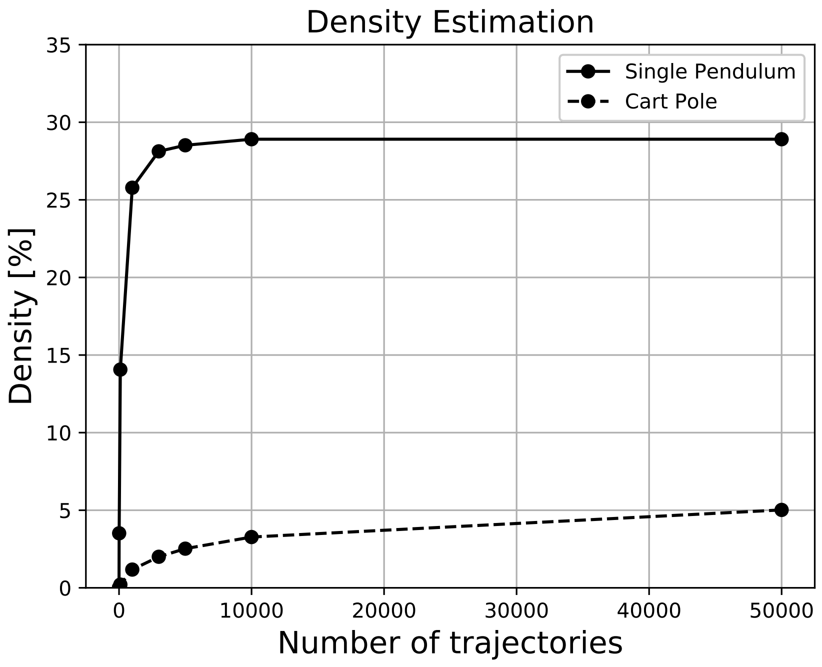


Figure 3-4: Dataset coverage for different dataset sizes. In case of the single pendulum, the trajectory space is much better covered than for the cart pole system. To achieve similar coverage, at least an estimated $500k$ trajectories are needed.

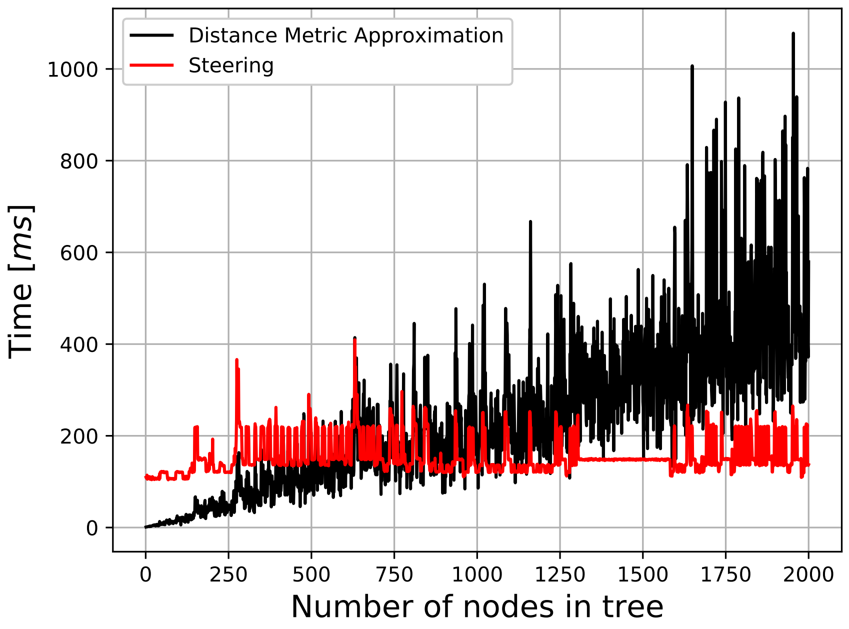


Figure 3-5: Computation times for the functions `approximateDistanceMetric(`$\mathcal{T}$`,`x_r`,`$f(\cdot)$`)` and `steer(`x_{near}`,`x_r`,`φ`,`c`)`. The computation time of the distance metric drastically increases over when the number of nodes in the tree increase, making RRT CoLearn very slow for larger trees. A different approximation method could solve this.

3-3 Computation Time

The main problem with kinodynamic planning is its computation time and RRT CoLearn is an interesting candidate to speed up kinodynamic planning. When investigating the algorithm on the cart pole system in Section 3-1, it appeared that the algorithm was slowing down as the tree grew. To find the cause for this, each step in Algorithm 8 was timed. It was found that two lines of Algorithm 8 were the main contributers to the computation time, namely the functions `approximateDistanceMetric(`$\mathcal{T}$`,`x_r`,`$f(\cdot)$`)` and `steer(`x_{near}`,`x_r`,`φ`,`c`)`. Figure 3-5 shows the computation time of these two steps plotted against the number of nodes in the tree. Two things can be seen on this figure. The steering function takes up a large chunk of time, which stays constant for every tree node. The time needed to estimate the distance metric however grows when more nodes are present in the tree. This is expected, as for every tree in the node a distance metric has to be found. The rate at which the computation time grows however was not expected to be this large.

The speed at which the tree is built can be largely increased by selecting a different function approximator which computes the distance metric. In this section, two different function approximators are investigated that are supposed to be faster than kNN. In [7] a huge speedup was achieved by approximating the optimal cost-to-go using LWPR, hence LWPR will be applied on this data as well. Besides this Artifical Neural Networks will be investigated as well due to its high speed capabilities.

D	RFs	$MSE(\varphi, \hat{\varphi})$	$MSE(c, \hat{c})$	$MSE(x_f, x_r)$	t_{approx}
0.1	22	22.360 ± 0.214	0.405 ± 0.026	0.746 ± 0.022	0.036 ± 0.009
1	567	25.652 ± 0.434	0.376 ± 0.029	0.372 ± 0.013	1.045 ± 0.026
5	2760	46.031 ± 1.271	0.692 ± 0.039	0.262 ± 0.018	6.641 ± 0.719
10	3651	53.043 ± 2.294	0.825 ± 0.066	0.240 ± 0.029	7.488 ± 1.186
20	4868	49.476 ± 2.481	0.923 ± 0.024	0.224 ± 0.024	8.097 ± 0.659
50	7062	52.653 ± 4.074	1.072 ± 0.140	0.262 ± 0.019	10.421 ± 1.096
–	$\begin{bmatrix}4926\\4875\\4911\\4860\end{bmatrix}$	46.378 ± 1.524	0.910 ± 0.072	0.245 ± 0.027	8.313 ± 0.200

Table 3-3: Results for the initial tuning of the LWPR function approximator and the final optimized model. The steering error is larger than when using kNN, and actually slower compared to the results in Table 3-1 and Table 3-2. This is caused by the high number of local models needed to approximate the function, as visible in the second column.

3-3-1 LWPR

LWPR is a function approximator which is based on local linear models which get activated by receptive fields. The width of these receptive fields is a parameter which need to be tuned to get optimal performance. The tuning is done by setting a parameter D which is inversely related to the width, hence a larger D leads to more local models. In order to get the best performance, the value of D can be optimized, however the optimization does need an initial guess in order to converge. To find this initial guess, multiple LWPR models are fitted on a dataset that consists of 3000 trajectories. The trajectories are created by integrating the equations of motion of Section 3-1-3 from $t = 0.0$ to $t = 2.0$ with $\Delta t = 0.01$. The model is then applied on 10 equally generated testsets, containing 1000 trajectories. The score for the initial guess will be the steering error $MSE(x_f, \hat{x}_f)$, the error between the goal state and the reached state. For insight the error in the costate estimation, the error in the cost estimation, the number of receptive fields and the average approximation time are computed as well. The implementation used is made by the authors of the algorithm[54]. In the documentation it is recommended to train the model multiple times with the same data, in order to remove bias towards the datapoints that are entered first. Therefore each model is trained for 10 epochs. The results for finding this initial guess are listed in Table 3-3.

The steering error $MSE(x_f, x_r)$ using LWPR decreases when a larger initial width is selected for the receptive fields, until it reaches $D = 20$. An interesting observation is that the even though the steering error is decreasing, the approximation errors for $\hat{\varphi}$ and $\hat{c}$ are almost constantly increasing for larger D. This indicates overfitting, however since the steering error is smallest for $D = 20$ this initial width is selected as guess for the optimization. The results for the optimized model are shown in Table 3-3 as well.

The steering error actually did not improve compared to the non-optimized model with $D = 20$, even though the difference is marginal. The expected reason for this is the high non-linearity of the relationship between the input data $\{x_i, x_f\}$ and the output data $\{\varphi, c\}$. Since LWPR assumes local linearity, it has difficulty generalizing the data. This also shows in the very high number of local models, almost 5000 models for each output variable. This high number of models also leads to a larger approximation time t_{approx}, of which the intention was to decrease it. Hence, due to the high non-linearity of the relationship between input and output data, applying LWPR actually would lead to an increase in computation time and an inaccurate steering function.

3-3-2 Neural Networks

The second function approximator that was investigated in order to bring down the computation time, is a neural network. The main tuning parameter of a neural network is its shape: one can define the depth of the network by selecting the number of hidden layers, and the width by selecting the number of perceptrons per layer. For simplicity, no layers with a different number of neurons will be used. Hence the entire neural network can be described by its depth and width. Furthermore as activation function the ReLU function from equation 2-39 is selected, which proved empirically to have better performance and faster convergence compared to other activation functions. The implementation used is that of scikit-learn[53], in the form of the `MLPRegressor` class.
To assess the quality of the neural network and the effect of its shape with respect to performance, the previous experiment is repeated. Multiple neural networks are created of different shapes, which are trained on a dataset of trajectories. The dataset that is trained on is equal to the training set used in the previous experiment. The results for this experiment are shown in Table 3-4. The shapes that are chosen for the network are such that there is a range from a very small network (1×16) to a very big network 5×640. Similar to the LWPR model,

	$MSE(\varphi_i, \hat{\varphi}_i)$	$MSE(c_i, \hat{c}_i)$	$MSE(x_f, x_r)$	t_{approx} $[ms]$
1×16	22.483 ± 0.445	0.183 ± 0.012	0.350 ± 0.023	0.397 ± 0.045
1×128	25.040 ± 0.440	0.194 ± 0.019	0.283 ± 0.022	0.376 ± 0.032
5×16	29.495 ± 0.903	0.255 ± 0.022	0.368 ± 0.032	0.438 ± 0.010
5×128	39.376 ± 1.149	0.176 ± 0.015	0.202 ± 0.013	0.641 ± 0.024
1×640	33.608 ± 1.476	0.117 ± 0.013	0.209 ± 0.021	0.560 ± 0.079
5×640	43.514 ± 1.367	0.065 ± 0.005	0.259 ± 0.040	3.073 ± 0.088

Table 3-4: Approximation errors and steering error when using different shapes of neural networks. The lowest steering error was achieved using a (5×128) shaped network.

the results are not very good. The steering error $MSE(x_f, x_r)$ is larger for every shape of network compared to the kNN model. The smallest steering error was achieved when using a (5×128) shaped network, $MSE(x_f, x_r) = 0.202$, the decrease in performance when using larger models is probably due to overfitting. the error is still large, the experiment does have a positive result. The computational time for the neural network is much small than that of kNN, all but the largest network are faster. Hence if we can improve the quality of the steering as well, a neural network could be used for a fast and accurate steering function.
The expected reason for the bad performance of the approximation is the same as for LWPR. The highly non-linear relationship between the input and output data makes it very difficult to generalize it, and thus makes it hard to make a good approximation. So in order to use a neural network for the steering function, the relationship between the input and output data needs to be simplified.

3-3-3 Time Gain

Even though the approximations are not good enough to be used in RRT, the effect of using different approximators than kNN in RRT CoLearn can still be shown. To show this, the experiment of Section 3-1 is repeated, with a minor change. The stopping condition in Line 8 of Algorithm 8 is changed to $N_{nodes} \leq 100$, making the algorithm build a tree until 100 nodes are added. The function `doRRT(`$f(\cdot)$`)` is then timed while using the different approximation methods kNN, LWPR and ANN. The experiment is repeated 100 times and the system it is applied on is the cart pole system. As shape for the neural network (5×128) is chosen, since that shape resulted in the lowest steering error. The LWPR model used is the optimized one. The results are shown in Table 3-5.
Even though the experiment only concerns 100 nodes, the effect of using a different function approximator is still visible. As expected, the LWPR model made building the tree much slower since its approximation was slower overall. Using a neural network however sped up the building of the tree by 1.5 second. This might not seem a lot, but recall that this is for only 100 nodes. The experiment was repeated with $N_{nodes} \leq 2000$ as stopping condition, which led to building times of over 500 seconds for kNN while it remained around 300 seconds for ANN. Hence the effect is significant.

	kNN	LWPR	ANN
Time $[s]$	14.939 ± 0.236	131.642 ± 5.384	13.462 ± 1.010

Table 3-5: Computation time of building a tree with 100 nodes. Using ANN is faster than kNN, LWPR is much slower due to the high number of local models.

3-4 Data Generation

The previous two sections provided two motivations to investigate different forms of generating data for the RRT CoLearn algorithm. The investigation of Section 3-2 lead to the discovery that the trajectories generated using indirect optimal control barely cover the trajectory space, only 5% when using 50000 trajectories for the cart pole system. This low coverage is expected to be the reason RRT CoLearn does not converge for the cart pole system. Different data generation could solve this problem. Second, in Section 3-3, the effect of the approximation method on the computation time was investigated. It showed that by using kNN, the needed time for adding nodes to the tree increased quickly when the tree grew. Using a different approximator such as LWPR or ANN could highly increase the building speed. However, due to the difficult, highly nonlinear relationship between the input data $\{x_i, x_f\}$ and the output data $\{\varphi, c\}$, it was impossible to properly generalize the data in a simplified model. If a different approximator is to be used, this relationship has to be simplified and thus the data generation has to change. In this section Hamiltonian formalism is investigated as a candidate to for the data generation.

3-4-1 Hamiltonian Formalism

In the previous sections the equations of motion were derived using Lagrangian mechanics. Hamiltonian formalism can greatly simplify the equations of motion, as is shown in the following section.

Derivation

Most of the times dynamics of systems are modelled using Lagrangian formalism. In this method, first a Langrangian $\mathcal{L}$ is defined which is the difference between all kinetic energy T in the system and all potential energy V, as in equation 3-6. Now when each configuration coordinate is described q_i, and its time derivative as $\dot{q}_i$, the eqatuations of motion can directly be derived from this Lagrangian as in Equation 3-7. Using this formulation, the accelerations $\ddot{q}$ are written in terms of the velocities $\dot{q}$ and the configuration q of the system.

$$\mathcal{L} = T - V = \sum_i \frac{1}{2} m_i v_i^2 - V \tag{3-6}$$

$$\frac{d}{dt}\frac{\partial \mathcal{L}}{\partial \dot{q}_i} - \frac{\partial \mathcal{L}}{\partial q_i} = 0 \tag{3-7}$$

In Hamiltonian formalism the system is described in terms of generalized momenta p instead of velocities. The derivation of the equations of motion is done in a similar way, but using a different energy function, namely the Hamiltonian $\mathcal{H}(p, q)$. Note that this Hamiltonian is a different one than the one used in the optimal control in Section 2-2. To find the Hamiltonian, the Legendre transformation in Equation 3-8 can be used. However, now it still is a function the velocities $\dot{q}_i$. To get rid of these, $\dot{q}$ can be written in terms of p and q as in Equation 3-9. Substituting $\dot{q}$ then leaves the Hamiltonian only in terms of the generalized coordinates p and q.

$$\mathcal{H} = \sum_i \dot{q}_i \frac{\partial \mathcal{L}}{\partial \dot{q}_i} - \mathcal{L} \tag{3-8}$$

$$p = \frac{\partial \mathcal{L}}{\partial \dot{q}} = M(q)\dot{q} \rightarrow \dot{q} = M^{-1}(q)p \tag{3-9}$$

The equations of motion for a system can then be found by applying Equation 3-10. These equations can then again be transformed into the equations for optimal state evolution by applying the same steps as in Section 2-2. Doing so leads to a different solution for the optimal control input u^*.

$$\begin{aligned} \dot{q} &= \frac{\partial \mathcal{H}}{\partial p} \\ \dot{p} &= -\frac{\partial \mathcal{H}}{\partial q} + u \end{aligned} \tag{3-10}$$

Substituting Equation 3-10 into Equation 2-8 and then applying the derivation steps for optimal control leads to Equation 3-11. To differentiate between the Hamiltonion used for

	$MSE(\varphi_i, \hat{\varphi}_i)$	$MSE(c_i, \hat{c}_i)$	$MSE(x_f, x_r)$	t_{approx} $[ms]$
kNN	48.129 ± 1.894	0.617 ± 0.037	0.143 ± 0.008	1.189 ± 0.038
ANN	38.227 ± 1.848	0.056 ± 0.019	0.165 ± 0.013	0.730 ± 0.033
LWPR	49.336 ± 1.383	0.877 ± 0.067	0.257 ± 0.014	16.349 ± 0.415

Table 3-6: Results

the equations of motion and the control hamiltionan, the latter is indicated with a subscript $\mathcal{H}_c$.

$$\begin{aligned}\mathcal{H}_c(x,u) &= C(x,u) + \lambda_x^T \dot{x} \\ \mathcal{H}_c(p,q,u) &= C(p,q,u) + \lambda_q^T \dot{q} + \lambda_p^T \dot{p} \\ \mathcal{H}_c(p,q,u) &= C(p,q,u) + \lambda_q^T \frac{\partial \mathcal{H}}{\partial p} + \lambda_p^T \frac{\partial \mathcal{H}}{\partial q} + \lambda_p^T u\end{aligned} \tag{3-11}$$

The difference with the normal derivation happens when we take the derivative of $\mathcal{H}_c$ to u to find u^* which minimizes $\mathcal{H}_c$:

$$\frac{\partial \mathcal{H}_c}{\partial u} = \frac{\partial C}{\partial u} + \lambda_p \tag{3-12}$$

Hence if $C(p,q,u)$ is chosen such that it is of the from of Equation 3-13, with W being a weight matrix on the inputs, then the optimal control input u^* is found to be as in Equation 3-14.

$$C(p,q,u) = w + \frac{1}{2} u^T W u \tag{3-13}$$

$$u^* = -W \lambda_p \tag{3-14}$$

3-4-2 Experiment

Whether or not using Hamiltonian formalism indeed improves the quality of the dataset can be tested with an experiment similar to Algorithm 9, but with some minor changes. The sampling will still be done in state space, such that the bounds of the sample space is still intuitive. The sampled states will then be transformed to phase space by applying Equation 3-9, in which the approximation and steering is done. The reached state is then transformed back to state space to be able to compute the steering error. Hence, the dataset used for training and testing are generated differently. The training set contains samples from phase space $\{q_i, q_f\}$, whereas the test set contains samples from state space, $\{x_i, x_f\}$. The cost and input parameterization $\{c, \varphi\}$ are from phase space in both datasets. The datasets used are compiled of 3000 trajectories which are generated by integrating the equations of motion from $t = 0.0$ to $t = 2.0$ with $\Delta t = 0.01$. All three function approximators handled in the previous section, kNN, LWPR and ANN will be tested on this data. The metaparameters of the function approximators are found in the same way as in the previous section. In case of the ANN, the shape of the network that was found to perform best is (5×128). The LWPR approximator was found to perform best with an initial receptive field width of $D = 10$. The results are shown in Table 3-6. The effect of implementing the equations of motion in Hamiltonian formalism are not as initially was expected. In case of the kNN approximator the steering error slightly decreased, $MSE(x_f, x_r) = 0.143$ versus 0.148 in Lagrangian formalism,

which is not significant. The approximation errors for φ and c are also in the same order. Even though the equations of motion are much simpler, the data still proves too difficult to learn. The relationship between the initial and final state pairs in phase space $\{q_i, q_f\}$ and the corresponding initial costates remains highly non-linear. The only difference is the transformation from state space to phase space, given in equation 3-9. This transformation should not infer much extra difficulty compared with the original data.
In case of the LWPR approximator, the results remained similar to when using Lagrangian mechanics. The main difference is the computation time which almost doubled. This is caused by the high number of local models that are added to the model, the number of receptive fields are $RF = \begin{bmatrix} 8464 & 8363 & 8471 & 8303 \end{bmatrix}$. Clearly LWPR is not capable of generalizing the data properly, making it actually slower than kNN.
When using ANN as approximator the steering function does appear to perform slightly better with $MSE(x_f, x_r) = 0.164$ compared to 0.202 when using Lagrangian mechanics. Interesting to see is that the approximation of the cost appears to be much better, $MSE(c, \hat{c}) = 0.056$ instead of 0.176 when using the same shape. However when using a very wide network (1×640) with Lagrangian mechanics the approximation error of the cost to go was in the same order of magnitude.

3-4-3 Direct Optimal Control

In an attempt to find a dataset that has more correlation between the input and output data, a small investigation was done in to direct optimal control. In direct optimal control the control input is parameterized into a vector u of fixed length. The optimal control vector u^* that steers the system from x_i to x_f is then found by optimizing a certain cost function, subject to the initial state constraint $x(0) = x_i$, the final state constraint $x(t_f) = x_f$ and the differential constraint $\dot{x} = f(x, u)$. As already stated in the background, solving this boundary value problem would solve the motion planning problem entirely. The problem is that solving it is very difficult.

$$\begin{aligned} u^* = & \arg\min_u C(x, u) \\ \text{s.t. } & x(0) = x_i \\ & x(t_f) = x_i \\ & \dot{x}(t) = f(x, u) \end{aligned} \tag{3-15}$$

Instead of solving the full problem, a dataset is created by randomly sampling a random x_i and a random x_f in the neighborhood of x_i and solving that small direct optimal control problem. This way a short trajectory from some x_i to some x_f is created which is stored as shown in Equation 3-16.

$$\mathcal{D} = \begin{bmatrix} x_0 & x_1 & u_0^* \\ x_1 & x_2 & u_1^* \\ \vdots & \vdots & \vdots \\ x_{n-2} & x_{n-1} & u_{n-2}^* \\ x_{n-1} & x_n & u_{n-1}^* \end{bmatrix} \tag{3-16}$$

A big problem with direct optimal control however is its sensitivity to an initial guess. The optimizer used to find the control vector u needs an initial guess to start its optimization.

If this guess is far away from an optimum, the algorithm will not converge. In practice this means that a very well educated guess needs to be made, which is often impossible. For the single pendulum data generation worked fine, however it turned out to be difficult to create a similar dataset for the cart pole system. However, most importantly, the investigation into direct optimal control inspired for a different approach based on learning the inverse dynamics. Looking at the dataset in Equation 3-16, each datapoint can be read as '*To get from x_0 to x_1, apply u_0^**'. Instead of using the optimal control input u^* one could also just apply a random u. This idea is handled in depth in Chapter 4.

3-5 Summary

In this chapter, the RRT CoLearn algorithm has been investigated in-depth. First the full algorithm was implemented in order to reproduce the results of [14]. The algorithm was applied to the single pendulum setup with the task to compute a swingup. The only difference was the hardware performed on. This lead to a computation time slightly longer than in [14], but in the same order of magnitude. In order to answer the question '*What are the consequences for RRT CoLearn when it is applied to a multi-DOF system?*', the algorithm was applied to a cart pole system. In this case, the algorithm did not converge which lead to a more in depth investigation of seperate parts of the algorithm. Since approximation of cost-to-go has been done before succesfully, the focus lay on the steering function. An experiment was performed to assess how good the approximated steering function could steer the system towards a goal state, and a comparison was made between the single pendulum and the cart pole system. When using a larger dataset, the steering error was in the same order of magnitude as with the single pendulum. The steering function thus was not the problem.
Next the quality of the generated datasets was considered. An experiment was set up to estimate how much of the trajectory space was covered by the data generated using indirect optimal control. This was done by splitting up the trajectory space into multiple bins and count the number of occurances in each box. This showed that, in case of the single pendulum, a 28% coverage was achieved compared to a 5% coverage for the cart pole system. For the cart-pole system to achieve the same coverage at least 500000 trajectories would be needed. The true figure is believed to be much higher due to the very optimistic extrapolation it was calculated with, assuming that it will even reach it at all. This lead to the belief that a different way of generating the data would benefit this coverage, and consequently make the algorithm converge.
Since the main problem with kinodynamic is the computation time, this was investigated for RRT CoLearn as well. During the first baseline experiment, it was noted that the time for adding a node to the tree increased when the tree grew. This is expected, since per random sample the distance metric needs to be computed for every node in the tree. The current implementation using kNN however took more than 1 second per node when the tree contained over 1500 nodes. To decrease this computation time, different function approximators than kNN were investigated. The distance function approximation and steering function were both implemented using ANN and LWPR. It turned out that both function approximation methods were not capable of generalizing the data. The steering error was significantly higher, and to achieve this such a large model was needed that the intended gain in timing dissapeared. To still be able to use a different, faster function approximator, the relationship between the input and output data needs to be simplified.

Since both investigations lead to the idea that the problem lies with the data generation, the last section investigated formulating the problem in Hamiltonan formalism. Using Hamiltonian instead of Lagrangian formalism leads to the optimal control input being parameterized by its corresponding costate. The idea was that this simplified relationship would ease the learning, and thus improve the performance of the steering function and the algorithm overall. When using kNN the performance remained equal, which was to be expected since kNN does not generalize the data but simply finds the best match from a dataset. In case of the neural network the performance did improve. The steering error decreased from $MSE(x_f, x_r) = 0.202$ to 0.164 when using a (5×128) network. Hence the steering error did not decrease enough to match that of the single pendulum. Using LWPR as approximator did not increase the performance, it actually was almost twice as slow as when using Lagrangian mechanics.
Finally, direct optimal control was briefly investigated, but discarded for two reasons: Generating the data using direct optimal control requires a good initial guess, which as a consequence had that the generation often failed due to not finding a solution. Second, it inspired to investigate the learning of inverse dynamics, which immediately showed promising results. This investigation will be handled in the next chapter.

Chapter 4

Learning Inverse Dynamics

In the previous chapter it was found that RRT CoLearn did not converge for other systems than the single pendulum. This led to a broader investigation in how to apply supervised learning in kinodynamic planning, with the research question '*How can supervised learning be used to speed up kinodynamic planning?*'. This question led to a new steering function based on learning the inverse dynamics of the system, which will be described in this chapter.

4-1 Data Generation

RRT CoLearn replaces the steering function with a kNN function approximator which tries to approximate the mapping between a trajectory and a control input parameterization. The trajectories the approximator is trained on are generated by using indirect optimal control, and the input parameterization the approximator needs to learn are the initial costates. Two motivations were found for discarding indirect optimal control and using a different type of data generation. First, the used method led to a small coverage of the trajectory space, which was expected to be the cause for the algorithm not converging to a solution for other systems then the single pendulum. A different data generation method could improve the coverage and thus make the algorithm converge for other systems as well. Second, a large improvement in computation time can be achieved when a different approximation method is used than kNN. However, training different function approximators than kNN on the data which was generated using indirect optimal control, led to very high approximation and steering errors. Different forms of data generation were tried in order to overcome this problem, which finally led to the idea of learning inverse dynamics.
The new proposed form of data generation no longer makes use of optimal control. Instead, the trajectories are created by applying random control inputs. The data is generated by first sampling a random initial state $x_i \in \mathcal{X}$. For every initial state, a random control input u is sampled from $\mathcal{U}$. The system's equations of motion are then integrated for a fixed time step Δt, which results in a final state x_f. This integration leads to a single datapoint: $\{x_i, x_f, u\}$ which is appended to a dataset $\mathcal{D}$. To make sure that multiple directions through the state

space are available from every x_i, which should improve the dataset's coverage of the trajectory space, the sampling of u and integration are done multiple times for every x_i. The integration of the equations of motion is done using the Runge-Kutta method. Algorithm 10 outlines the procedure for data generation. The main difference between indirect optimal control and the new data generation method is the control input selection. It was expected that by using indirect optimal control the system was already steered into a certain direction, leading to a poor coverage of the trajectory space. By using random control inputs, it is more likely that the system is steered into all directions.
All the trajectories are stored in a dataset $\mathcal{D}$ which is then used to train a function approximator in order to learn the mapping $f : \{x_i, x_f\} \mapsto u$. The mapping between $\{x_i, x_f\}$ and u is expected to be much easier to learn than the mapping from trajectory to costate λ, since it simply are the inverse dynamics that need to be learned. For example in [34] similar dynamics are approximated succesfully and used in computed torque control. Since the new method learns the inverse dynamics, I named it *Inverse Dynamics Learning* (IDL).
Next a series of experiments will be done to find out if using IDL solves the problem of poor coverage of the trajectory space and whether the mapping $f : \{x_i, x_f\} \mapsto u$ indeed is easier to learn.

Algorithm 10: Algorithm used for generating inverse dynamics data.

```
1 $\mathcal{D} \leftarrow \{\}$
2 for $k \leftarrow 1$ to $NUM\_TRAJECTORIES$ do
3     $x_i \leftarrow$ randomSample($\mathcal{X}_{free}$)
4     $u \leftarrow$ randomSample($\mathcal{U}$)
5     for $k \leftarrow 1$ to $NUM\_SUB\_TRAJECTORIES$ do
6         $x_f \leftarrow$ integrateEOM($x_i$,$u$,$\Delta t$)
7         append($\mathcal{D}$,$\{x_i, x_f, u\}$)
8     end
9 end
```

4-1-1 Dataset Coverage

One of the two reasons for investigating different forms of data generation was to improve the coverage of the sample space. Using indirect optimal control led to a coverage of 28% for the single pendulum system but only 5% for the cart pole system, which is expected to be the cause for RRT CoLearn not converging. To assess whether the coverage is indeed better when using IDL based generation, the same experiment is repeated as was done in Section 3-2-3.
Each dimension of the trajectory space is split up into four bins, splitting the entire trajectory space up into multiple boxes. The coverage of the trajectory space is then defined as the percentage of boxes that have datapoints in them. The inverse dynamics dataset is generated using $\Delta t = 0.1$s, the RRT CoLearn dataset is generated using $\Delta t = 0.01$s and $t_{max} = 2$s. The reason for taking much shorter trajectories is that in IDL each branch in the tree will have the same time length Δt, hence shorter trajectories lead to a higher resolution in searching through state space and the found paths will have shorter time duration. Note that the

indirect optimal control dataset contains trajectories from multiple time lenghts, from $t = 0$s to $t = 2$s in steps of $\Delta t = 0.01$, hence much more than in the IDL dataset, both shorter and longer and with a higher resolution. This in combination with the fact that the differential equations for the optimal state evolutions are unstable leads to trajectories that can travel across the entire state space. Doing so within $\Delta t = 0.1$s is simply impossible with the limited control input used in IDL. Now, if the form of data generation was not the problem, using indirect optimal control would lead to a much higher coverage than IDL. However, the results in Figure 4-2a and 4-2b show differently.
The indirect optimal control dataset does provide better coverage than when using the IDL approach. However, it does not show a huge difference, which should be the case due to the much longer trajectories present. Even though the indirect optimal control dataset provides better coverage, it is still expected that the IDL approach will work better in RRT. The reasoning behind this is shown in Figure 4-1, where some mockup trajectories are shown in statespace, all starting from the same initial state x_i. The green boxes represent the filled boxes from last experiment when using IDL, the grey boxes show them for indirect optimal control. The trajectories generated using indirect optimal control provide longer trajectories, leading to more filled boxes and thus a higher coverage of the trajectory space. However, they all evolve into the same direction. Hence when building a tree using these trajectories, it will not be able to explore the entire state space. Using IDL might lead to a lower trajectory space coverage, but since the trajectories move in all directions it will lead to a much better exploration when doing RRT, since it does contain trajectories in all directions.

To find out whether this reasoning is correct, i.e. that the trajectories generated using the

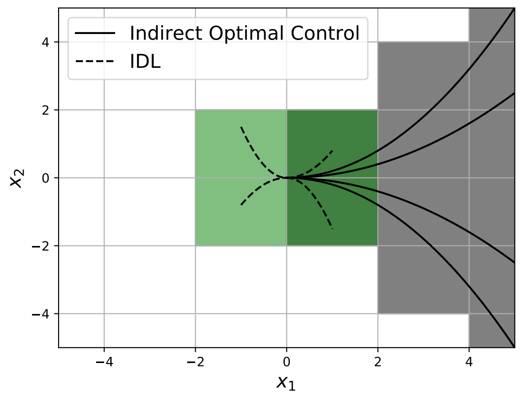
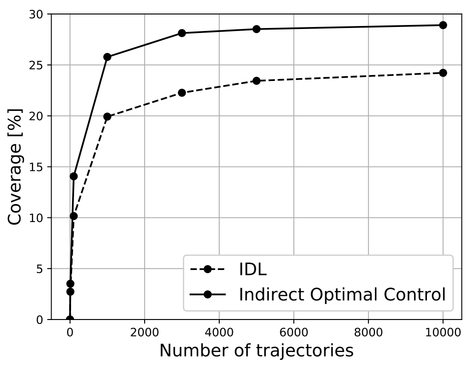


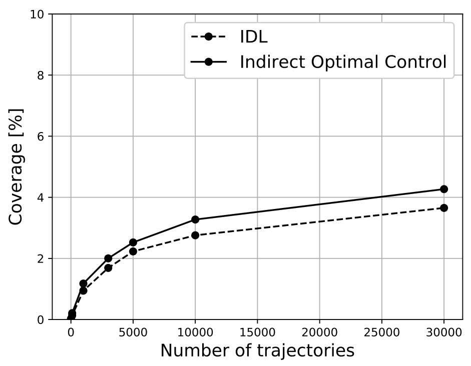
Figure 4-1: Illustration of the difference between trajectories generated using indirect optimal control or the IDL approach. The indirect optimal control trajectories lead to a better coverage since they are longer, as shown by the area in grey. However, the IDL approach leads to trajectories into more directories leading to the coverage shown in green.

IDL approach travel into more different directions than when using indirect optimal control, the experiment is repeated but now $\Delta t = 2$s is used for the IDL dataset. Figure 4-2c and 4-2d show the results when using equal trajectory length. The trajectory space coverage is indeed much higher when using the IDL approach. In case of the single pendulum the coverage went up to 47%, the coverage for the cart pole increased to 11%. From this the conclusion can be made that when using IDL the trajectories travel into more directions. This means that with

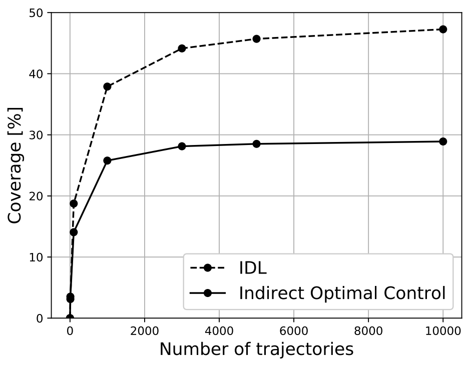
a shorter time step of $\Delta t = 0.1$ the coverage of the trajectory space might not be higher, but it is better.
Finally note that, even though the number of trajectories used while generating the data is equal, the number of datapoints in a dataset are not. A single integration using indirect optimal control leads to up to 200 datapoints per trajectory, whereas the IDL approach only yields one. Hence the number of trajectories can be increased a lot more when using IDL, while still having less datapoints. This would lead to an even better coverage with still a smaller number of datapoints.



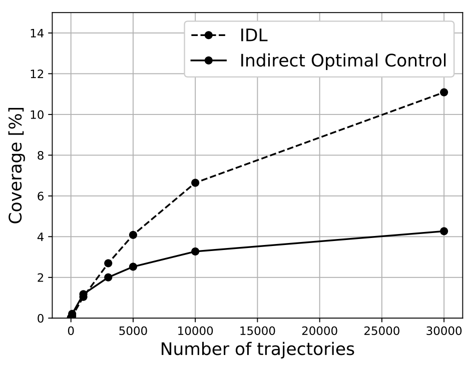
(a)



(b)



(c)



(d)

Figure 4-2: Coverage of the trajectory space with a dataset generated using indirect optimal control and IDL. (a) and (b) show the coverage for the single pendulum and the cart pole system respectively, where the inverse dynamics dataset is generated using $\Delta t = 0.1$s. (c) and (d) shows the coverage when using $\Delta t = 2.0$s. The indirect optimal control dataset provides a higher coverage when using short trajectories, however the coverage when using IDL is better. This is shown by (c) and (d) where the trajectory lengths are equal but IDL leads to a higher coverage.

4-2 Approximating Inverse Dynamics

The previous section showed that the dataset coverage indeed improved when the IDL approach was used, however for it to be applicable in RRT, the mapping $f : \{x_i, x_f\} \mapsto u$ needs to be learnable as well. The mapping between the trajectory $\{x_i, x_f\}$ and the input parameterization φ used in RRT CoLearn showed to be too difficult to learn. To find out whether the mapping $f : \{x_i, x_f\} \mapsto u$ is easier to learn, two function approximators will be tested, neural networks and LWPR.

4-2-1 Neural Network

To find out whether the mapping $f : \{x_i, x_f\} \mapsto u$ can be approximated by a neural network, several networks will be trained on the dataset to learn the mapping $f : \{x_i, x_f\} \mapsto u$. As a score for the performance the steering error $MSE(x_f, x_r)$ will be used. However for insight the approximation error of the control input $MSE(u, \hat{u})$ and the approximation time t_{approx} will be computed as well. The dataset the model is trained on contains 30000 trajectories generated with $\Delta t = 0.1$s. The equations of motion used are those of the cart pole system, derived in Section 3-1-3. The results are shown in Table 4-1.

The first thing that stands out is the small approximation and steering errors. The steering

	$MSE(u, \hat{u})$ [$\times 10^{-3}$]	$MSE(x_f, x_r)$ [$\times 10^{-3}$]	t_{approx} [ms]
(1×16)	37.926 ± 1.596	9.072 ± 0.557	0.564 ± 0.027
(3×16)	5.392 ± 1.250	1.226 ± 0.262	0.653 ± 0.043
(5×16)	4.678 ± 1.575	1.006 ± 0.361	0.662 ± 0.032
(1×32)	17.547 ± 0.639	3.871 ± 0.147	0.550 ± 0.018
(3×32)	1.795 ± 0.057	0.408 ± 0.016	0.604 ± 0.009
(5×32)	1.196 ± 0.047	0.264 ± 0.015	0.690 ± 0.038
(1×64)	7.865 ± 0.354	1.784 ± 0.108	0.603 ± 0.059
(3×64)	1.230 ± 0.015	0.278 ± 0.015	0.825 ± 0.176
(5×64)	4.097 ± 0.108	0.846 ± 0.026	0.956 ± 0.203

Table 4-1: Approximation error and steering error for different shapes of neural network. The best steering error is achieved with a (5×32) network, $MSE(x_f, x_r) = 0.264 \cdot 10^{-3}$. The steering error is almost 300 times smaller than when using indirect optimal control, which in the previous chapter was found to be $MSE(x_f, x_r) = 0.076$. Even with the smallest, (1×16) network, the steering error is smaller than when using indirect optimal control. The wider networks with 64 neurons per layer shows signs of overfitting, since the steering error increases.

error $MSE(x_f, x_r)$ is three orders of magnitude smaller than in RRT CoLearn, even for the worst performing neural network. This proves that the data that is generated using the new approach has a relationship which is indeed much easier to learn. The best results were achieved using a (5×32) network, with a steering error of $MSE(x_f, x_r) = 0.264 \times 10^{-3}$. The approximation time when using this shape was reduced to $t_{approx} = 0.690$ seconds, which is almost half of the time the kNN approximator needed.

4-2-2 LWPR

The same experiment will be repeated but now for LWPR. First the function approximator is trained with a fixed receptive field width D. This is repeated for multiple D in order to find the best guess which will then be used for optimizing the receptive field width. The results are shown in Table 4-2. Again it shows that the mapping $f : \{x_i, x_f\} \mapsto u$ can be

D	RFs	$MSE(u, \hat{u})$ [$\times 10^{-3}$]	$MSE(x_f, x_r)$ [$\times 10^{-3}$]	t_{approx} [ms]
0.5	301	127.741 ± 5.758	22.080 ± 1.279	0.086 ± 0.001
1.0	955	80.110 ± 2.618	13.553 ± 0.646	0.286 ± 0.006
1.5	1874	60.344 ± 2.205	9.746 ± 0.707	0.751 ± 0.020
2.0	3044	52.204 ± 2.654	8.437 ± 0.508	1.267 ± 0.008
3.0	5793	51.022 ± 1.559	7.929 ± 0.467	2.567 ± 0.074
1.5	7295	71.712 ± 3.922	10.643 ± 0.830	3.259 ± 0.249

Table 4-2: Approximation error and steering error for different initial widths of the LWPR receptive fields. Optimizing D actually leads to worse results. Initially a larger D leads to lower steering error, at the cost of longer approximation time. $D = 1.5$ is selected as initial guess for the optimization.

approximated much easier, for all initial widths tested the steering error was much larger than when using indirect optimal control. Of all the trained LWPR models with a fixed receptive field width, the best result in terms of steering error was achieved with $D = 3.0$, which led to $MSE(x_f, x_r) = 7.929 \times 10^{-3}$. However, this also led to a relatively large approximation time of $t_{approx} = 2.567$ms, compared to $t_{approx} = 0.690$ms when using neural networks. Taking this in account, the model with $D = 1.5$ is selected for optimization. The steering error of $9.746 \cdot 10^{-3}$ is only slightly higher, but the approximation time is reduced to a third.
Optimizing however led to a worse performance than when using a fixed width. The optimized model (shown in the bottom row of Table 4-2) achieved a steering error of $MSE(x_f, x_r) = 10.643 \times 10^{-3}$. Which is slightly higher than when using $D = 1.5$. The expected reason for this is overfitting. The optimizer minimizes a cost function which depends on both the approximation error and a penalty for the number of receptive fields. The penalty is a tuning parameter which needs to be tuned. However, tuning this parameter yielded worse results for any value but the default. Since the difference between the optimized width and the fixed width is relatively insignificant, simply the model with the fixed width will be selected as best model.

4-3 Fixed Time Inversed Dynamics in RRT

So far a new data generation method has been proposed which is based on learning the inverse dynamics of the system. It was shown that this new form of data generation led to a better coverage of the trajectory space and that the mapping between the trajectory $\{x_i, x_f\}$ and the required control input u can be approximated very accurately and thus that a system can be steered from some x_i to some x_f very accurately as well.
The next step is to apply the new function in RRT and test whether it converges for other systems than the single pendulum as well. However, before this can be done another problem has to be solved. RRT CoLearn inherently provided a distance metric, the cost to go, which

is no longer available since the trajectories learned are no longer optimal. Hence a different distance metric needs to be defined.

4-3-1 Distance Metric

The IDL steering functions attempts to steer the system from some node x_i to another node x_f. The system will most likely not reach x_f exactly, but it will move towards x_f. The error between the goal state x_f and the reached state x_r can be used as a distance metric. By doing so the node the node from which the random sample x_r is best *reachable* is selected for expansion. Please note that an indepth review of the distance metric is beyond the scope of this thesis, this section just explains how it is implemented and shows that it works. Various studies have shown that the optimal cost-to-go would be a better distance metric[6][23][15]. To show that the steering error can be indeed seen as a measure for reachability, the steering error is computed for points which are known to be reachable and randomly drawn $\{x_i, x_f\}$. The steering error is computed using a (5×32) neural network. In Figure 4-3 the steering error is plotted. The points on the left are steering errors for $\{x_i, x_f\}$ generated by integrating the equations of motion, hence these points are known to be reachable. The points on the right are steering errors for randomly drawn x_i and x_f, which are most probably but not necessarily unreachable. A clear difference is visible between the reachable points and the randomly drawn points, the steering error is much lower in case of reachable points. Hence the steering error could be interpreted as a measure of reachability.

Computing the error for every node in the tree is a tedious process which will drastically

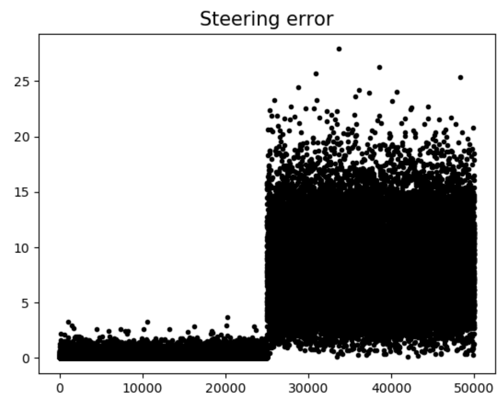


Figure 4-3: The difference in steering error for $\{x_i, x_f\}$ which are known to be reachable and randomly drawn $\{x_i, x_f\}$. The randomly drawn pairs on average produce a much higher steering error.

slow down the building of the tree. Therefore the steering error will be approximated as well. The approximation is implemented as a second neural network, which will approximate the mapping $f_\varepsilon : \{x_i, x_f\} \mapsto \varepsilon$, where ε is the steering error. The approximated mapping is schematically shown in Figure 4-4. When using a neural network, the shape of the network is chosen to be equal to the neural network approximating the steering input. When using LWPR, the same initial widths are used. Note that optimizing the network shape or RF field widths as done in Section 4-2 could further improve the computation time, however for a proof of concept this was not deemed necessary. To make sure the dataset the approximator is trained on is rich enough, the training set is created by sampling the $\{x_i, x_f\}$ that are fed into the function of Figure 4-4 from both the reachable set and a random set.

To find out whether the approximated steering error can be used as a distance metric, it

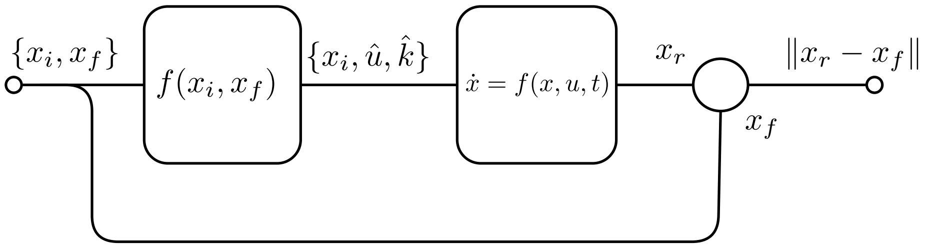


Figure 4-4: Schematic picture of the error mapping that is used as distance metric.

first is tested whether the steering error can be approximated at all. A testset is generated containing both reachable and unreachable datapoints, similar to the datapoints shown in Figure 4-3. This dataset will then be fed into the error approximator $f_\varepsilon(\cdot)$. The quality of the approximation is tested by looking at the MSE between the error and the estimated error, $MSE(\varepsilon, \hat{\varepsilon})$. The system used is the cartpole system and the dataset contains 30000 datapoints. The estimation error is computed for 10 different models, each estimating 1000 datapoints. The estimation error and the approximation time t_{approx} are shown in Table 4-3.

From these results it can be concluded that the steering error can indeed be approxi-

	$MSE(\varepsilon, \hat{\varepsilon})$	t_{approx} [ms]
ANN	0.020 ± 0.002	0.161 ± 0.002
LWPR	0.534 ± 0.019	2.130 ± 0.078

Table 4-3: Approximation error of the distance metric, the expected steering error. Using a neural network leads to a much lower approximation error, and it does so faster than when using LWPR.

mated. The neural network does a much better job approximating the steering error, with $MSE(\varepsilon, \hat{\varepsilon}) = 0.020$ compared to $MSE(\varepsilon, \hat{\varepsilon}) = 0.534$ when using LWPR. On top of that the neural network does it a lot faster as well, almost 20 times faster than LWPR.

Now that it is known that the steering error can be approximated, it has to be tested whether it can be used as distance metric in RRT or not. This is a question that is difficult to answer directly, therefore its performance has been tested empirically. It is checked whether the nodes that are selected for expansion for a given random sample ‘make sense’ intuitively. This is done by building a tree for a single pendulum system. This system is chosen since its state space is easy to visualize and interpret, because it is a 1-DOF system. Furthermore, the system is made underactuated by limiting the control input to $u \in [-1, 1]$. In an underactuated system, a node that is near in euclidean space can actually be very hard to reach, since it cannot travel there directly. A good distance measure should be able to discriminate these nodes from nodes that can actually reach the sample. To assess whether the approximated steering error can do this, it is monitored which random sample leads to the extension of which nodes while building a tree. An example of such a test is shown in Figure 4-5, where six steps in the building of the tree are shown. The green star in each figure shows the randomly drawn sample, the red dot marks the node that is selected for extension. As follows from these figures, the nodes that are selected are often not the nodes that are nearest in terms of euclidean distance. For example when a sample is drawn with a certain angle ϑ_r, then a node is selected for expansion which has a $\vartheta_{near} < \vartheta_r$ and a positive angular velocity ω. These examples are clearly visible in 4-5, from which it is concluded that the selected nodes indeed make sense intuitively and thus that the distance metric can be used as a distance metric.

The final test for the distance metric is of course simply applying it in RRT. If the distance metric does not work, the algorithm most likely will not converge. Therefore it is expected that the algorithm will perform better when using neural networks, since that led to a higher accuracy in approximating the steering error.

4-3-2 RRT Results

With the new distance metric which was empirically shown to work properly, the new steering function based on learning inverse dynamics can be tested in the full RRT algorithm. The experiment done is listed in Algorithm 7 and 8, with the only difference that the cleaning of the data is skipped, since it is no longer necessary. The experiment is repeated using both LWPR and neural networks for the function approximators, with the hyperparameters found in the previous section. The algorithm is applied on the swingup problems for the single pendulum and the cart pole system, which is repeated 100 times. In case of the single pendulum the planning attempt is stopped when the tree contains more than 2000 nodes. In case of the cart pole system up to 5000 nodes are allowed in the tree. When a tree contains more than 2000 or 5000 nodes, the planning is stopped and the attempt is considered a failure. As a measure for the performance, the average planning time and number of nodes in the tree are measured. The results are shown in Table 4-4.

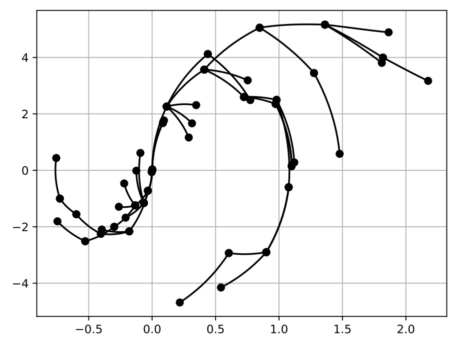
When using a neural network as function approximator, the algorithm converges much

		Succes rate [%]	Planning Time [s]	Number of Nodes
LWPR	Single Pendulum	30.0	170.513 ± 176.461	956.3 ± 568.7
	Cart Pole	- - -	- - -	- - -
ANN	Single Pendulum	95.0	3.480 ± 3.543	667.3 ± 482.8
	Cart Pole	25.0	67.659 ± 62.766	2215.6 ± 1211.9

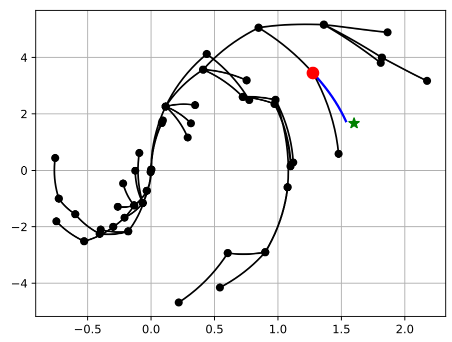
Table 4-4: Results of planning a swingup for the single pendulum and cart pole system when using LWPR or ANN and fixed time trajectories. The algorithm performs much better when using ANN than with LWPR.

more often than when using LWPR. The expected reason for the neural network performing better is its capability to approximate the distance metric as well, since the approximation of the steering function worked fine when using LWPR. The high number of required tree nodes when using LWPR reinforces this presumption, since that is a typical sign of a failing distance metric.

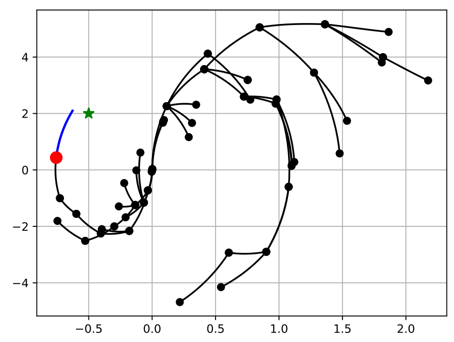
Most interesting is of course the result that the algorithm does converge for the cart pole system when using neural networks. Even though the algorithm only converges 25% of the



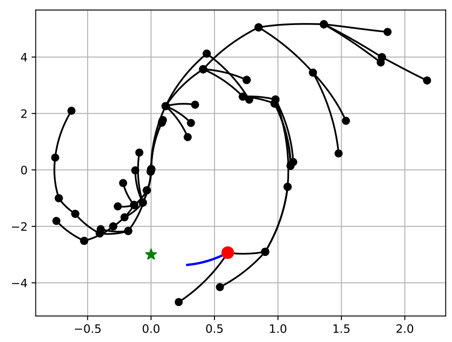
(a)



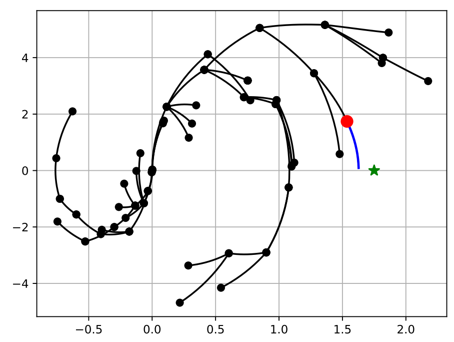
(b)



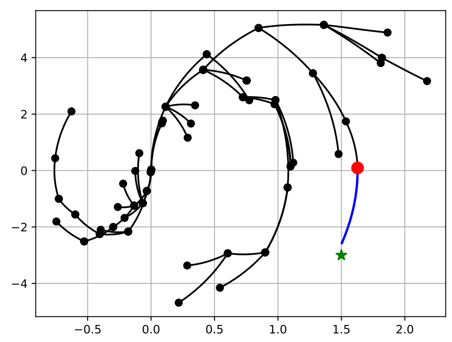
(c)



(d)



(e)



(f)

Figure 4-5: Building of a RRT for an underactuated single pendulum system. Green stars represent the random sample x_r and the red dots mark the node selected for expension. The selected nodes are not the nearest in terms of euclidean distance, but they are nodes that could actually reach the random sample.

attempts, this is a large improvement over RRT CoLearn which did not converge at all. Furthermore it does so at a reasonable time of 67.659 seconds. Since the algorithm performs much better with neural networks than with LWPR, from now on only neural networks will be considered.
The algorithm can be further improved however. Using the current form of data generation, all trajectories are of equal length. The next section describes how trajectories of variable length can be used.

4-4 Variable Time

In the previous section a new steering function was proposed which is based on learning inverse dynamics. As opposed to RRT CoLearn, IDL does converge when it is applied on a cart pole system. However, it only did so 25% of the time. To improve this, the generation of the trajectories on which the approximators are trained is slightly changed. Instead of using trajectories of fixed time length, the trajectories in the dataset now can vary in time length. Doing so should improve the dataset's coverage of the trajectory space, which is expected to improve the converage rate and time of the full RRT algorithm. This section describes the implementation of using variable time trajectories and the results of using it in RRT.

4-4-1 Data Generation

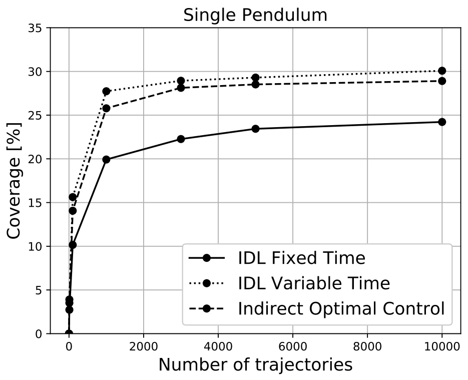
The main difference with the method of the previous section is the data generation. The generation is done similar to how it is described in Section 4-1, with a minor difference. Again, first a random initial state $x_i \in \mathcal{X}$ and a random control input $u \in \mathcal{U}$ are drawn. However, instead of integrating the equations of motion for a single time step of Δt seconds, they are now integrated for a number of time steps, $k_{max} \cdot \Delta t$. A single generated trajectory then leads to k_{max} datapoints which is stored as shown in Equation 4-1.

$$\mathcal{D} = \begin{bmatrix} x_i & x_{f1} & u & 1 \\ x_i & x_{f2} & u & k \\ \vdots & \vdots & \vdots & \vdots \\ x_i & x_{fn-1} & u & n-1 \\ x_i & x_{fn} & u & n \end{bmatrix} \tag{4-1}$$

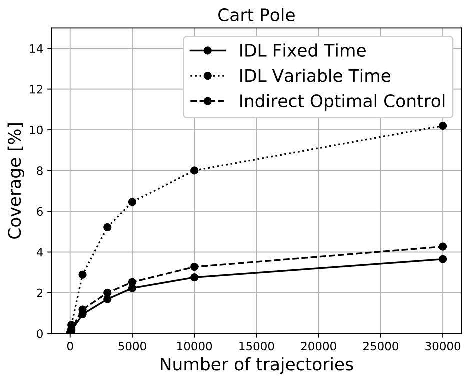
A line in the dataset should be read as: to get from x_i to x_{fk}, the control input u should be integrated for $t = k \cdot \Delta t$ seconds. This way longer trajectories are present in the dataset as well, without increasing the resolution. Hence using variable time trajectories should greatly improve the coverage of the trajectory space.

4-4-2 Dataset Coverage

A new form of data generation was presented which uses variable time trajectories, allowing longer trajectories to be present in the dataset. This form of data generation should improve the coverage of the trajectory space. To assess whether this is the case, the experiment from 4-1-1 is repeated with a dataset generated using variable time trajectories. Each dimension of the trajectory space is split up into four bins, splitting the entire trajectory space up into multiple boxes. The coverage of the trajectory space is then defined as the percentage of boxes that have datapoints in them. Figure 4-6 shows the dataset coverage for the single pendulum and cart pole system, with a dataset generated using indirect optimal control, IDL with a fixed $\Delta t = 0.1$ and IDL with variable trajectory length with $\Delta t = 0.1$ and $k_{max} = 5$. the



(a)



(b)

Figure 4-6: Coverage of the trajectory space for datasets generated using indirect optimal control, IDL with a fixed integration time and IDL with a variable integration time. Datasets generated with IDL with a variable integration time lead to a much higher coverage of the trajectory space.

dataset with variable time trajectories indeed covers a much larger portion of the trajectory space than when using a fixed time step. It actually outperforms the indirect optimal control method, which is proof that the trajectories in this dataset indeed travel into more different directions than when using indirect optimal control, since the coverage is higher while the trajectories are still four times shorter.

Using variable time trajectories in the approximated steering function leads a different mapping that needs to be approximated. Instead of $f : \{x_i, x_f\} \mapsto u$ the mapping now reads $f : \{x_i, x_f\} \mapsto \{u, k\}$. Since k is a discrete value, a different type of neural network can be used.

4-4-3 Classification of k

Since the value k that needs to be approximated is discrete, a classifier fits better in this task then a standard function approximator. The classifier is constructed by replacing the activation function in the output layer of a neural network with the softmax function of Equation 2-42. Before it is implemented in the full steering function, it is tested whether the value k can be approximated. Furthermore the approximation of u is reconsidered as well, since this approximation is expected to have become slightly more difficult due to the longer trajectories.
To assess whether the control input u can still be approximated accurately and whether k can be classified correctly, the experiment of Section 3-3 is repeated. Both the classifier and approximator network will be trained using different network shapes, and the accuracy of the

model will be tested using a test set. Three things will be measured: The approximation error $MSE(u, \hat{u})$, the steering error $MSE(x_f, x_r)$ and a classification score for k. The classification score will be the percentage of correctly classifed values. The classification of k will first be tested on true values of u, hence it will be used directly from the dataset. The equations of motion of the cart pole system are used for data generation, and the training set contains 30000 trajectories, generated with $\Delta t = 0.1$s and $k_{max} = 5$. The results are shown in Table 4-5. Both the control input u and the required number of time steps k can be approximated very accurately using neural networks. Even with the smallest network of (1×16), the classifier predicts the value of k correctly 94.7% of the times. The approximation error of the control input also remains small, even though the data has become more difficult to learn. The best result was achieved with the largest network shape tested, (5×64), with an estimation error of $MSE(u, \hat{u}) = 11.998 \cdot 10^{-3}$. The same holds for the time step classification with a score of 97.13%.

	$MSE(u, \hat{u})$ [$\times 10^{-3}$]	Score k [%]
(1×16)	113.379 ± 13.799	94.68 ± 0.915
(3×16)	83.908 ± 15.347	96.17 ± 1.103
(5×16)	37.444 ± 10.741	93.80 ± 1.348
(1×32)	74.160 ± 10.160	96.04 ± 1.177
(3×32)	28.365 ± 13.194	96.17 ± 0.857
(5×32)	21.811 ± 8.896	96.48 ± 1.047
(1×64)	68.319 ± 6.851	96.89 ± 0.791
(3×64)	12.614 ± 5.519	97.10 ± 1.234
(5×64)	11.998 ± 2.335	97.13 ± 1.051

Table 4-5: Estimation error for the control input u and prediction score for the required time steps k when using different shapes of neural network. The best performing shape is the largest one of (5×64), however the difference with the (3×64) network is minimal. Even the small networks predict the correct value 94% of the times. The approximation of the control input is indeed more difficult, since all approximation are higher than in Table 4-1.

4-4-4 Full Steering Function

As a final test, the full steering function will be considered. The full steering function is schematically shown in Figure 4-7. The two networks are combined, connecting the output of the control input approximator with the input of the time step classifier after being clipped such that the approximated $u \in \mathcal{U}$. To find out whether the new steering function works, the same experiment as in 4-2 will be repeated. Both neural networks will be trained on a dataset containing 30000 trajectories, generated by integrating the equations of motion with $\Delta t = 0.1$s and $k_{max} = 5$. and will be tested on a testset of 1000 trajectories. Note that instead of selecting the best performing network shapes, the second best is chosen for the classification of k. The difference in score is negligible, however the training of the network was found to be much faster when using the smaller shape of (3×64). The steering function is scored by its steering error, $MSE(x_f, x_r)$. The prediction score for k is also measured, since those predictions are now no longer computed using the true value of u, but with its estimated value $\hat{u}$. The entire experiment is repeated 10 times.

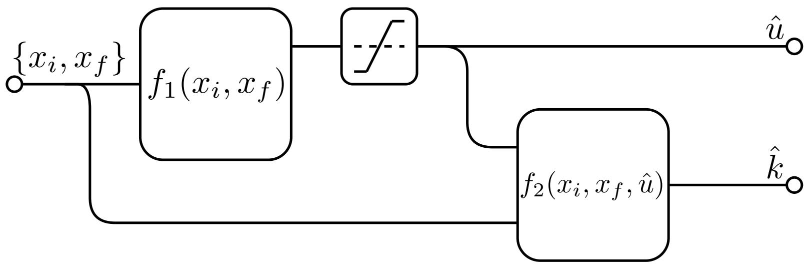


Figure 4-7: Structure of the steering function that makes use of learning inverse dynamics. The mappings f_1 and f_2 are implemented using neural networks.

	$MSE(u, \hat{u})$ [$\times 10^{-3}$]	Score k [%]	$MSE(x_f, x_r)$ [$\times 10^{-3}$]
Single Pendulum	3.064 ± 0.278	98.28 ± 0.214	0.867 ± 0.106
Cart Pole	12.821 ± 4.307	97.85 ± 0.559	59.181 ± 11.267

Table 4-6: Steering error for the full steering function when using variable time trajectories. Even though the steering error is higher than with fixed time trajectories, it still is marginal for both the single pendulum and the cart pole. Hence the steering function works properly with variable time trajectories as well.

The results are shown in Table 4-6. In case of the cart pole system, the steering error increased to $MSE(x_f, x_r) = 59.181 \cdot 10^{-3}$, higher than when using a fixed time step. This increase is expected since the training set and test set contain trajectories that are five times longer. Hence the distance travelled through state space is larger and thus the error is also expected to be slightly larger. Furthermore the data that the networks are trained on has become more complex than with a fixed time and thus more difficult to learn. Nevertheless the error is still negligible small and thus the steering function is assumed to work properly.

4-5 Variable Time Inverse Dynamics in RRT

In the previous section an improvement was proposed on the fixed time IDL steering function. Using variable time trajectories the dataset coverage greatly improved, while still being able to steer the system from some x_i to some other x_f. It is expected that this improvement will lead the RRT algorithm converge faster and more often. To find out if this is the case, the experiment outlined in Algorithm 7 is repeated with the new steering function. Apart from the two known systems, the single pendulum and the cart pole system, the algorithm is also applied on an underactuated version of the single pendulum. Its equations of motion are equal to that of the normal single pendulum, however the mass, rod length and gravity are set to $\{m, l, g\} = \{0.5, 0.5, 9.81\}$.

4-5-1 RRT Results

Table 4-7 shows the results after applying the new steering function on the planning problems. The underactuated single pendulum is marked with '(u.a.)'. In the following sections each system will be handled separately.

	Succes [%]	t_{plan} [s]	Number of Nodes
Single Pendulum	100	0.425 ± 0.400	111.7 ± 67.9
Single Pendulum (u.a.)	100	4.351 ± 6.613	536.2 ± 484.3
Cart Pole	100	16.409 ± 20.970	758.2 ± 725.9

Table 4-7: Results after applying the IDL steering function with variable time trajectories on several planning problems. Each problem converged 100% of the times, and much faster compared to fixed time IDL or RRT CoLearn.

The Single Pendulum

Running RRT with the steering function trained on variable time trajectories greatly sped up the planning of a swing up for the single pendulum. It converged 100% of the time with an average time of 0.425 seconds. Compared to RRT CoLearn this is much faster, since RRT CoLearn on average converged in 3.4 seconds. Hence, the approach using learned inverse dynamics is over six times faster than RRT CoLearn. Figure 4-8 shows a swing up trajectory plotted in state space for the single pendulum. Compared to fixed time trajectories, this approach also needs much less nodes in the tree, on average a tree consisted of only 111.7 nodes, which is about six times less. Most likely this is due to the longer trajectories that are learned, leading to longer tree branches as well.

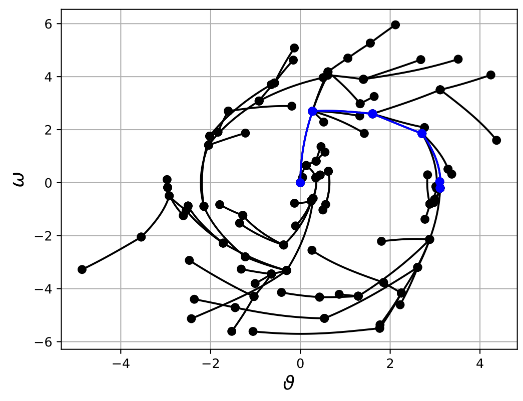


Figure 4-8: A swingup of the single pendulum plotted in state space. The tree built to found this trajectory is plotted in black, the path from x_i to x_{goal} is plotted in blue.

Cart Pole

Applying RRT with the new steering function to the cart pole system also greatly improved the performance, the algorithm converged 100% of the times as well. The average computation time was found to be only 16.409 seconds, compared to 67.659 seconds when using fixed time trajectories. Compared to RRT CoLearn this is a major improvement, since with RRT CoLearn the system did not converge at all. Figure 4-9 shows a swing up trajectory plotted in statespace for the cart pole system.

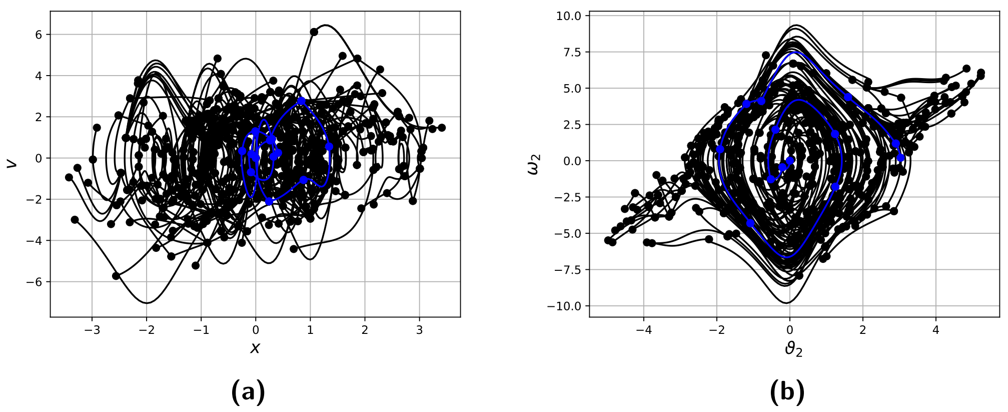


Figure 4-9: A swingup of the cart pole system plotted in state space. The tree built to found this trajectory is plotted in black, the path from x_i to x_{goal} is plotted in blue.

Underactuated Single Pendulum

Figure 4-10 shows a swing up trajectory plotted in statespace for the underactuated single pendulum. The required back and forth swings show up in the trajectory as the outwards spiraling motion. Planning a trajectory for an underactuated pendulum took a bit longer than for the fully actuated pendulum, which is expected since the trajectory is more complex and thus needs more nodes. The average computing time was 4.351 seconds and it required 536 nodes on average. It converged 100% of the attempts as well.

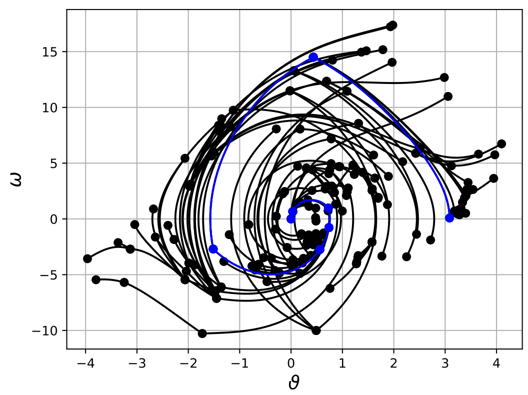


Figure 4-10: A swingup of the underactuated single pendulum plotted in state space. The tree built to found this trajectory is plotted in black, the path from x_i to x_{goal} is plotted in blue. Due to the underactuation it has to swing back and forth multiple times to gain momentum, which leads to a spiral in state space.

As shown by these three examples, the new steering function based on variable time trajectories led to a much lower convergence time and converging more often than when using fixed time trajectories. Each system converged 100% of the times, even the cart pole system. In comparison with RRT CoLearn, the new steering method lead to a 6 times faster convergence for the single pendulum, it went from $t = 3.4$s to $t = 0.425$s, but most importantly it also converges for multi-DOF systems, which did not happen with RRT CoLearn.

4-6 Summary

With the goal to increase the coverage of the trajectory space and simplify the relationship between the data that needs to be learned, a new form of data generation was investigated. Instead of using optimal control to create the trajectories in the dataset, the trajectories were simply generated by integrating the equations of motion for a short time Δt while a constant control input u was applied to the system. Section 4-1-1 showed that the coverage indeed increased for both the single pendulum and cart pole system, when equal trajectory lengths were used. This indicates that the trajectories in the dataset evolve into more different directions than when using indirect optimal control.
In Section 4-2-1 and 4-2-2 it was tested whether the relationship was simplified as well. A neural network and a LWPR model were trained on the generated dataset, in order to learn the mapping $f(\cdot) : \{x_i, x_f\} \mapsto u$, hence it tries to learn the *inverse dynamics*. Using the trained models it was tested how accurately they could learn the required control input and how accurate they could steer the system from some initial x_i to some other goal state x_f. It showed that the relationship was indeed much simpler to learn, since the approximation error of the required control input was only $MSE(u, \hat{u}) = 1.196 \times 10^{-3}$ for the cart pole system, which is very small. It also showed that the inverse dynamics were much better and faster approximated when using neural networks than when using LWPR. More important is the steering error, which was reduced to $MSE(x_f, x_r) = 0.264 \times 10^{-3}$, compared to $MSE(x_f, x_r) = 0.076$ when using RRT CoLearn. Hence the system can be steered much more accurately when learning the inverse dynamics. Since the new method learns the invesre dynamics, it was called '*Inverse Dynamics Learning*' (IDL).
An advantage of using optimal control instead of just random trajectories is the direct availability of a good distance metric, the optimal cost to go. Since the cost to go was no longer present, a different distance metric has been proposed: the estimated steering error. A second neural network is trained to learn the mapping $f_\varepsilon : \{x_i, x_f\} \mapsto \varepsilon$. The larger the expected error is, the further away the node is assumed to be. In other words, the node from which the steering function can most accurately reach x_r is assumed to be the nearest.
Since the distance metric was not the focus of this thesis, it has only been briefly tested. First it was tested whether it can be accurately approximated using a neural network which was indeed the case. Second it was applied in RRT to find whether the nodes it selected were sensible or not, hence if it took in account the dynamics of the system. It appeared to do so as is shown in Figure 4-5. The final test for the distance metric was its application in RRT. Since the planning attempts did converge it was assumed that the distance metric functioned properly. Using this distance metric and the steering function based on inverse dynamics, the RRT algorithm converged 95% of the time for the single pendulum and 76% of the time for the cart pole, both when using neural networks as approximators. The single pendulum converged within comparable time as RRT CoLearn, however the main difference is that the

cart pole system converged as well.
To further improve the coverage of the trajectory space, the data generation was improved further. Instead of using fixed length trajectories, the trajectories trained became of variable length. The equations of motion were integrated for $k_{max} \cdot \Delta t$ seconds, with k_{max} a positive integer. A single integration then leads to k_{max} trajectories. The steering function now needs to approximate the mapping $f : \{x_i, x_f\} \mapsto \{u, k\}$, hence it needs to learn what control input u needs to be applied for $k \cdot \Delta t$ seconds in order to steer the system from x_i to x_f. The implementation of the approximation of k was done using a neural network classifier, since k is discrete. Section 4-4 showed that the required number of time steps can be estimated very accurately and that even though the data is slightly more difficult to learn since the trajectories are longer and more trajectories from the same initial point are present in the datset, the steering function still steers with the same order of accuracy.
Finally the new steering function was applied in the full RRT algorithm. The single pendulum has been considered twice, once in the same configuration as used in the RRT CoLearn publication, and once such that it was underactuated. Using the variable time trajectories, the algorithm converged 100% of the time for both the single pendulum and the cart pole system. The planning time for the single pendulum was reduced to an average of 0.534 seconds, the cart pole on average converged in 16.409 seconds. All results of the final experiment are shown in Table 4-7.
In conclusion, to answer the question '*How can supervised learning be used to speed up kinodynamic planning?*', a new steering function is proposed which makes use of neural networks that greatly increases the computation speed of the swing up of a single pendulum compared to RRT CoLearn, more than 6 times faster. On top of that the RRT also converges for the 2-DOF cart pole system when using the learned inverse dynamics, as opposed to RRT CoLearn.

Chapter 5

Conclusion & Recommendations

This thesis started of with an investigation into RRT CoLearn to find out whether the claims of the authors of [14] would hold for other systems than the single pendulum. The complex mapping that is approximated in RRT CoLearn turned out not to work at all on other systems, leading to a broader research investigating the application of supervised learning in RRT. This lead to a new steering function based on learning the inverse dynamics. This chapter contains conclusions for this research and recommendations for future work.

5-1 RRT CoLearn

In [14] a new RRT method was proposed based on learning an input parameterization from trajectories. The trajectories were generated using indirect optimal control and the input was parameterized by the initial costates λ needed by indirect optimal control and the corresponding cost-to-go c. A dataset was then created by integrating the optimal equations of motion for some time t_{max} with time step Δt. A kNN function approximator was then used to learn the mapping $f : \{x_i, x_f\} \mapsto \{\lambda, c\}$. The method was applied on a single pendulum swing up problem leading to an average planning time of 2.4s. In order to rule out any differences in the research which could be caused by hardware or implementation differences, a baseline analysis was done which led to an average planning time of 3.323s, which is slightly slower. RRT CoLearn showed to be a promising solution for speeding up kinodynamic planning, however it has not yet succesfully been applied on other systems than the single pendulum, leading to the research question '*What are the consequences for RRT CoLearn when it is applied to a 2-DOF System?*'.
In order to answer this question, the algorithm was applied on a swing up problem for a cart pole system. The derivation of the equations of motion and an illustration of the system can be found in Section 3-1-3. The implementation had a minor difference with the implementation in [14], where the initial costates are again parameterized, such that only a single parameter had to be learned. This could not be done with the cart pole system, therefore the initial costates were learned directly. Applying RRT CoLearn *as is* did not result in

convergence to a solution. Trying to get the algorithm to converge, many adjustments were made:

- Larger datasets, up to 600 million datapoints, to rule out the curse of dimensionality.
- Different resolutions in data generation, between $\Delta t = 0.001$s and $\Delta t = 0.1$s.
- 1-NN classifier instead of 3-NN, to remove the need and influence of cleaning.

None of these adjustments, nor any combination of them led to convergence of the algorithm. Looking at the approximation errors it was expected that the inability to converge was caused by the steering function, since the approximation of the cost was only $MSE(c, \hat{c}) = 0.110$. However, when looking at the steering error, the mean squared error between the goal state and reached state, it turned out the steering function was not the problem. Using a large dataset of 100000 trajectories the steering error was reduced to $MSE(x_f, x_r) = 0.076$. Hence the steering function worked properly and thus could not be the cause of RRT CoLearn not converging.
The second suspicion was that the data generated using indirect optimal control lead to a directional bias. This means that the trajectories present in the dataset all propagate in a certain direction, instead of into all directions. This would cause the tree to build only into those directions, instead of into all directions. To test if this is the case, the coverage of the trajectory space was measured. Each dimension of the space was split up into bins, leading to a grid-like division of the trajectory space into boxes. The datapoints in the dataset were sorted in these boxes, and the percentage of filled boxes was defined as the space coverage. The datasets based on the single pendulum lead to a trajectory space coverage of 28% with 3000 trajectories, whereas the cart pole dataset only reached 4.2% with 30000 trajectories. This large difference confirms the suspicion of the directional bias, and led to research into different forms of data generation.

Apart from the attempts of making RRT CoLearn work on a 2-DOF system, an investigation was done into what parts of the algorithm took up most of the computation time. It was found that the largest chunk of time was consumed by the approximation of the cost to go, of which the computation time grew together with the size of the tree. Using a different, faster function approximator could greatly speed up the building of the tree and thus kinodynamic planning.
Two different function approximators have been implemented in order to find out if they could replace the kNN approximator. A Locally Weighted Projection Regression (LWPR)[29] model and a neural network were trained to learn the mapping $f : \{x_i, x_f\} \mapsto \{c, \lambda\}$, which were then used to test how good they could approximate the mapping, and how well they performed in the steering function. It turned out that the mapping was extremely non-linear, making it very difficult to approximate using a generalizing method such as LWPR or neural networks. The lowest steering error was achieved when using a (5×128) neural network, with $MSE(x_f, x_r) = 0.202$.
Using a neural network did not lead to convergence of the algorithm, but it was shown that using a neural network instead of kNN would indeed speed up the algorithm. Building a tree of only 500 nodes was done about 1.5s faster, a decrease in computation time of almost 10%. However, to be able to use a neural network, the relationship between the trajectories and

the control input parameterization needs to be simplified since the mapping from trajectory to initial costates is too non-linear to be approximated using neural networks. Hence this investigation also led to research into different forms of data generation.

With two motivations to research different forms of data generation, the use of Hamiltonian formalism in deriving the equations of motion was investigated. It was found that when using the cost function $C(x, u, t) = w + \frac{1}{2}u^T W u$, Hamiltonian formalism leads to an input parameterization equal to the negative costates corresponding to the momenta of the actuated joints, $u^* = -\lambda_p$. It was expected that using this in the data generation would simplify the mapping that needs to be approximated, allowing neural networks to be used as function approximator in the steering function.
The approximation of the mapping did indeed slightly improve compared to when using Lagrangiang formalism, a steering error of $MSE(x_f, x_r) = 0.165$ was achieved. However, this error was expected to be too high to be useable in RRT. Furthermore since it is still generated using indirect optimal control, it still has the problem of directional bias explained earlier.
Finally a different form of optimal control was considered for the data generation, direct optimal control. A single datapoint in a dataset created using direct optimal control would contain an initial state x_i and final state x_f, and the optimal control input u^* needed to travel from x_i to x_f. To generate these datapoints using direct optimal control turned out to be difficult due to its need for a good initial guess for the optimizer. However, the dataset that would be created quickly inspired to research approximating the inverse dynamics which would lead to an almost equal dataset, and therefore direct optimal control was not further investigated.

The answer the question '*What are the consequences for RRT CoLearn when it is applied to a 2-DOF System?*' was found to be that it does not converge due to a directionial bias introduced by using indirect optimal control. A different form of data generation could solve this problem, by straying from optimal control and approximating the inverse dynamics of a system.

5-2 Inverse Dynamics Learning

The limitiaons of RRT CoLearn reported in the previous sections led to a broader investigation in how to apply supervised learning in RRT, with the research goal of successfully finding a swingup trajectory for the cart pole system using supervised learning in RRT. This led to a new steering function based on learning the inverse dynamics of a system, called *Inverse Dynamics Learning* (IDL).
In RRT CoLearn indirect optimal control is used for data generation. Optimal control is discarded in IDL and the trajectories in the dataset are now generated using random selected control inputs. The dataset is generated by integrating the equations of motion using a randomly drawn fixed control input u for a fixed time Δt. First of all this should lead to a better coverage of the trajectory space, since the directional bias caused by indirect optimal control is no longer present. Second, the mapping between the trajectories and the control input is much less non-linear and should be easier to learn by a generalizing function approximator such as a neural network or LWPR, which could greatly reduce the building time of the tree.

It indeed turned out that using IDL led to a higher trajectory space coverage, for the cart pole system it increased to 11% when using 2s trajectories. However, short trajectories are preferred since each branch in the tree will inherit the length of the trajectories. Using trajectories of 0.1s led to a coverage of of 3.6%, slighty lower than when using indirect optimal control. However, the trajectories generated using IDL are expected to traverse into more directions due to the lack of directional bias, which is confirmed by the coverage being almost equal with 20 times shorter trajectories.
To test whether the new data could be used as a steering function, the mapping from trajectory to control input $f : \{x_i, x_f\} \mapsto u$ was approximated using LWPR and a neural network. A dataset was created for the cart pole system using $\Delta t = 0.1$s, containing 30000 trajectories. Using this dataset the accuracy of the approximation was tested. The best results were achieved by using a neural network, with a steering error of $MSE(x_f, x_r) = 0.264 \cdot 10^{-3}$ when using a (5×32) network versus $MSE(x_f, x_r) = 7.929 \cdot 10^{-3}$ for LWPR. Note that these errors are remarkably low compared to the RRT CoLearn steering function, which had a steering error of $MSE(x_f, x_r) = 0.076$. This confirms that the mapping is easier to learn than the mapping that is used in RRT CoLearn.
Before the new IDL steering function can be tested in RRT, a new distance metric had to be constructed. Since optimal control is no longer used, the cost to go is no longer available. The expected steering error was proposed as a new distance metric. Steering a system from one state to another always results in some error. The node with the lowest expected steering error is assumed to be nearest, since steering from that node gets the system closest to the random sample. Computing the distance metric exactly is a time consuming process, its value is approximated using neural network as well. Using a network of (5×32) the approximation error was found be $MSE(\varepsilon, \hat{\varepsilon}) = 0.020 \cdot 10^{-3}$ for the cart pole system, hence it can be approximated very accurately. Finally the distance metric was shown to work empirically, by looking at which node would be selected for expansion while building a tree.
The new steering function and distance metric were then implemented in the RRT algorithm and applied to the single pendulum and cart pole swing up problems. Using neural networks the single pendulum swingup converged 95% of the time with an average planning time of 3.480 seconds. The cart pole swing up problem was solved 25% of the time with an average planning time of 67.659 seconds. In comparison with RRT CoLearn the method performs slightly worse on the single pendulum, since RRT CoLearn converged 100% of the time with an average planning time of 3.323 seconds. However, the IDL approach does work on the cart pole system, as opposed to RRT CoLearn, which is an improvement.
To improve performance of the steering function the data generation has been changed. Instead of trajectories with a fixed length, the function approximators are now trained on trajectories with variable length. The dataset is created by integrating the equations of motion for $k \cdot \Delta t$ seconds, with $k \in \{1, 2, \cdots, k_{max}\}$. Hence each integration leads to k_{max} datapoints. This causes the mapping that needs to be approximated as well, the function approximator now has to learn the mapping $f : \{x_i, x_f\} \mapsto \{u, k\}$. The mapping from the trajectories to u remains equal, the mapping from the trajectories to k is implemented using a classification neural network, since the value of k is discrete. A dataset based on the cart pole system was created using $\Delta t = 0.1$s and $k_{max} = 5$, on which the approximators were trained. It was shown that the mapping from $\{x_i, x_f\}$ to u has become slightly more difficult to approximate, with an average approximation error of $MSE(u, \hat{u}) = 11.998 \cdot 10^{-3}$, compared to $MSE(u, \hat{u}) = 1.196 \cdot 10^{-3}$ when using fixed time trajectories. The approximation error is 10 times higher, however it still is marginally small. The classification of k

was correct 97.13% of the time. Using the new approach a steering error was achieved of $MSE(x_f, x_r) = 59.181 \cdot 10^{-3}$. The steering error is also higher then when using fixed time trajectories. First of all because the mapping is more difficult to learn, but also because of the longer trajectories present. A longer trajectory means more distance is travelled through state space, and thus leads to a higher error as well. Still the steering error is much smaller than when using RRT CoLearn.
Applying the steering function in RRT on the swing up problems of the single pendulum and the cart pole system led to a large improvement. Both planning problems converged 100% of the times. The single pendulum did so on average within 0.425 seconds, almost 8 times faster than RRT CoLearn. The cart pole system converged within 16.409 seconds on average. The results are summarized in Table 5-1. The new steering function based on learning the inverse dynamics of a system performs much better than RRT CoLearn.

	RRT CoLearn		Fixed Time IDL		Variable Time IDL	
	Succes	t_{plan} [s]	Succes	t_{plan} [s]	Succes	t_{plan} [s]
Single Pendulum	100%	3.323	95%	3.480	100%	0.425
Cart Pole	0%	$---$	25%	67.659	100%	16.409

Table 5-1: Results for planning the single pendulum and cart pole swing up systems when using different types of steering function. RRT CoLearn did not converge on the cart pole at all. IDL with variable time trajectories converged every time with an average planning time of only 16.409 seconds due to its higher coverage of the trajectory space and the much simpler mapping that is learned by the neural network.

The research goal was achieved with a new steering function based on learning the inverse dynamics of the system, which was named *Inverse Dynamics Learning* (IDL). It was implemented using two neural networks and achieved much better results compared to RRT CoLearn, it converged to a solution faster on the single pendulum system and on the cart pole as well.

5-3 Recommendations for Future Research

A new steering function called IDL has been proposed which shows promising results in speeding up kinodynamic problem. However, so far it is only a proof of concept on an example system. Further research is needed to find out how useable the method really is, and there are some points on which IDL could be improved. Below is a list of recommended research directions and improvements.

- So far IDL has only been tested on a 2-DOF system. Further research is needed to find out whether the speed gains also hold for more complex systems, such as industrial manipulators. The dynamics of these systems are much more complex, making them more difficult to approximate. Furthermore these systems contain multiple control inputs, something that has not yet been tested. Other interesting systems could be non-holonomic systems such as self driving cars or UAVs.

- In Chapter 3, direct optimal control was briefly investigated as a possible method for trajectory generation. Due to the inspiration for and good results of IDL it was quickly discarded. However, it still is expected to be a good method, for three reasons. First, it will not have the directional bias indirect optimal control has. Second, it will supply a cost-to-go which could then be used as distance metric instead of the estimated steering error used in IDL. Finally it is expected that the quality of the data improves. In the current dataset there could be multiple equal trajectories with a different control input parameterization. For example, starting on x_i, the state x_f could be reached by applying $u = k$ for $t = 2 \cdot \Delta t$ seconds, but also by applying $u = 2 \cdot k$ for $t = \Delta t$ seconds. Using optimal control these double entries could be removed by picking the one with minimal cost, improving the quality of the dataset.

- Literature suggests the optimal cost-to-go to be the best possible distance metric, but IDL currently uses the estimated distance metric. Possibly a further decrease in computation time could be achieved by actually using the optimal cost to go. When using a different distance metric, the steering function can also be better compared to other steering functions than it can now. Since then the effect of the steering function can be isolated.

- In the RRT experiments in Chapter 4, the sampling of the state space is done uniformly. Deterministic sampling such as in RG-RRT could greatly improve convergence speed, since then the steering function will work much more accurately. Only trajectories can then be sampled that the function approximators are trained on, leading to a much lower approximation and thus steering error.

- A very interesting improvement could be to train the steering function as a whole, instead of training the function approximators separately. Instead of minimizing the estimation error, the steering error $MSE(x_f, x_r)$ can then be minimized. Hence the steering function will then be trained and penalized on its actual function instead of underlying functions.

Bibliography

[1] IFR, "Executive Summary of World Robotics 2016 Service Robots," 2016.

[2] E. S. Ruiz and F. M. U. R. Perez, "Robotics, the new industrial revolution," *IEEE technology and society magazine*, no. June, pp. 51–58, 2012.

[3] R. I. Association, "North American Robotics Market Surges 32 Percent in Unit Volume," 2017.

[4] S. M. LaValle, *Planning Algorithms.* Cambridge: Cambridge University Press, 2006.

[5] S. M. LaValle, "Rapidly-Exploring Random Trees: A new Tool for Path Planning," tech. rep., Department of Computer Science, Iowa State University, Ames, IA, 1998.

[6] S. M. LaValle and J. J. Kuffner Jr., "Randomized Kinodynamic Planning," *The International Journal of Robotics Research*, vol. 20, no. 5, pp. 378–400, 2001.

[7] M. Bharatheesha, W. Caarls, W. J. Wolfslag, and M. Wisse, "Distance Metric Approximation for State-Space RRTs using Supervised Learning," in *International Conference on Intelligent Robots and Systems (IROS)*, (Chicago, IL), pp. 252–257, 2014.

[8] L. Palmieri and K. O. Arras, "Distance Metric Learning for RRT-Based Motion Planning for Wheeled Mobile Robots," in *International Conference on Robotics and Automation*, (Seattle, WA), pp. 637–643, 2015.

[9] A. Shkolnik, M. Walter, and R. Tedrake, "Reachability-guided sampling for planning under differential constraints," in *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 2859–2865, 2009.

[10] R. E. Allen, A. A. Clark, J. A. Starek, and M. Pavone, "A Machine Learning Approach for real-Time Reachability Analysis," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2202–2208, IEEE, sep 2014.

[11] J. González-Quijano, M. Abderrahim, F. Fernández, and C. Bensalah, "A kinodynamic planning-learning algorithm for complex robot motor control," *2012 IEEE Conference*

on Evolving and Adaptive Intelligent Systems, EAIS 2012 - Proceedings, no. May 2012, pp. 80–83, 2012.

[12] R. Caruana and A. Niculescu-Mizil, "An empirical comparison of supervised learning algorithms," *Proceedings of the 23rd international conference on Machine learning*, vol. C, no. 1, pp. 161–168, 2006.

[13] V. Cherkassky, D. Gehring, and F. Mulier, "Comparison of adaptive methods for function estimation from samples," *IEEE Transactions on Neural Networks*, vol. 7, pp. 969–984, jul 1996.

[14] W. J. Wolfslag, M. Bharatheesha, T. M. Moerland, and M. Wisse, "Learning indirect optimal control for dynamic motion planning with RRT," in *Conference on Dynamic Walking*, (Stockholm), pp. 1–2, 2017.

[15] P. C. P. Cheng and S. LaValle, "Reducing metric sensitivity in randomized trajectory design," *Proceedings 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems.*, vol. 1, pp. 43–48, 2001.

[16] Y. Li and K. E. Bekris, "Learning approximate cost-to-go metrics to improve sampling-based motion planning," in *2011 IEEE International Conference on Robotics and Automation*, pp. 4196–4201, IEEE, may 2011.

[17] J. Bialkowski, S. Karaman, M. Otte, and E. Frazzoli, "Efficient collision checking in sampling-based motion planning," *Springer Tracts in Advanced Robotics*, vol. 86, pp. 365–380, 2013.

[18] J. Kuffner and S. M. LaValle, "RRT-connect: An efficient approach to single-query path planning," *Proc. IEEE International Conference on Robotics and Automation ICRA '00*, vol. 2, no. Icra, pp. 995—-1001 vol.2, 2000.

[19] S. A. Jacobs, N. Stradford, C. Rodriguez, S. Thomas, and N. M. Amato, "A scalable distributed RRT for motion planning," *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 5088–5095, 2013.

[20] C. Rodriguez, J. Denny, S. A. Jacobs, S. Thomas, and N. M. Amato, "Blind RRT: A probabilistically complete distributed RRT," *IEEE International Conference on Intelligent Robots and Systems*, pp. 1758–1765, 2013.

[21] R. Geraerts and M. Overmars, "Sampling techniques for probabilistic roadmap planners," *Proceedings International Conference on Intelligent Autonomous Systems*, pp. 600–609, 2004.

[22] A. Yershova, L. Jaillet, T. Siméon, and S. M. LaValle, "Dynamic-domain RRTs: Efficient exploration by controlling the sampling domain," *Proceedings - IEEE International Conference on Robotics and Automation*, vol. 2005, pp. 3856–3861, 2005.

[23] E. Glassman and R. Tedrake, "A quadratic regulator-based heuristic for rapidly exploring state space," *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 5021–5028, 2010.

[24] A. Perez, R. Platt, G. Konidaris, L. Kaelbling, and T. Lozano-Perez, "LQR-RRT * : Optimal Sampling-Based Motion Planning with Automatically Derived Extension Heuristics," *Icra*, no. 0, 2012.

[25] D. E. Kirk, *Optimal Control Theory - An Introduction.* Mineola, New York: Dover Publications, Inc., 13 ed., 2004.

[26] T. M. Cover and P. E. Hart, "Nearest Neighbor Pattern Classification," *IEEE Transactions on Information Theory*, vol. 13, no. 1, pp. 21–27, 1967.

[27] N. Bhatia, "Survey of Nearest Neighbor Techniques," *International Journal of Computer Science and Information Security (IJCSIS)*, vol. 8, no. 2, pp. 302–305, 2010.

[28] A. Hinneburg, C. C. Aggarwal, and D. A. Keim, "What is the nearest neighbor in high dimensional spaces?," in *Proceedings of the 26th International Conference on VLDB*, (Cairo, Egypt), pp. 506–515, 2000.

[29] S. Schaal, C. G. Atkeson, and S. Vijayakumar, "Scalable Techniques from Nonparametric Statistics for Real Time Robot Learning," *Applied Intelligence*, vol. 17, no. 1, pp. 49–60, 2002.

[30] S. Schaal and C. G. Atkeson, "Constructive incremental learning from only local information," *Neural computation*, vol. 10, no. 8, pp. 2047–84, 1998.

[31] S. Vijayakumar and S. Schaal, "Locally Weighted Projection Regression: An {O}(n) Algorithm for Incremental Real Time Learning in High Dimensional Space," *Proceedings of the Seventeenth International Conference on Machine Learning*, vol. 1086, pp. 1079–1086, 2000.

[32] S. Vijayakumar, A. D 'souza, and S. Schaal, "Incremental Online Learning in High Dimensions," *Neural Computation*, vol. 17, pp. 2602–2634, 2005.

[33] D. Nguyen-tuong, J. Peters, and M. Seeger, "Local Gaussian Process Regression for Real Time Online Model Learning and Control," *Nips*, pp. 1–8, 2008.

[34] D. N.-T. D. Nguyen-Tuong, M. Seeger, and J. Peters, "Computed torque control with nonparametric regression models," *2008 American Control Conference*, pp. 212–217, 2008.

[35] C. E. Rasmussen and C. K. I. Williams, *Gaussian processes for machine learning.* MIT Press, 2006.

[36] D. Nguyen-Tuong, B. Schoelkopf, and J. Peters, "Sparse Online Model Learning for Robot Control with Support Vector Regression," in *International Conference on Intelligent Robots and Systems*, (St Louis), pp. 3121–3126, 2009.

[37] R. Murugadoss and M. Ramakrishnan, "Universal Approximation of Nonlinear System Predictions in Sigmoid Activation Functions Using Artificial Neural Networks," in *2014 IEEE International Conference on Computational Intelligence and Computing Research*, pp. 1–6, 2014.

[38] F. Rosenblatt, "The Perceptron - A Perceiving and Recognizing Automaton," 1957.

[39] R. H. Hahnloser, R. Sarpeshkar, M. a. Mahowald, R. J. Douglas, and H. S. Seung, “Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit.,” *Nature*, vol. 405, no. 6789, pp. 947–951, 2000.

[40] Z. Zainuddin and P. Ong, “Function Approximation Using Artificial Neural Networks,” *International Journal of System Applications, Engineering & Development*, vol. 1, no. 4, pp. 173–178, 2007.

[41] G.-b. Huang, “Learning Capability and Storage Capacity of Two-Hidden-Layer Feedforward Networks,” *IEEE Transactions on Neural Networks*, vol. 14, no. 2, pp. 274–281, 2003.

[42] D. Stathakis, “How many hidden layers and nodes?,” *International Journal of Remote Sensing*, vol. 30, no. 8, pp. 2133–2147, 2009.

[43] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning Representations By Back-Propagating Errors,” *Nature*, vol. 323, pp. 533–536, 1986.

[44] D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” *Iclr*, pp. 1–15, 2015.

[45] G. Zhang, “Neural networks for classification: a survey,” *IEEE Transactions on Systems, Man and Cybernetics, Part C (Applications and Reviews)*, vol. 30, no. 4, pp. 451–462, 2000.

[46] A. Graves, M. Liwicki, S. Fernández, R. Bertolami, H. Bunke, and J. Schmidhuber, “A novel connectionist system for unconstrained handwriting recognition,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 31, no. 5, pp. 855–868, 2009.

[47] X. Li and X. Wu, “Constructing Long Short-Term Memory Based Deep Recurrent Neural Networks For Large Vocabulary Speech Recognition,” *Icassp*, pp. 4520–4524, 2015.

[48] W. Zhang, K. Itoh, J. Tanida, and Y. Ichioka, “Parallel distributed processing model with local space-invariant interconnections and its optical architecture.,” *Applied optics*, vol. 29, no. 32, pp. 4790–7, 1990.

[49] R. M. French, “Catastrophic forgetting in connectionist networks,” *Trends in Cognitive Sciences*, vol. 3, no. 4, pp. 128–135, 1999.

[50] G. Cybenko, “Approximation by Superpositions of a Sigmoidal Function,” *Mathematical Control Signals Systems*, vol. 2, pp. 303–314, 1989.

[51] K. Hornik, M. Stinchcombe, and H. White, “Multilayer Feedforward Networks are Universal Approximators,” *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989.

[52] P. Orponen, “Computational Complexity of Neural Networks: a Survey,” *Nordic Journal of Computing*, vol. 629, no. 2, pp. 50–61, 1994.

[53] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, G. Louppe, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and É. Duchesnay, “Scikit-learn: Machine Learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2012.

[54] S. Klanke, S. Vijayakumar, and S. Schaal, “A Library for Locally Weighted Projection Regression,” *Journal of Machine Learning Research*, vol. 9, no. 1, pp. 623–626, 2008.