Computational Science and Engineering (Int. Master's Program)

Technische Universität München

Master's Thesis

Automatic Heuristic Generation for Optimal Control

Author:	Erik Wannerberg
1st examiner:	Prof. Dr.-Ing. Klaus Diepold
2nd examiner:	Prof. Dr. Hans-Joachim Bungartz
Assistant advisors:	Dr. Dirk Hartmann
	Dominik Meyer
	Birgit Obst
	Dr. Hao Shen
Thesis handed in on:	July 14, 2017

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

August 30, 2017 Erik Wannerberg

Abstract

Control problems involving dynamical systems are ubiquitous in today's products and systems. A general tool to solve them optimally is *Model Predictive Control* (MPC). However, large amounts of resources and expert knowledge are required to realize MPC algorithms such that they can run in real-time on control hardware, giving the approach limited success. In this thesis, a novel method is proposed and implemented to make MPC and especially nonlinear MPC more viable, by using *diffusion maps*, a manifold learning method.

Inspired by previous successes approximating MPC optimization functionality with small sets of heuristics, the method works by finding low-dimensional nonlinear parametrizations of the dynamics of the controlled system, following a method by Dietrich et al. [1] called *closed observables*. The key idea is to generate large numbers of state trajectories and, through diffusion maps, find their low-dimensional coordinates. They are utilized to produce functions interpolating the dynamics of the calculated control. This allows one to create a controller similar to the heuristics-based ones, but fully automatically generated. As this can fulfil the real-time requirements even for a large, complicated nonlinear system, this opens the door for fully automated MPC integration, e.g. starting directly from engineering models, without requiring excessive expert knowledge.

In order to generate data for the method, a general nonlinear MPC framework is developed. We test this for a simplified nonlinear system of a vehicle driving energy-efficiently along a hilly road, using real-world height data. The proposed method is applied to the generated control data and a controller based on the parametrizations is created. The performance is compared to that of MPC. We conclude the thesis by looking at potential improvements to performance and usability.

Contents

Outline of the Thesis ix

I. Introduction and Theory 1

1. Introduction 3

2. Model Predictive Control 9

2.1. Control problem 9

2.1.1. Dynamical system 11

2.1.2. Optimal control problem 11

2.2. Model Predictive Control 12

2.2.1. Solution methods for the Model Predictive Control problem 12

2.2.2. Ipopt and `pyomo` 14

2.3. Model scenario 14

2.3.1. State and control variables 15

2.3.2. Physical laws and constraints 15

2.3.3. Costs and objectives 18

2.4. Distance reparametrization 19

2.4.1. Distance-parametrized system equations 20

2.5. External parameters and data 21

3. Learning Control Heuristics 23

3.1. Approximation of Model Predictive Control as a function 23

3.2. Representation learning 23

3.3. Diffusion maps 25

3.3.1. Method 27

3.3.2. For discrete datapoints 31

3.3.3. Repeated eigendirections 31

3.3.4. Diffusion maps for dynamical systems 31

3.3.5. Closed observables 32

3.4. Mathematical and computational methods 34

3.4.1. Parameter reduction 34

3.4.2. Sampling 34

3.4.3. Interpolation methods 35

3.4.4. Large eigenvalue computations 36

II. Methodology 39

4. Algorithm Overview 41
4.1. Schematic diagram . . . 41

5. Practical Model Predictive Control 45
5.1. Height data acquisition . . . 45
5.2. Model Predictive Control in `pyomo` . . . 46
5.3. Model Predictive Control code organization . . . 46
5.4. Parameter and method choices . . . 48

6. Diffusion Maps Framework 51
6.1. Overview of implementation . . . 51
6.2. Preprocessing . . . 51
6.3. Code structure . . . 53

7. Validation 55
7.1. Matlab implementation . . . 55

III. Conclusion 57

8. Results and Discussion 59
8.1. Model Predictive Control results . . . 59
8.2. Diffusion maps results and discussion . . . 59
8.3. Outlook on further work . . . 65
8.4. Conclusion . . . 65

Appendix 69

A. Parameter Values 69

B. Model Predictive Control Results 71

C. Source Code 75
C.1. Model Predictive Control . . . 75
C.2. Diffusion maps . . . 77
C.3. Closed observables and interpolation . . . 85
C.4. Validation . . . 86

Bibliography 89

Outline of the Thesis

Part I: Introduction and Theory

CHAPTER 1: INTRODUCTION

A broad overview of the current state of Model Predictive Control and other techniques for optimal control, along with the context of the motivation behind this thesis.

CHAPTER 2: MODEL PREDICTIVE CONTROL

A review of the theory underpinning Model Predictive Control, along with some background on the tools used.

CHAPTER 3: LEARNING CONTROL HEURISTICS

The theory behind diffusion maps, and how it is used to learn control heuristics.

Part II: Methodology

CHAPTER 4: ALGORITHM OVERVIEW

An introduction and overview for the overarching algorithmic process that is the results of this thesis.

CHAPTER 5: PRACTICAL MODEL PREDICTIVE CONTROL

The implementation and choice of methods for the Model Predictive Control problem.

CHAPTER 6: DIFFUSION MAPS FRAMEWORK

The specific implementation of the diffusion maps framework and the considerations there.

CHAPTER 7: VALIDATION

A presentation of the validation framework that was used.

Part III: Conclusion

CHAPTER 8: RESULTS AND DISCUSSION

A discussion of the results and outcomes of the thesis, as well as interesting outlooks for the future.

Part I.

Introduction and Theory

1. Introduction

Throughout the years, there have been many attempts to solve control problems optimally. Taking a formal point of view, they can be reduced to a mathematical optimization problem: **find *a control function* u such that u *minimizes cost* under condition of *these constraints***. This control function can take many forms — a set of parameters for a mechanical device, an electronic programmable signal, or an ideal setpoint for a system. This is then implemented in a control system.

As there are so many processes that use digital control, there is an enormous amount to gain from developing better controllers. For example, in most cases, economical gains can be made by steering machinery in factories to operate in a coordinated fashion and require less energy or produce more. Environmental gains may be significant when controlling processes such as chemical plants not to release harmful chemicals. Additionally, there might even be some types of processes that are not possible without using very complex control, be it moon landings, advanced chemical reactions or self-driving cars.

One general algorithm for solving these types of control problems involving dynamical systems — systems that evolve with time — is Model Predictive Control [2], also known as Receding Horizon Control. With Model Predictive Control, the problem is solved up to a set time horizon, and the state of the system is predicted within this interval using a model of the system, to find the control values that makes the system fulfil the constraints while minimizing the cost. The first part of this optimal control is then implemented, and the problem is again solved after observing how the system evolved during that time, shifting the interval that is solved for forward (the horizon "recedes"). Many different flavours of Model Predictive Control have been developed over the years, and are suitable for different types of applications, as can be seen in the comparison in table 1.1.

Setting up a mathematical optimisation problem is not always easy. First, we need to understand the controlled system in more or less depth, and build up the model for how it works. This also includes finding all the physical and non-physical parameters that govern the system. In many cases, this knowledge is however already made available from the design and engineering process. For using this knowledge to calculate the best way of controlling it, we may then additionally require large amounts of analytical and computational work. And when finally implementing the control system, there might be things that are impossible to take into account on beforehand, such as discrepancies between the model and the actual system, external disturbances during operation, or goals and objectives that change in real-time. This is most often solved by performing *online optimization*, (re-)computing parts of the solution as it is running, mostly using *feedback* from the system.

This solution is very versatile, but requires computational power as the system is running, and that the solution is available fast enough to implement in real-time. If the optimization algorithm is complex or requires very many computations, we thus require very much computational power. This poses a problem, as we might not always have a super-

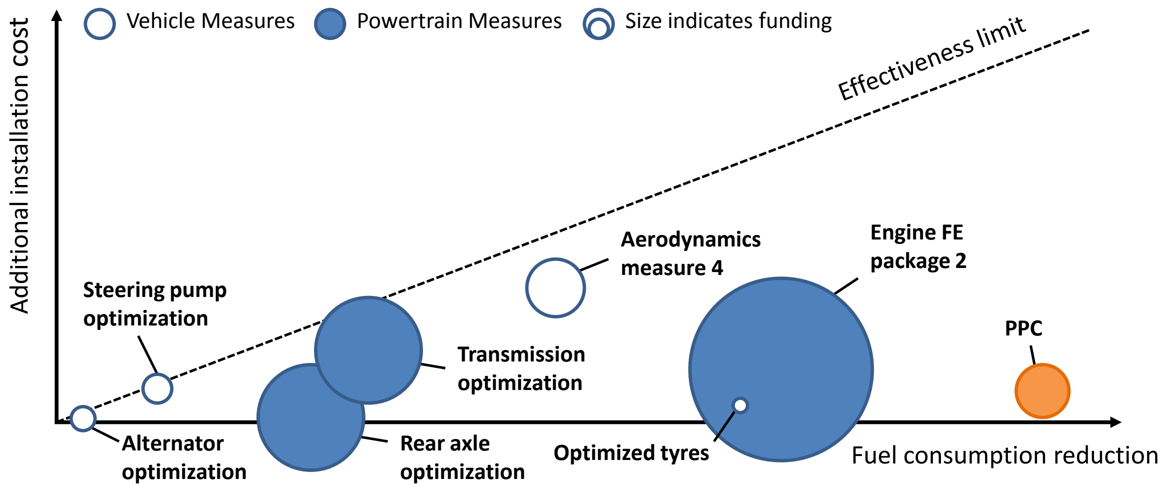


Figure 1.1.: Potential fuel savings from truck improvements, according to [3]. The automatic speed control *Predictive Powertrain Control* (PPC), can give large fuel savings for relatively small additional installation costs. A video explaining PPC can be found at https://www.youtube.com/watch?v=8Z62SCSaJhQ.

computer available nearby,[1] or the money required for one. Additionally, there needs to be someone that can convert the understanding into a working control system — an expert or a team of experts, which often is scarce at hand.

In today's industry, this problem is solved in different ways. In most cases, one uses a simpler controller, such as the industry standard PID-controller. For some systems, it may be sufficient to solve a simplified problem, for example by linearization. For example, linear Model Predictive Control has had enormous success in the industry over the last decades, and is used for many systems that are not linear, where it most often produces a suboptimal solution. Another solution is ignoring how the problem changes over time, by solving a static problem. Both of these approaches may be simple enough that a solution can be implemented on less advanced controllers. Often, no model is even required, as the parameters necessary for tuning the control can be readily identified from the system. This is another reason why these approaches are popular. An example which can be solved by PID-controllers is a car speedometer, or cruise control, where the controller increases or reduces motor throttle until the car has the desired speed.

A more involved approach to retain the advantages of the optimal solution is also to try and calculate the optimal solutions, then try and find patterns in them to replace them with something implementable. Although this requires extensive analysis and expert knowledge, it might still be worthwhile. In some cases, with few enough variables, linear Model Predictive Control can be approximated this way. The solution is then calculated explicitly — the name is therefore *explicit* Model Predictive Control — and is only calculated once, and is reused in each control step, which makes it incredibly fast.

Another approach, that has been tried by several truck companies have in the last years,

[1]Certainly true at least if one's standards with regards to supercomputers are not from the '90s, at which time the supercomputers were about as powerful as today's smartphones. Luckily for us, the supercomputer standards have improved since then.

	Linear MPC	Nonlinear MPC	Explicit MPC
Solution time	+	- -	++
Guaranteed solution	+	-	+
Complexity of cases	-	++	- -
Optimality	+	++	+
Model availability	+	- -	-

Table 1.1.: A comparison of some of the different flavours of Model Predictive Control (MPC).

involves approximate the optimal control patterns with simpler heuristics. There, they implemented *predictive cruise control*, where height data from the road ahead of the truck is used to calculate optimal speeds (and gear changes) for trucks. By analysing the optimal control strategies calculated off-line, fuel-saving patterns[2] have been identified and implemented using simpler control heuristics.[3] Thus, the knowledge of the optimal control can be used without having to calculate it during operation. This is a crucial factor for real-time control, where time or computational resources might not always be available. The numbers quoted in how much fuel this saves vary, but usually lie in the 3–6% range. As this is a very large saving for a relatively small implementational cost, the approach has gained in popularity, and is now implemented by a variety of truck manufacturers. A comparison on fuel savings and installation costs in different areas, which can be seen in figure 1.1, clearly shows the potential gains.

One approach to try to understand or model a system is using statistical methods and the relatives of them commonly referred to as *machine learning*, by the virtue of that patterns and implications in data are found using the power of machine algorithms. Machine learning is already used in several control applications:[4] both as a modelling tool, to produce a model of a system in order to simulate and choose optimal control for it, and as an optimisation tool, for example training a robot to coordinate its different leg actuators to take a step.

Often, machine learning is mentioned in the same breath as another popular term, *big data*, as the methods tend to rely on very large datasets. For the problem of control, this can pose a problem, as data might be hard to obtain. In some cases, measurements are needed where it is impossible or infeasible to place a sensor. For industry, the data might be unavailable intellectual property. In addition, during stages of rapid development, it might not be viable to wait a long time to gather enough data before the control can be used — the system might not even be built yet.

Fortunately, data might not always need to come from the real world. Using the design and engineering models from development of a product, many systems can be simulated

[2]Such as, decreasing the speed on the top of a hill and letting gravity do the work of regaining speed.

[3]Such as, "**if there is** *hilltop with enough downhill afterwards*, **then** *reduce gas before hilltop*".

[4]Other than the control strategies in trying to beat humans in all kinds of games; Google's AlphaGo that beat the reigning Go master was heavily trained using Reinforcement Learning, and research is currently being done in developing Artificial Intelligence by making computers understand computer games.

to generate data. The amount of simulations needed for the optimization tools of machine learning, often complicated and sometimes still not completely understood, might however still be inhibitory. Instead, with the models available, one can take a shortcut of using machine learning to approximate another model-based optimisation algorithm — such as Model Predictive Control.

In several cases, researchers have been able to accomplish exactly this: using the universal approximator property of *neural networks* to simulate the output of linearized Model Predictive Control, without needing to solve its equations [4, 5]. This provides a solution similar to explicit Model Predictive Control, but is more versatile, as it can be used in more cases. However, training a neural network is not always a simple task, and may produce varying results. Therefore, methods that are more predictable or easier to understand are sometimes preferred.

Diffusion maps [6] is another type of machine learning based on manifold learning, which uses the structure of data to build a good representation of it. By using a measure of similarity between points (e.g. distance), a type of diffusion operator is created. The key part of the method is that the eigenvectors of this diffusion operator form an intrinsic parametrization of the underlying geometry, analogously to how the eigenfunctions of a heat- or diffusion operator produce a basis for the geometry on which they operate. For data with very many dimensions, but lying along a lower-dimensional space in the high-dimensional space, diffusion happens only along the manifold, and therefore is oblivious to the extra data dimensions. By choosing the most relevant eigenvectors, one obtains a nonlinear low-dimensional approximation of the data. This approximation can be visualized and interpreted as the low-dimensional manifold of the data, or used directly as a basis for functions along the manifold.

As the state of a dynamical system moves along a low-dimensional manifold in a high-dimensional state space, diffusion maps are a good candidate to use to build up a model of a dynamical processes. The method of *closed observables* [1], additionally uses time-series of the state variables of interest, to build up interpolated functions for the dynamics. Thus, patterns in time and whole dynamic relationships can be encoded, making it particularly suitable for approximation of a dynamic state, or a function of it. We can therefore use closed observables to encode the effect of Model Predictive Control — going from state to the optimal controls.

With these low-dimensional representations of optimal control inputs, we may then finally construct a controller that utilizes them to control the system. This control then shares the benefits of Model Predictive Control's good performance, but is much simpler to calculate, as it is just a low-dimensional interpolation. This allows for using any model of the system, such as complex engineering models from production, that with the aid of for example Model Order Reduction [7] can be directly utilized for nonlinear Model Predictive Control.

To summarize; in this thesis, an alternative way of combining Model Predictive Control and machine learning is proposed, using diffusion maps. Using control data from offline simulations with Model Predictive Control in different scenarios for a system, a diffusion maps model for the control can be built. This is then used as the control strategy in a controller, that can provide control for a real system. This can give the performance and optimality of nonlinear Model Predictive Control, but at a fraction of the time, and provide an approximate solution where a solver might fail to converge. Then, using data collected

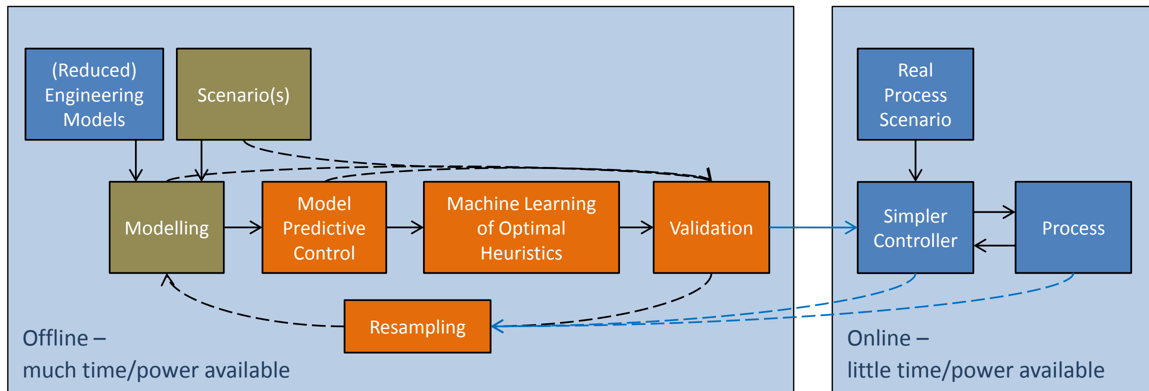


Figure 1.2.: A proposed control system for using Model Predictive Control to provide optimal control approximated by the machine learning method Diffusion Maps. Offline simulations provide data for approximation of optimal control. The control system can provide the control performance of nonlinear Model Predictive Control, without needing to solve complicated nonlinear optimization problems during the online phase. This allows real-time implementation of complex controllers.

in the real case, the model and the control can also be improved. The proposed system can be seen in figure 1.2.

Similar to above, the goal of this thesis is to describe and show the feasibility of approximating nonlinear Model Predictive Control. This will be using the machine learning methodology of diffusion maps with closed observables, to allow to automatically derive a controller with the same optimality, but with much lower computational effort in application. This will allow fully nonlinear Model Predictive Control to become an as useful and applicable industry tool as its linear counterpart, and possibly even more accessible.

2. Model Predictive Control

A control problem may be solved with many approaches, with the unifying goal that one wishes to manipulate a system in order to minimize a cost. Common examples include minimizing a difference between the system state and some ideal state, or a penalty on some system variable, or them both combined, but the cost could be any abstract function of the system. In an *open-loop* control scenario, we do this by calculating the best possible strategy beforehand, based on how we know the system reacts, and then executing it on our system. However, if there are some unknown disturbances or discrepancies in the model, this might lead to bad results. It is therefore advantageous to use the current state of the system for calculating the control response, for *feedback* or *closed-loop* control. For this, a well-known example is the PID-controller, which outputs a control signal that has parts that are Proportional to its input, and parts that scale with the Integral and the Derivative of the input as well. However, such simple controllers can mostly still only act *reactively*, since they only take into account information from the *past*. This limits the optimisation that could be done when also including known information about the future.

A *Model Predictive Controller* (MPC controller) on the other hand, contains a model of the controlled system, and uses this in optimisation to choose the best control strategy based both on the current state and the predicted future state when following the control. In the optimization, the optimizer thus chooses not only the very next control strategy to apply, but also during a whole *prediction horizon*. In that way, strategies that are better in the longer run are followed, and for example longer-term instabilities or infeasibilities may be completely avoided.

However, every model has uncertainties, which is why an MPC controller only executes the first part of its strategy, and then recomputes the optimal control based on newer measurements of the state. This allows the system to be flexible and react also to unexpected events, as long as the model is not too far from reality. This whole process is illustrated in figure 2.1.

Typically, to reduce computational efforts, the prediction horizon is chosen to have a finite length, as long as needed to give stability and predictive power, but as short as possible to reduce excess calculations. To deal with stability issues, a *terminal cost* term may also otherwise be added in the cost function, which represents the cost of instabilities or other effects after the end of the horizon.

2.1. Control problem

As the field of optimization is broad and well-studied, let us start with some relevant and necessary concepts and definitions.

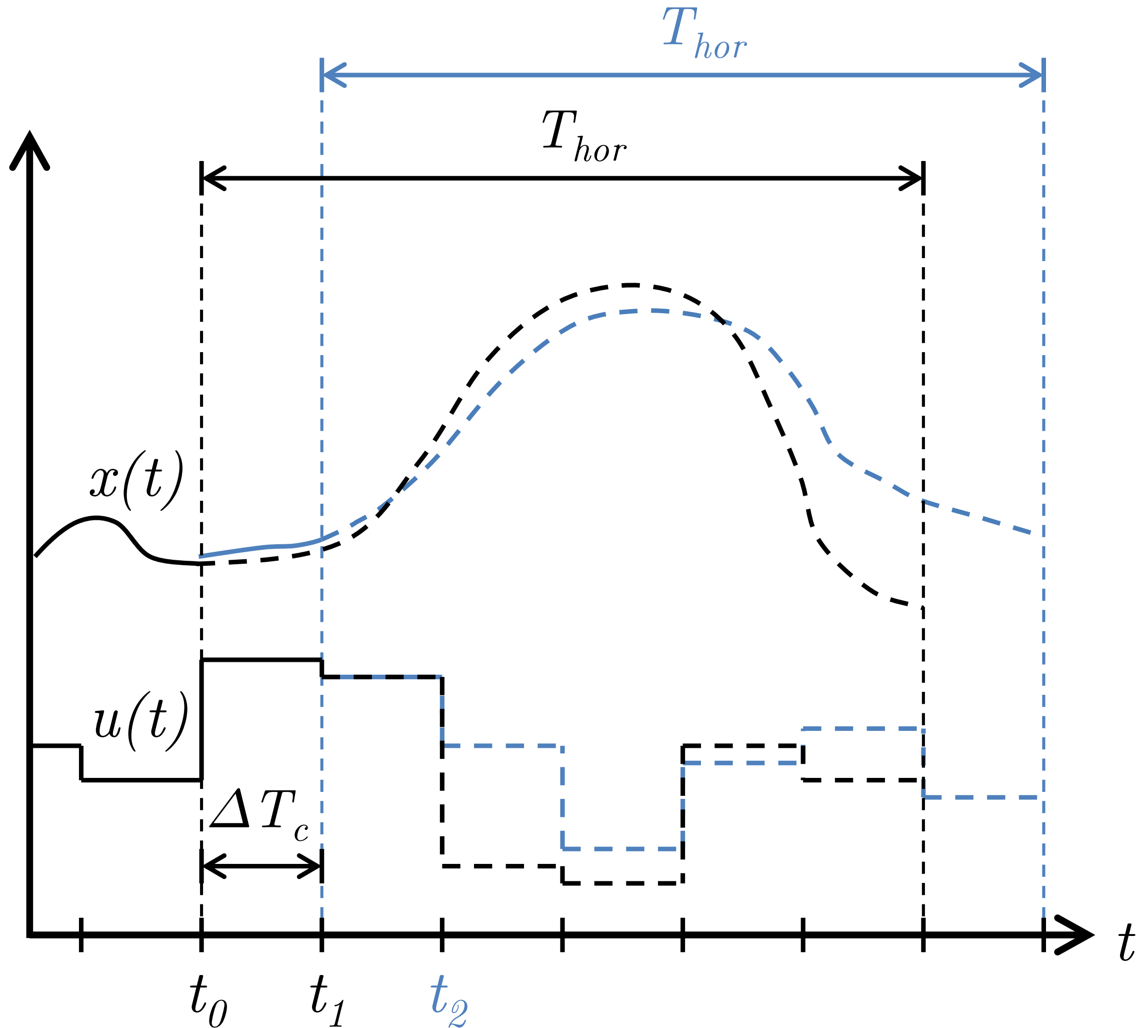


Figure 2.1.: An illustrative sketch for the MPC optimization method for the one-dimensional state $x(t)$ and control $u(t)$. Based on the known data for x and u at t_0 (black solid lines) and the dynamics $\dot{x} = f(x, u)$ of the system, a cost functional $C[x(\cdot), u(\cdot)]$ (not shown) is minimized for the prediction horizon $t \in [t_0, t_0 + T_{\text{hor}})$. The resulting optimal control values for u and corresponding predicted states x (black dashed lines) are then used for the next ΔT_c until t_1, at which point the procedure is repeated for the interval $t \in [t_1, t_1 + T_{\text{hor}})$, resulting in new state and control values (blue). Note that the predictions need not always be correct, in case of uncertainties (x, blue curve, leftmost solid part).

2.1.1. Dynamical system

To begin with, we need to define what we mean with a dynamical system, in particular a *controllable* dynamical system. We follow [8], and define it as a tuple of:

- a time $t \in \mathbb{R}$
- a state variable $\boldsymbol{x} \in \mathbb{R}^{d_x}$
- parameters of the system $\boldsymbol{p} \in \mathbb{R}^{d_p}$
- control variables $\boldsymbol{u} \in \mathbb{R}^{d_u}$, a vector of all system variables that we can manually control
- a model for the dynamics of the system:

$$\begin{aligned} \boldsymbol{f} : \mathbb{R} \times \mathbb{R}^{d_x} \times \mathbb{R}^{d_u} \times \mathbb{R}^{d_p} &\to \mathbb{R}^{d_x} \\ \dot{\boldsymbol{x}} &= \boldsymbol{f}\left(t, \boldsymbol{x}, \boldsymbol{u}, \boldsymbol{p}\right) \end{aligned}$$

2.1.2. Optimal control problem

Additionally, we might have some parameters that we would like to control for. For our optimal control problem, we may also have:

- constraint functions governing constraints for inequalities $\boldsymbol{d} : \mathbb{R} \times \mathbb{R}^{d_x} \times \mathbb{R}^{d_u} \times \mathbb{R}^{d_p} \to \mathbb{R}^{d_d}$ and equalities $\boldsymbol{r} : \mathbb{R} \times \mathbb{R}^{d_x} \times \mathbb{R}^{d_u} \times \mathbb{R}^{d_p} \to \mathbb{R}^{d_r}$

$$\begin{aligned} \mathbf{0} &\geq \boldsymbol{d}\left(t, \boldsymbol{x}, \boldsymbol{u}, \boldsymbol{p}\right) \\ \mathbf{0} &= \boldsymbol{r}\left(t, \boldsymbol{x}, \boldsymbol{u}, \boldsymbol{p}\right) \end{aligned}$$

- a (scalar) cost or *optimality criterion* $\Phi(t, \boldsymbol{x}, \boldsymbol{u}, \boldsymbol{p})$ (possibly also containing integrals over time) which we want to keep as low as possible.

Thus, the optimal control problem is defined as finding the control functions $\boldsymbol{u}^*$ that make the system follow its bounds while operating with the minimal cost:

$$\begin{aligned} &\text{Find } \boldsymbol{u}^*(t) \text{ s.t.:} \\ &\quad \boldsymbol{u}^* = \underset{\boldsymbol{u}}{\operatorname{argmin}}\, \Phi\left(t, \boldsymbol{x}, \boldsymbol{u}, \boldsymbol{p}\right) \\ &\quad \dot{\boldsymbol{x}} = \boldsymbol{f}\left(t, \boldsymbol{x}, \boldsymbol{u}, \boldsymbol{p}\right) \\ &\quad \mathbf{0} \geq \boldsymbol{d}\left(t, \boldsymbol{x}, \boldsymbol{u}, \boldsymbol{p}\right) \\ &\quad \mathbf{0} = \boldsymbol{r}\left(t, \boldsymbol{x}, \boldsymbol{u}, \boldsymbol{p}\right) \end{aligned}$$

2.2. Model Predictive Control

Model Predictive Control (MPC) is well-established as a relatively general, easy-to-grasp and practically applicable control method for when a model of a system is known. There are therefore a variety of textbooks available for many different types and applications of MPC. Camacho and Bordons's book [2] is for example a popular, application-oriented start, whereas Grüne and Pannek's book [9] provides a more theoretical foundation for nonlinear MPC. For this thesis at hand, a general outlook on optimal control and MPC will be most appropriate. We will follow the notation and definitions of another good introduction, the Ph.D. thesis of Frasch [8]. Below, we will start with a some necessary definitions before going on to discuss the MPC problem itself and the way we can solve it.

To get one step closer to a solution, we parametrize our control problem by introducing a basis for our control functions, depending on some finite set of control parameters $\{q_k\}$, such that $\boldsymbol{u} = \boldsymbol{u}(\{q_k\})$. This makes the problem discrete, which allows for solution by direct methods. In figure 2.1, this parametrization simply corresponds to the constant value of the (here one-dimensional) control function during a time interval, a common choice for simplicity known in control engineering as *zero-order hold*. Again, we may however use any parametrization we would like, such as a piecewise linear basis function parametrization or splines. The problem now looks as follows:

$$
\begin{aligned}
&\text{Find } \{q_k^*\} \text{ s.t.:} \\
&\qquad \{q_k^*\} = \underset{\{q_k\}}{\operatorname{argmin}}\, \Phi\left(t, \boldsymbol{x}, \boldsymbol{u}\left(\{q_k\}\right), \boldsymbol{p}\right) \\
&\qquad\qquad \dot{\boldsymbol{x}} = \boldsymbol{f}\left(t, \boldsymbol{x}, \boldsymbol{u}\left(\{q_k\}\right), \boldsymbol{p}\right) \\
&\qquad\qquad \boldsymbol{0} \geq \boldsymbol{d}\left(t, \boldsymbol{x}, \boldsymbol{u}\left(\{q_k\}\right), \boldsymbol{p}\right) \\
&\qquad\qquad \boldsymbol{0} = \boldsymbol{r}\left(t, \boldsymbol{x}, \boldsymbol{u}\left(\{q_k\}\right), \boldsymbol{p}\right)
\end{aligned}
\tag{MPC-cont}
$$

2.2.1. Solution methods for the Model Predictive Control problem

Throughout the years, many different solution approaches have been explored in order to solve the MPC problem for very many different applications.

For a start, it is possible to integrate and solve the *ordinary differential equation* (ODE) of the state analytically or include the exact integrals in the formulation (for example using the *maximum principle* [10, 11] and then solving the resulting *boundary-value problem*). However, this is in most cases a hard task, and typically requires a different approach for each model. Although this might be feasible in some cases, the more common approach is to first discretize the ODE along the (finite) time horizon, resulting in (in general) a finite-dimensional *nonlinear programming* (NLP) problem. This class of problems is well-studied in mathematics, and has a range of different approaches for efficient solution.

In MPC's history, the most widely used approach has been *linear* MPC, where one assumes (or approximates) that the dynamics $\boldsymbol{f}$, as well as the constraints $\boldsymbol{d}$ and $\boldsymbol{r}$ are linear, and the cost Φ at most quadratic, in the state $\boldsymbol{x}$ and the control $\boldsymbol{u}$. In this case, the problem is reduced to a *quadratic programming* (QP) problem, which can be solved by *Active Set* methods, *Feasible Direction* methods or *Pivoting* to name a few. If one further restricts the cost to 1-norm–type functions, that is, using absolute values of combinations of states and

controls, then the problem can even be cast as a linear programming problem, which has even more efficient solution [2].

Explicit MPC [12] is an adaptation of linear MPC developed for situations where the computational resources are more restricted than what linear MPC typically allows for. In the developing work, Bemporad et al. proved that the solution to the MPC problem with constraints in some cases could be approximated by a piecewise linear function of state and control. This means, that to access the online solution to the control problem, one only requires very fast linear function evaluations, making the access of the optimal control orders of magnitude faster. The computational cost of optimisation is instead mainly contained in one offline solution of a *multiparametric* QP problem, in order to find the coefficients for the function, as well as the domains for each piecewise approximation. Using this method, MPC is made available also in systems requiring very fast response times or with very limited computational resources. Given the system equations and constraints, explicit MPC can be implemented easily by available software, such as MATLAB® Predictive Control Toolbox™ [13] and its graphical programming environment Simulink® [14].

In this work, we would however also like to include non-linear processes. For this more general case, solution methods also exist, but the problem is a much harder one. Firstly, since the problem no longer is quadratic, the same linearizations can no longer be made, making other iterative solutions necessary. Also, in the general case, the optimisation problem that has to be solved also becomes non-convex. This means that problems can now arise from being stuck in local optima, and the initialisation of the method can become much more important. The constraints may also create regions which are locally infeasible, even for problems that have a feasible solution.

The solution methods for the general NLP problem can be classified into different categories, and are in general solved by iterative sequences of QP problems, which are getting closer and closer to a solution. Typically, the gradients and sometimes Hessians (the matrix of second derivatives) are needed. There are however also cases of NLPs that some or most methods have problems finding solutions to, or where very many iterations and thus computation time is needed. This lack of theoretical or practical bound on the time to solution is one major reason why the fully nonlinear MPC is not used as much as linearized or approximated linear MPC. Another reason is the difficulty of obtaining or approximating nonlinear models of the state of a system, which however is too large a field to be in the scope of this thesis. For an overview, we refer to the one given in [8].

Many commercial and open-source solvers are available to solve optimisation problems, from linear to nonlinear and nonconvex. Examples of open-source ones are Ipopt [15], described in section 2.2.2 below, for nonlinear programming, and the GNU Linear Programming Kit [16], for linear programming. Proprietary ones include AMPL [17], which is both a nonconvex-capable solver and a mathematical language for formulating optimization problems, that also other solvers use, CPLEX [18], which also is for both linear and quadratic nonconvex problems, and Gurobi [19], for linear and mixed-integer programming. The tools Mathematica, MAPLE and MATLAB® also all provide optimisation toolboxes or frameworks for many different cases. A much more comprehensive list can be found in [20].

2.2.2. Ipopt and pyomo

In order to build a model of a problem in a both general and applicable way, the usage of a modelling language is helpful. It typically provides tools for defining the necessary mathematical parts of an optimization problem such as objectives, constraints and variables, and sometimes more advanced like expressions and set relations.

The python module pyomo [21] provides an algebraic modeling language in a software package designed for solving most types of optimization problems. Due to its integration in python, it allows for advanced high-level scripting using all of python's capabilities as a full-fledged programming language, such as defining functions for constraints and objectives dynamically, and handling input and output of data on-the-fly.

To solve the optimization problem itself, pyomo utilizes a user-specified optimization solver such as Ipopt [15] or Bonmin, and translates the optimization problem formulation into the modeling language of the solver. This allows for an advanced, general, high-level model description, while at the same time using the highly performant, low-level solvers available and appropriate for the user's need. *Interior Point Optimizer* (Ipopt) uses an interior point line search filter method to solve general nonlinear programming problems. Both pyomo and Ipopt are open source and freely available online, and are therefore excellent options to use.

A recent addition to pyomo was pyomo.dae, the pyomo differential and algebraic equation toolbox [22], which extends pyomo with support for optimization using differential equation and integral constraints and objectives. This makes pyomo.dae one of the few open-source, freely available frameworks that can model and optimize most classes of differential equations, especially without having to turn them into canonical forms. In pyomo.dae, standard discretization schemes such as backward, forward and central finite difference schemes for differentials and trapezoidal quadrature for integrals are provided, as are extension capabilities for easy integration of own discretization schemes.

In fact, a nonlinear MPC software has already been implemented in pyomo, and is possibly going to be released as an official extension [23].[1]

2.3. Model scenario

Our primary test case, illustrated in figure 2.2, is that of a motorized vehicle driving on a hilly road, trying to keep a set velocity on average, but occasionally going above and below this velocity to save fuel. One example for this is rolling without gas over hilltops and using downhill slopes to gain momentum for coming uphills. To add some nonlinearities, we may also put in constraints, for example that the motor temperature must not get too high. This is a typical predictive control scenario, as the amount of gas given depends on the future (predicted) slope of the road, and is similar to that for which several truck companies sell their products, advertised under names such as Predictive Powertrain Control (from Daimler [24]), Active Prediction (from Scania [25]), I-See (from Volvo [26]) and EfficientCruise® (from MAN [27], using Continental's eHorizon system [28]).

[1] Although attempts to contact the creators of the software have been made, in order to provide a comparison, no reply was received by the time of completion of this thesis.

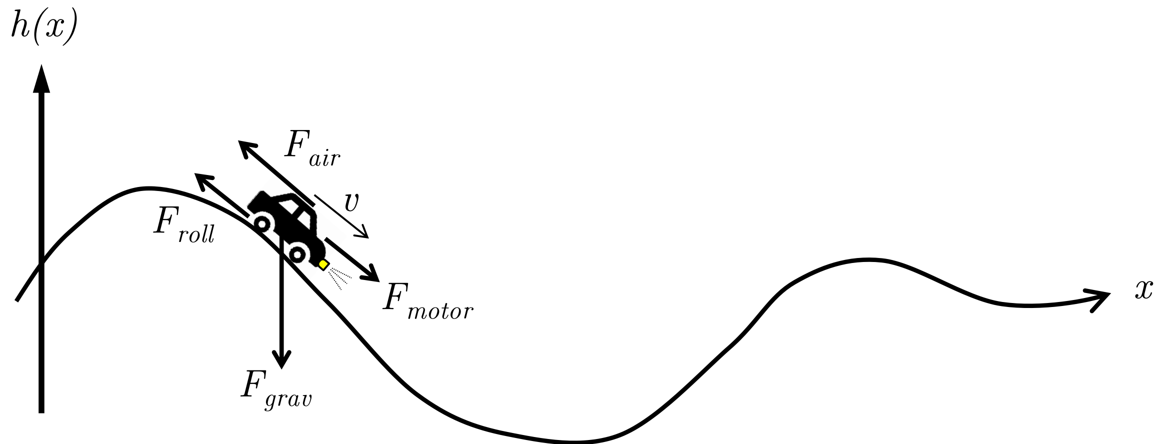


Figure 2.2.: A sketch of the primary model scenario used in this thesis. A motorized vehicle drives on a hilly road, affected by various forces, controlling its trajectory through by the thrust of its motor (F_{motor}). Various parameters affect the dynamics, summarized in table 2.1, together with the relevant dynamic equations and variables.

2.3.1. State and control variables

For our model scenario, we have a the following **state variables**:

- The one-dimensional position along the road x, with derivatives velocity $\dot{x}$ and acceleration $\ddot{x}$
- The motor temperature T

We also have **control variables**:

- The motor thrust or force F_{motor}
- The braking force F_{brake}

Using the distance along the road rather than horizontal distance, we keep our problem one-dimensional.

2.3.2. Physical laws and constraints

In addition, we then have a set of physical laws, or constraints, that govern the dynamics of the system:

- A set of **external forces** (see figure 2.2) on the vehicle:
  - Air resistance $F_{\text{air}} = -c_{\text{air}} |\dot{x}| \dot{x}$

Variable name	Symbol	Equivalent expression
Time	t	
Distance along road	x	$x(t)$
Velocity	v	$\dot{x}(t)$, $\frac{\mathrm{d}x}{\mathrm{d}t}(t)$
Acceleration	$\ddot{x}$	$\dot{v}(t)$, $\frac{\mathrm{d}^2x}{\mathrm{d}t^2}(t)$
Height	h	$h(x)$
Slope	s	$\frac{\mathrm{d}h}{\mathrm{d}x}(x)$
Motor temperature	T	$T_{\text{motor}}, T_{\text{motor}}(t)$
Motor thrust force	F_{motor}	$F_{\text{motor}}(t)$
Braking force	F_{brake}	$F_{\text{brake}}(t)$
Air resistance	F_{air}	$-c_{\text{air}} \lvert\dot{x}\rvert \dot{x}$
Effective gravitational force	F_{grav}	$-m_{\text{car}} g s(x)$
Rolling resistance	F_{roll}	$-\operatorname{sgn}(\dot{x})\, c_{\text{roll}} m_{\text{car}} g \,(1 - \lvert s(x)\rvert)$
Motor waste heat power	P_{heat}	$(1-\eta)\lvert\dot{x} F_{\text{motor}}\rvert$
Air cooling power	P_{cool}	$-k_{\text{cool}}\lvert\dot{x}\rvert(T - T_{\text{air}})$
Radiative cooling power	P_{rad}	$-k_{\text{rad}}(T - T_{\text{air}})$

(a) The relevant variables for the physical test case scenario.

Description	Constraint expression
Force balance	$m_{\text{car}}\ddot{x} = F_{\text{motor}} + F_{\text{brake}} + F_{\text{air}} + F_{\text{grav}} + F_{\text{roll}}$
Heat energy balance	$C_{\text{motor}}\dot{T} = P_{\text{heat}} + P_{\text{cool}} + P_{\text{rad}}$
Velocity limits	$v_{\text{low}} \leq \dot{x} \leq v_{\text{high}}$
Motor temperature limits	$(T_{\text{low}} \leq)\ T_{\text{motor}} \leq T_{\text{high}}$
Motor thrust limits	$F_{\text{motor, low}} \leq F_{\text{motor}} \leq F_{\text{motor, high}}$
Braking limits	$F_{\text{brake, low}} \leq F_{\text{brake}} \leq F_{\text{brake, high}}$

(b) The constraints for the physical test case scenario.

Objective	Expression
Motor energy	$\int_{t_0}^{t_{\text{end}}} F_{\text{motor}}(t')\dot{x}(t')\,\mathrm{d}t'$
Keeping velocity	$\int_{t_0}^{t_{\text{end}}} \lvert v_{\text{dev}}(\dot{x}(t'))\rvert^2\,\mathrm{d}t'$
Acceleration comfort	$\int_{t_0}^{t_{\text{end}}} \lvert\dddot{x}(t')\rvert^2\,\mathrm{d}t'$
Thrust smoothness	$\int_{t_0}^{t_{\text{end}}} \left\lvert\ddot{F}_{\text{motor}}(t')\right\rvert^2 + \left\lvert\ddot{F}_{\text{brake}}(t')\right\rvert^2\,\mathrm{d}t'$

(c) The objective terms in the cost function for the physical test case scenario.

Table 2.1.: A summary of the relevant variables, constraints and parameters in the physical test case scenario.

  - Work against gravity, $F_{\text{grav}} = -m_{\text{car}} g s(x)$, where m_{car} is the mass of the car, g is the acceleration of gravity ($\approx 9.8\text{ms}^{-2}$), and $s(x) = \frac{\mathrm{d}h}{\mathrm{d}x}$ is the slope of the road at distance x,[2] with $h(x)$ being the height.
  - Rolling resistance, $F_{\text{roll}} = -\operatorname{sgn}(\dot{x})\, c_{\text{roll}} F_{\text{normal}} = -\operatorname{sgn}(\dot{x})\, c_{\text{roll}} m_{\text{car}} g \left(1 - |s(x)|\right)$

  which summarizes to a total force determining the acceleration of the car according to Newton's second law:

$$\begin{aligned} m_{\text{car}}\ddot{x} = & F_{\text{tot}} \\ = & F_{\text{motor}} + F_{\text{brake}} + F_{\text{air}} + F_{\text{grav}} + F_{\text{roll}} \end{aligned} \qquad \text{(Force Bal.)}$$

- Laws governing the **evolution of the temperature** of the motor, modeled with the following energetical components:
  - Heating from waste heat, proportional to the motor power times the efficiency (η) inverted: $P_{\text{heat}} = (1-\eta)|\dot{x}F_{\text{motor}}|$
  - Cooling from the cooling system, roughly proportional to the velocity and the temperature difference to the air: $P_{\text{cool}} = -k_{\text{cool}}|\dot{x}|(T - T_{\text{air}})$
  - Cooling due to passive radiation of heat, roughly proportional to the temperature difference between the motor and the outside: $P_{\text{rad}} = -k_{\text{rad}}(T - T_{\text{air}})$

  Thus, a very rough energetical model of the engine temperature could be summarized as:

$$C_{\text{motor}}\dot{T} = P_{\text{exchange}} \qquad \text{(Temp. energy bal.)}$$
$$= P_{\text{heat}} + P_{\text{cool}} + P_{\text{rad}} \qquad (2.1)$$

  where C_{motor} is the thermal capacity of the motor.

Whereas these nonlinear differential equations will be imposed as constraints based on physics, we might also have a set of **upper and lower bounds on variables** that we want to impose due to economy, comfort or durability of the materials. In our case, these might be:

- Upper (and lower) bounds for the motor temperature to ensure optimal performance and long life ($T_{\text{low}} \leq$) $T_{\text{motor}} \leq T_{\text{high}}$
- Upper and lower bounds for the velocity, to keep in pace with traffic, to hold comfort and to reach a destination in time $v_{\text{low}} \leq \dot{x}(t) \leq v_{\text{high}}$, where the bounds v_{low} and v_{high} may also be varying (due to traffic, or to slow down in curves)
- Upper and lower bounds on motor power, similarly.

[2] The slope s is thus defined as the height gained per distance travelled along the road, which simplifies the expression, and limits s to the interval $(-1, 1)$.

2.3.3. Costs and objectives

Finally, we may decide on **objectives**, which are the functions to be optimized for, for example:

- Energy consumption by the motor, the integral of power over time:

$$\begin{aligned}\Phi_{\text{motor}} &= \int_{t_0}^{t_{\text{end}}} P_{\text{motor}}(t')\mathrm{d}t' \\ &= \int_{t_0}^{t_{\text{end}}} F_{\text{motor}}(t')\dot{x}(t')\mathrm{d}t'\end{aligned} \tag{En. cost}$$

- Deviations from the desired velocity band, such that this is only left when enough energy is saved:

$$\Phi_{\text{vel}} = \int_{t_0}^{t_{\text{end}}} \left|v_{\text{dev}}(\dot{x}(t'))\right|^2 \mathrm{d}t' \tag{Vel. cost}$$

  where

$$v_{\text{dev}}(\dot{x}) = \begin{cases} \dot{x} - v_{\text{band}}^{+}, & \text{if } \dot{x} > v_{\text{band}}^{+} \\ v_{\text{band}}^{-} - \dot{x}, & \text{if } \dot{x} < v_{\text{band}}^{-} \\ 0, & \text{otherwise.} \end{cases}$$

- Passenger comfort; punishing the first or second derivatives of acceleration and motor/braking force, in order to avoid too violent acceleration (also keeping the control variable smooth):

$$\Phi_{\text{acc}} = \int_{t_0}^{t_{\text{end}}} \left|\dddot{x}(t')\right|^2 \mathrm{d}t' \tag{Acc. cost}$$

$$\Phi_{\text{force}} = \int_{t_0}^{t_{\text{end}}} \left|\ddot{F}_{\text{motor}}(t')\right|^2 + \left|\ddot{F}_{\text{brake}}(t')\right|^2 \mathrm{d}t' \tag{Mot. cost}$$

- We may also wish to use several of those in combination, for example through such as a weighted sum:

$$\Phi = w_{\text{motor}}\Phi_{\text{motor}} + w_{\text{vel}}\Phi_{\text{vel}} + w_{\text{acc}}\Phi_{\text{acc}} + w_{\text{force}}\Phi_{\text{force}}$$

Using the energy consumption cost, even if together with the velocity deviation cost, typically shifts the observed velocities downwards. To compensate for this, we may introduce another cost term to the energy integrand, such as $-\alpha(\dot{x} - v_{\text{set}} + C)^2$, where α and C are appropriately scaled constants. To then set an equilibrium velocity for a constant, flat height profile, one can find the minimum of the resulting cost integrand with respect to velocity (by differentiating) and set α and C such that the minimum of the cost is at the desired velocity value.

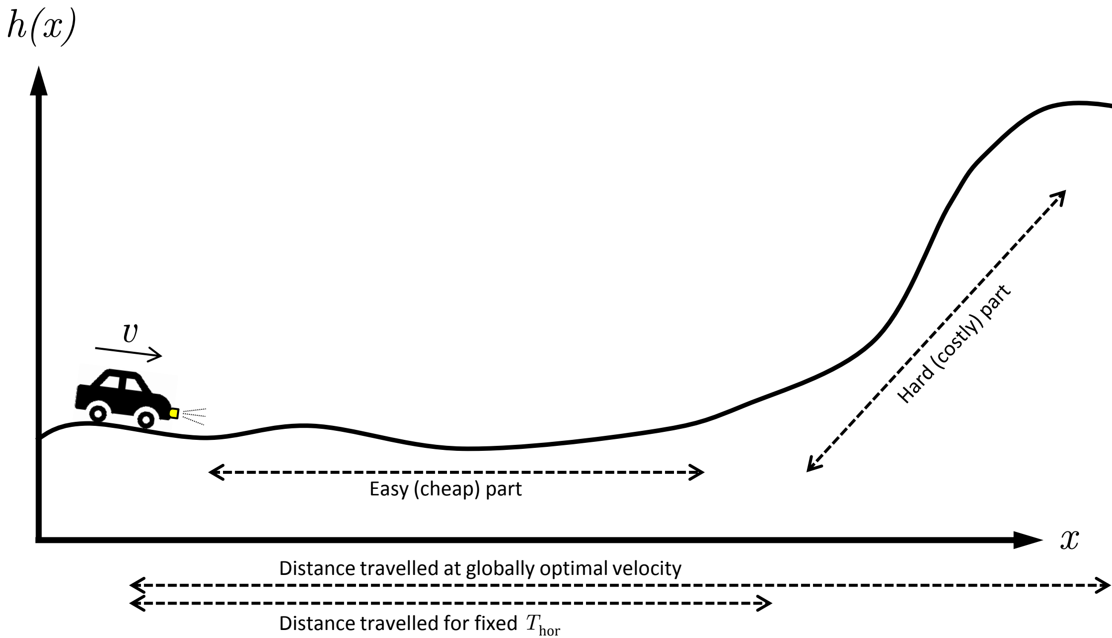


Figure 2.3.: Illustration of possible consequences of fixing the endpoint in time rather than in space.

2.4. Distance reparametrization

In most cases, such as above, the parametrization of dynamical systems is most naturally done by time; most equations defining the dynamics are usually described with time derivatives, for example Newton's second law $m\frac{\mathrm{d}^2\boldsymbol{x}}{\mathrm{d}t^2} = \boldsymbol{F}$, defining the linear relationship between the second time derivative of an object's position and the net force applied to it. Additionally, some of the integral terms of the cost function, such as the net acceleration experienced by a passenger, are inherently easiest to express relative to time.

However, when discretizing the MPC problem in equation (MPC-cont) with the scenario of section 2.3 with respect to time, one encounters other problems. For the first, the description of some other variables may be trickier to express in a straightforward fashion with respect to time. In our case, this becomes evident for the height variable, that is interpolated from tabulated values with respect to distance, and thus also for its distance derivative, defining the slope. As the position at a certain discretization timepoint then is strongly coupled to velocity and position at neighbouring points, and thus indirectly through them to all points, the problem becomes very strongly coupled and possibly harder to solve, as each slope value contributes in the force balance equation (see equation (Force Bal.) in section 2.3.1) through gravity.

Additionally, we need to take care how to define our prediction horizon. If we define it to have a fixed length of time, the calculated solution also only needs to take this time into account, and thus only the stretch of road travelled during this time. Since some of the parameters that influence the cost of the problem — the height profile of the road, or the corresponding slope — are functions of distance only, this means that, by controlling such that high-cost (i.e. steep uphill) parts are not within the prediction horizon, the cost

is lowered. In terms of the car, illustrated in figure 2.3, this means that it is locally more optimal to slow down, such that the hill only comes after the prediction horizon. However, if letting the prediction horizon be until the car has passed the high-cost part, this might be a suboptimal strategy.

To combat such problems, where a more optimal state of the current optimisation iteration influences the next one negatively, there are various strategies. One can put constraints on the ending state, such that stability, feasibility or some kind of minimal requirement is fulfilled, or add a *terminal cost* for the final state, which reflects the possible disadvantage to the next iteration. However, these measures need to be cleverly designed, in order to actually achieve the desired affect, while still allowing the optimisation freedom to explore and find the optimal control solution.

In our case, we can follow the simpler approach of letting the prediction horizon have a spatial length rather than a temporal length, that is, we optimize until a certain distance point. This is possible using cut-off terms in the cost integral, to make the cost go to zero for anything past the horizon, and adding relaxation terms to our constraints and dynamical equations to make them automatically fulfilled once past the horizon. However, these terms would be highly nonsmooth and incredibly nonlinear, and therefore harder to optimize for.

The even simpler approach is therefore to discretize our domain with respect to distance instead of time. Since there are strictly positive lower bounds on velocity $v = \dot{x} = \frac{\mathrm{d}x}{\mathrm{d}t} > 0$, we have that distance (time) is a strictly increasing, bijective, function of time (distance), and both variables can equally well be used to describe the system. Since the system does not explicitly depend on time (but does on distance), no parametrization problems such as those of the height variables in the time–parametrized system appear,[3] eliminating one more problem.

In order to transform the system, one thus only need to reparametrize all variables with respect to distance instead of time, and also transform all time–dependent derivatives and integrals to distance integrals. The resulting equations corresponding to the ones in section 2.3 are slightly more nonlinear (see below) in terms of the discretization variable, but this is more than outweighed by the constant values for beginning and end of the prediction horizon in terms of difficulty of solution.

2.4.1. Distance-parametrized system equations

When changing to the distance parametrization, most expressions stay the same. However, those including a time derivative or integral will change. Through the transformations

$$\frac{\mathrm{d}}{\mathrm{d}t} = v(x)\frac{\mathrm{d}}{\mathrm{d}x}, \qquad \mathrm{d}t = \frac{\mathrm{d}x}{v(x)}$$

[3] This is of course not the case anymore if also regarding traffic; this is however far beyond the scope of this thesis.

the need for t as an optimization variable is eliminated. This transforms equations (Force Bal.) and (Temp. energy bal.) into

$$m_{\text{car}} v \frac{\mathrm{d}v}{\mathrm{d}x} = F_{\text{tot}} \qquad \text{(Dist. force bal.)}$$

$$C_{\text{motor}} v \frac{\mathrm{d}T}{\mathrm{d}x} = P_{\text{exchange}} \qquad \text{(Dist. temp. energy bal.)}$$

and, with x_0 and x_{end} replacing t_0 and t_{end}, transforms the cost expressions (En. cost), (Vel. cost), (Acc. cost) and (Mot. cost) into

$$\begin{aligned}
\Phi_{\text{motor}} &= \int_{x_0}^{t_{\text{end}}} \frac{\mathrm{d}P_{\text{motor}}(x')}{v} \,\mathrm{d}x' && \text{(Dist. en. cost)} \\
&= \int_{x_0}^{x_{\text{end}}} F_{\text{motor}}(x') \,\mathrm{d}x' \\
\Phi_{\text{vel}} &= \int_{x_0}^{x_{\text{end}}} \frac{|v_{\text{dev}}(v(x'))|^2}{v(x')} \,\mathrm{d}x' && \text{(Dist. vel. cost)} \\
\Phi_{\text{acc}} &= \int_{x_0}^{x_{\text{end}}} \frac{\left| v(x) \frac{\mathrm{d}}{\mathrm{d}x} \left(v(x) \frac{\mathrm{d}v(x)}{\mathrm{d}x} \right) \right|^2}{v(x)} \mathrm{d}x' && \text{(Dist. acc. cost)} \\
&= \int_{x_0}^{x_{\text{end}}} v(x) \left| (\frac{\mathrm{d}v(x)}{\mathrm{d}x})^2 + v(x) \frac{\mathrm{d}^2 v}{\mathrm{d}x^2} \right|^2 \,\mathrm{d}x' && (\textit{since } v > 0) \\
\Phi_{\text{force}} &= \int_{x_0}^{x_{\text{end}}} \left| \ddot{F}_{\text{motor}}(x') \right|^2 + \left| \ddot{F}_{\text{brake}}(x') \right|^2 \mathrm{d}t'. && \text{(Dist. mot. cost)}
\end{aligned}$$

As these new equations contain enough nonlinearities on their own, it was decided to simplify the overall system and remove the temperature variable, as well as its equations, to first focus on the most basic properties of the system. In addition, to keep the Φ_{force} cost term simple, it was decided to simply replace the time derivatives with distance derivatives, which although not exactly equivalent is not much different for (relatively) slowly changing velocities changing in a smaller range well away from zero. The effect remains the same; the force values are smooth.[4]

Similarly to in for the time-discretized version of the energy cost Φ_{motor}, the distance-descretized cost also requires an extra compensating term in order to control the equilibrium point. In order to affect the cost for velocities lower than the set point less (thus still promoting saving energy by accelerating over the set point through gravity), an asymmetrical term $-\frac{\alpha}{v}$ was added, where $\alpha = 2c_{\text{air}} v_{\text{set}}^3$ for an equilibrium velocity of v_{set} on a completely flat road.

2.5. External parameters and data

In order to generate scenarios similar to the real world, example height data along roads may by downloaded using the Google Maps Elevation API [29], which allow downloading

[4] This is a requirement for the coming steps of approximation with diffusion maps, as elaborated upon in section 3.2.

of their data (albeit not using it in commercial products or abusing it in publication) with a free license. A smooth height function h that can be differentiated to form the slope $s = \frac{\mathrm{d}h}{\mathrm{d}x}$ can then be formed for example by fitting of smooth functions. One good example would be splines such as *B–splines* or fitting with Fourier polynomials. In practice, implementing splines in `pyomo` does not come as naturally as Fourier approximations, as the expression for a spline changes across the domain, which is harder to build into the optimisation framework.

In practice, for the implementation of a controller, height data over the prediction horizon could be retrieved for a big enough area using services similar to the Google Maps Elevation API. With a current position localization through a GPS navigation system, height data for the upcoming road section could then be extracted for use in the controller.

3. Learning Control Heuristics

For those with the right tools, there are vast amounts of information to unlock in data – through analysis, pattern finding, approximation and much more. In this chapter, a broad introduction is given to the fields relevant for this thesis, in order introduce and clearly define the position of the methods used in relation with other ones in literature.

3.1. Approximation of Model Predictive Control as a function

Summarizing, the MPC method described in section 2.2 is essentially a function, which takes data (e.g. a system state and some external information about the future) and returns an output (e.g. values for applying a control strategy). Although this is a highly complex and nonlinear function, we can still hope to approximate it with the tools available. With the increase in data available in all areas of society in the latest years, these tools have become more and more important for both academia and business analysis, and much research has gone into developing both new tools and theory about them, in the area nowadays known as *Machine Learning*.

For the MPC method, approximating the solution of the whole problem is highly desired. The reasons for this are numerous: for example, to avoid long computation times that are undesirable in real-time settings by using something simpler to compute, or to provide a solution where an optimization solver might fail or get stuck in a local optimum. There are also many examples where such things have been tried. One successful one is Explicit MPC [12], where piecewise linear approximations are made from sampled solutions. More complex ones include using neural networks to approximate the input-output behaviour [4, 5].

3.2. Representation learning

Representation learning, as the name implies, is a subset of machine learning where the task is to learn a different *representation* of the data. There are several reasons to do so, which is usually reflected in what kind of representation we would want. The simplest one could for example be *compression*, where we are interested in an as small or compact representation of the data as possible, but many others include that we would additionally want to extract some relevant *features* or *structure*, that could aid in the understanding of the data. The differently represented data would then be subject to further analysis, and not uncommonly fed to another supervised or unsupervised machine learning algorithm, which would benefit from the new representation, for example because of the smaller amount of data to process, or more exposed relevant details.

However, in all cases we would want to preserve as much (relevant) information as possible in the data, while being as general as possible about what this information could

contain, since in many cases we might not know what qualities of the data we are looking for. What properties are then common to most data that we can use in finding the new representation? Following Bengio et al. in [30], we here list some:

- *Smoothness.* If two points are *similar* or close according to some measure for the original data, this relationship should be preserved in the representation, in order to aid our understanding. This allows us to extract *manifold-like structures*, or to find *natural clusterings* of the data.
- *Multiple explanatory factors.* There is often not only one factor explaining variation of data, and the behaviour of this factor for one part of the data often *generalizes* to the rest, so *disentangling* these can lead to a good representation. However, they can still often be *sparse*, meaning that only few of them are active at one data sample. Often, the factors can show *spatial* or *temporal coherence* on different scales, such fast fluctuations in a stock price around a longer-time trend, or be *hierarchical*, with many simple factors building up a hierarchy of more abstract ones, where the relations between the abstract ones might be simpler to describe. An example there could be the movement of motor parts in a moving car with respect to the outside world, which might be very complicated, but readily lets itself be decomposed in the translation of the vehicle, which is a common factor for the whole car, and in the working of the motor, which is much simpler to understand without the exterior movement. The movement of the motor part can then easily be described as ”the car movement + the movement within the motor”, which is a simple additive relationship, instead of a complex curve through space, a feature common in many other laws of physics.

Because we often do not know exactly what we are looking for in a representation, representation learning is usually *unsupervised*, that is, no correct examples or targets are provided to the method. Instead, as mentioned above, another supervised or unsupervised machine learning algorithm may be run on top of the new representation. If in this case another representation is obtained, that then is used as input for another layer of learning, with arbitrary numbers of repetitions, the model and the learning method is called *deep*, as in *deep learning*. There is a multitude of different methods available for representation learning, both single-layer and deep, and to give an extensive recount is unfortunately outside the scope of this thesis. However, to provide a sense of how the methods presented below are located within the field, the three categories outlined in [30] provide a good basis:

- *Probabilistic Models.* Here, the problem is posed as finding a set of *latent random variables* that describe the underlying distribution of data in an as likely but sparse way as possible, finding it by (approximately) maximising the *posterior probability* of the latent variables given the data. Examples include the popular *restricted Boltzmann machine*[31, 32] and other Markov random field and Boltzmann machine models, and (debateably, see [30, p. 1805]) *sparse coding*.
- *Direct Representation Parametrizers.* In contrast to probabilistic models, where the representation is in underlying variables that represent the data, the function that is learnt here *directly* encodes the representation of the data. Most methods do this by constructing a parametrized encoding–decoding scheme, and then minimize the

reconstruction error, that is, find the parameters that make the representation lose as little information as possible. Of course, in order to make the representation different from the original data, some structure has to be enforced upon the encoder (and decoder), such as lower output dimension or sparsity for the representation. Examples include all kinds of *autoencoders*, and other related methods such as *Predictive Sparse Decomposition* [33].

- *Manifold–based Models.* In manifold–based models, the data is expected to lie only on or close to a manifold (or several) with (much) lower dimension than the original data representation. The problem is then often posed as finding out the geometry of this manifold, and then describing it in an appropriate manner. Examples include *t-SNE* [34], and, described below, *diffusion maps*. In addition, some autoencoders might also be interpreted as describing a manifold (again, [30]).

Before we start describing diffusion maps, it might be helpful to give a very simple example of a technique which is used for the same purposes as many representation learning methods, *Principal Component Analysis* (PCA), for which an excellent tutorial picture is given in [35]. In PCA, we solve an eigenproblem to obtain a linear representation of the data, and as such, we also get a direct representation encoding, as eigenvectors allow us to extract the representation of any datapoint. The eigenvectors also span a (linear) manifold. From a probabilistic perspective, the eigenvectors are also related to the leading eigenvalues of the covariance matrix of a multivariate Gaussian distribution.

The method PCA is extremely useful for *dimensionality reduction*, which is at the heart of what we aim to achieve by finding a good representation. However, since it can only extract linear relationships, it is too limited for our purposes. For finding a non-linear low-dimensional manifold reduction, we therefore need to turn to other techniques.

3.3. Diffusion maps

Diffusion maps [36] are a data analysis tool for parametrizing lower-dimensional manifolds embedded in higher dimensions. The method is based upon distances between near neighbours, measured according to some kernel and metric, and as such. The idea is, that for points sampled in an underlying manifold, the distance to these closest ones in the high-dimensional space should approximately be the same as the distance in or along the manifold. This is used for a type of diffusion operator between the points, which can be normalized to produce an approximation of the real diffusion operator on the manifold, and in the limit of unlimited data also converge to the continuous Laplace-Beltrami operator[37]. As the eigenvectors or eigenfunctions of this continous operator provides useful parameterizations of the underlying manifold [38], the key idea is the discrete eigenvectors should do this as well. These parametrizations have been constructed synthetic and experimental data with success [6, 39, 40]. A sketched illustration of a parametrization is shown in figure 3.1.

What these results show, is that the eigenvector values can be interpreted as coordinates in the most significant directions, or the directions the have most contribution to the distance within this manifold. This is illustrated in the results of figures 3.2 and 3.3. In the

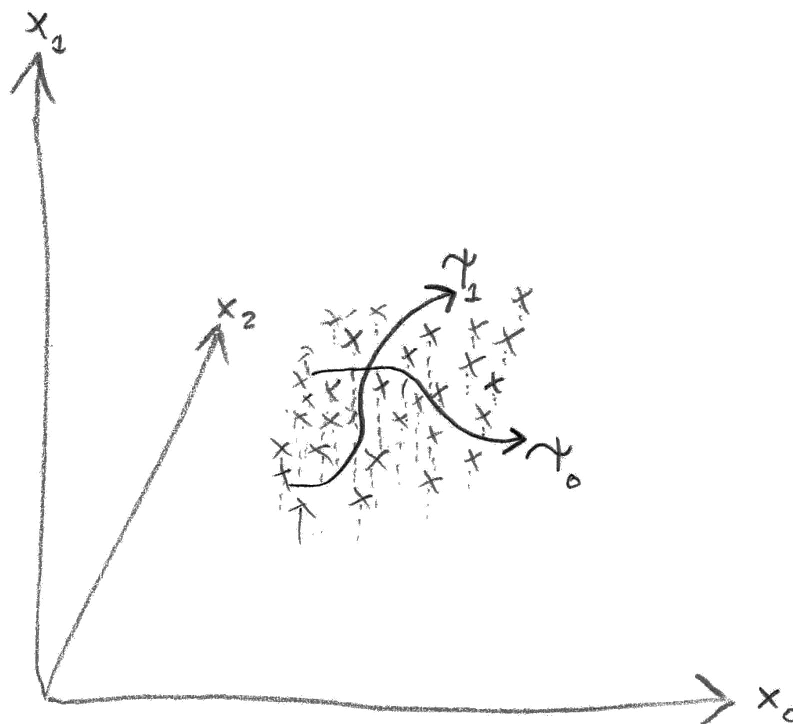


Figure 3.1.: Sketch of the parametrization by the diffusion maps method. For a set of data in a high number of dimensions (in the case, three: x_0, x_1 and x_2), the method parametrizes the datapoints by two component directions (ψ_0 and ψ_1), such as to faithfully represent distances along the data. These directions can be seen as similar to the principal components extracted in PCA, but are typically nonlinear.

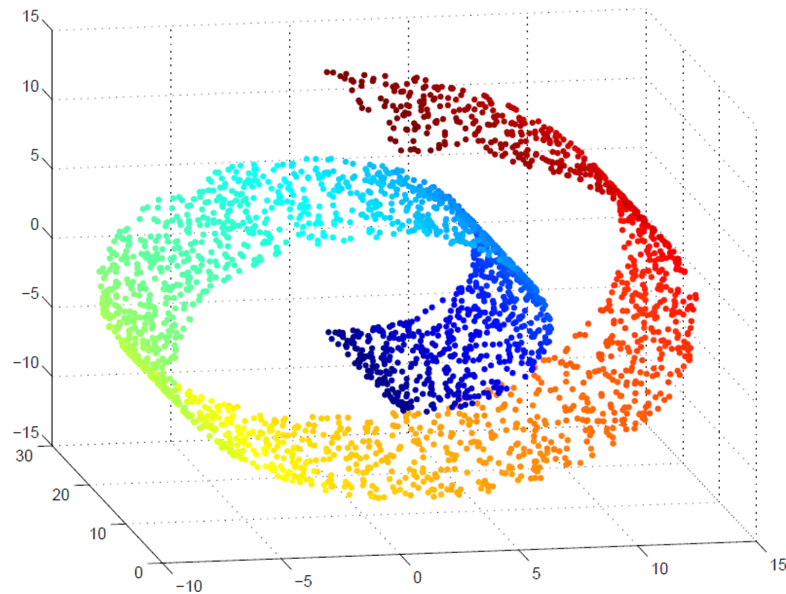


Figure 3.2.: A parametrization of a 2D surface in 3D. The surface is colored according to the first diffusion map coordinate. Picture from [41].

figure 3.2, a classical example, a curled-up two-dimensional manifold embedded in three-dimensional space is shown. The manifold is coloured according to the first eigendirection obtained by diffusion maps. It has captured a main feature of the data: the coordinate along the spiral.

In figure 3.3, the method was instead applied to a set of points uniformly distributed inside the logo of the Technical University of Munich. Again, the first eigendirection captures something essential; the length along the continuous, thick curve making up the logo. In addition, some other eigendirections are shown: the second eigendirection shows a repetition of the first one, folded over itself. This reflects the nature of diffusion maps as eigenmodes of the heat operator, whose eigenfunctions are oscillating functions over the manifold, with increasing frequency with increasing eigenvalue. However, we also see a second unique eigendirection: along the height of the logo. As this direction is shorter than the length of the curve, its eigendirection has a lower eigenvalue, and therefore comes later.

As eigenvectors, the eigendirections also form an orthogonal basis of the space of functions on the data, giving a method to parametrize other functions through them as well. This is also the basis for interpolating and extending functions through *Geometric Harmonics* [42] (see section 3.4.3).

3.3.1. Method

To understand how it works, we will go through the diffusion maps method for a data domain $\mathcal{D}$, with some measure $d\mu$. In this setting, the data can be interpereted both as a point density over continuous space (for example $d\mu(\boldsymbol{x}) = p(\boldsymbol{x})\, d\boldsymbol{x}$) and as as discrete points in space (with $d\mu$ preferably being the counting metric, which integrates to the

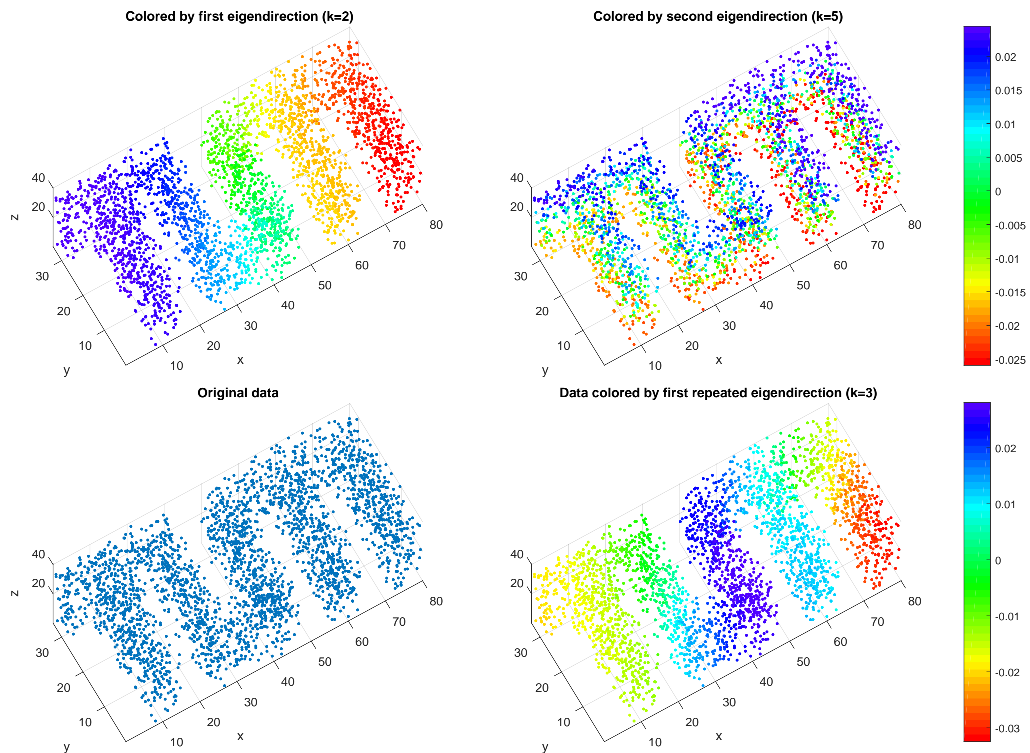


Figure 3.3.: A diffusion maps parametrization of the logo of the Technical University of Munich. Three eigenmodes are shown. Top-left, the first eigendirection is shown, which parametrizes the length along the logo. Top-right, the eigenmode parametrizing the height of the logo is seen. As the logo is longer than it is wide, this mode has a higher frequency and therefore comes later in the order. Another mode, which corresponds to the first eigendirection but doubled in frequency, can be seen in the bottom-right.

number of points in the domain of integration). We will use $L^2(\mathcal{D}, \mathrm{d}\mu)$ as the space of square-integrable functions on the data with respect to the metric $\mathrm{d}\mu$.

In the discrete case, functions and operators will in most cases naturally correspond to vectors and matrices. This simplifies integrals and operator applications to matrix and vector operations, and turns the eigenproblem into a symmetric matrix eigenvalue problem, for which there are many algorithms available. This makes the method highly practically implementable for discrete data.

Following, we look at the steps of the method.

- Choose a metric and a kernel for distances between datapoints: a scalar, symmetric, positive semi-definite function k that associates a real value to each pair of datapoints.

$$k : \mathcal{D} \times \mathcal{D} \to \mathbb{R}$$
$$k(\boldsymbol{x}, \boldsymbol{y}) = k(\boldsymbol{y}, \boldsymbol{x})$$
$$\int_{\mathcal{D}} \int_{\mathcal{D}} f(\boldsymbol{x}) f(\boldsymbol{y}) k(\boldsymbol{x}, \boldsymbol{y}) \, \mathrm{d}\mu(\boldsymbol{x}) \mathrm{d}\mu(\boldsymbol{y}) \geq 0, \forall f \in L^2(\mathcal{D}, \mathrm{d}\mu)$$

  A common choice is to take an *isotropic* kernel that has rotational invariance, i.e. $k(\boldsymbol{x}, \boldsymbol{y}) = h(\|\boldsymbol{x} - \boldsymbol{y}\|)$, with the most common one being the Gaussian kernel

$$k(\boldsymbol{x}, \boldsymbol{y}) = \exp\left[-\frac{\|\boldsymbol{x} - \boldsymbol{y}\|^2}{\varepsilon^2}\right].$$

  This metric is used to build a kernel operator $K : L^2(\mathcal{D}, \mathrm{d}\mu) \to L^2(\mathcal{D}, \mathrm{d}\mu)$, which takes a function and convolutes it according to the kernel, thus sampling it according the the distribution of the data.

$$Kf = \int_{\mathcal{D}} f(\boldsymbol{y}') \, k(\,\cdot\,, \boldsymbol{y}') \; \mathrm{d}\mu(\boldsymbol{y}')$$

- In order for the method to approximate only the geometry of the data, and not be influenced by the density of the samples, an extra normalization step is carried through. That is, we put less weight on regions with high density, such that their contribution is averaged over the neighbours in the same area. Through creating a measure for the local density using the kernel, $\rho(\boldsymbol{x}) = \int_{\mathcal{D}} k(\boldsymbol{x}, \boldsymbol{y}') \; \mathrm{d}\mu(\boldsymbol{y}')$, and using it to weight the kernel operator, we create a normalized kernel.

$$k_{\text{norm}}(\boldsymbol{x}, \boldsymbol{y}) = \frac{k(\boldsymbol{x}, \boldsymbol{y})}{\rho(\boldsymbol{x})\rho(\boldsymbol{y})} \tag{DM-norm}$$

- In order to make it a diffusion operator over the data, we need to create an operator that describes a probability to diffuse to other points in the data space. This means the operator must be *stochastic*, i.e. must conserve probability, which is done by using another normalization. Since $\tilde{k}(\boldsymbol{x}, \boldsymbol{y})$ defines the transition probability from $\boldsymbol{x}$ to $\boldsymbol{y}$, we want that the total probability of diffusing to any other point in the domain should be 1. We thus let

$$\tilde{k}(\boldsymbol{x}, \boldsymbol{y}) = \frac{k_{\text{norm}}(\boldsymbol{x}, \boldsymbol{y})}{\int_{\mathcal{D}} k_{\text{norm}}(\boldsymbol{x}, \boldsymbol{y}') \, \mathrm{d}\mu(\boldsymbol{y}')}$$

giving that

$$\int_{\mathcal{D}} \tilde{k}\left(\boldsymbol{x}, \boldsymbol{z}'\right) \mathrm{d}\mu(\boldsymbol{z}') = \frac{\int_{\mathcal{D}} k_{\text{norm}}\left(\boldsymbol{x}, \boldsymbol{z}'\right) \mathrm{d}\mu(\boldsymbol{z}')}{\int_{\mathcal{D}} k_{\text{norm}}\left(\boldsymbol{x}, \boldsymbol{y}'\right) \mathrm{d}\mu(\boldsymbol{y}')} = 1 \qquad (\forall \boldsymbol{x} \in \mathcal{D})$$

That is, the new kernel applied to a distribution of data, will "diffuse" it in space, but still keep the total amount constant.

- Find the eigenvectors and eigenvalues of the operator $\tilde{K}$, by transforming it to the similar, symmetric problem, with $v(\boldsymbol{x}) = \int_{\mathcal{D}} k\left(\boldsymbol{x}, \boldsymbol{y}'\right)\ \mathrm{d}\mu(\boldsymbol{y}')$ (so that $\tilde{k}\left(\boldsymbol{x}, \boldsymbol{y}\right) = \frac{k(\boldsymbol{x},\boldsymbol{y})}{v(\boldsymbol{x})}$)

$$\begin{aligned} k_{\mathrm{S}}\left(\boldsymbol{x}, \boldsymbol{y}\right) &= \tilde{k}\left(\boldsymbol{x}, \boldsymbol{y}\right) \sqrt{\frac{v(\boldsymbol{x})}{v(\boldsymbol{y})}} \\ &= \frac{k\left(\boldsymbol{x}, \boldsymbol{y}\right)}{\sqrt{v(\boldsymbol{x}) v(\boldsymbol{y})}} \end{aligned}$$

so that k_{S} is symmetric, and the eigenvalue problem turns into

$$\begin{aligned} K_{\mathrm{S}}\psi_{\mathrm{S}} &= \lambda_{\mathrm{S}}\psi_{\mathrm{S}} \\ &\left(= \int_{\mathcal{D}} \frac{k\left(\,\cdot\,, \boldsymbol{y}'\right)}{\sqrt{v(\,\cdot\,) v(\boldsymbol{y}')}} \psi_{\mathrm{S}}(\,\cdot\,)\, \mathrm{d}\mu(\boldsymbol{y}') \right) \\ &= \left(v^{\frac{1}{2}} \tilde{K} v^{-\frac{1}{2}} \right) \psi_{\mathrm{S}} \end{aligned}$$

and by right-multiplying with $v^{-\frac{1}{2}}$, we get

$$\tilde{K}\left(v^{-\frac{1}{2}}\psi_{\mathrm{S}}\right) = \lambda_{\mathrm{S}}\left(v^{-\frac{1}{2}}\psi_{\mathrm{S}}\right)$$

so that for every eigenpair $(\lambda_{\mathrm{S}}, \psi_{\mathrm{S}})$ of K_{S}, we have $\left(\lambda_{\mathrm{S}}, v^{-\frac{1}{2}}\psi_{\mathrm{S}}\right)$ as an eigenpair of $\tilde{K}$.

The resulting eigenfunctions can then be used as quantifiers of relevant coordinates on the manifold.

The normalization step in equation (DM-norm) allow the eigenfunctions of the kernel operator to exactly approximate those of the Laplace-Beltrami operator (sum of the second partial derivatives, $\Delta = \sum_i \frac{\partial^2}{\partial x_i^2}$) on the manifold [37]. Recalling that these eigenfunctions for a rectangular domains are a product of sines and cosines along the principal dimensions of the rectangle, we can now begin to understand why this should give a good parametrization. Since our diffusion is limited to within the data, we have correspondingly Neumann boundary conditions on our domain. Thus, the first eigenfunction should be constant, and the following ones should be sines along the "longest" directions of the manifold.

3.3.2. For discrete datapoints

If we want to use discrete datapoints in the above method, we might recall that the individual datapoints are in fact samples of the underlying distribution of the data. Therefore, the values we obtain from the diffusion maps method for discrete data are also only samples of the values of the underlying parametrization. This means that one does not explicitly obtain an expression of the underlying low-dimensional approximations, but only samples of them at the points of the data, i.e. if our eigenvector of choice for example describes the length along the TUM-logo, as in figure 3.3, the resulting eigenvectors gives the value of that length variable for each datapoint that was input to the method. This means that, if we want to know the value of the parametrization at a point that was not included in the original diffusion maps calculation, we need to use other methods, such as interpolation.

3.3.3. Repeated eigendirections

As seen in figure 3.3, one can have the problem that some of the directions are repeated. As already mentioned, his since the method is based on the eigenvectors of the diffusion operator, which are infinite in number and repeated in each dimension. To recover the most parsimonious directions, a method was developed by Dsilva et al. [40], going successively through the directions with highest weight, and trying to recreate the coordinates as local linear combinations of the previous ones. The method then rates the directions based on leave-one-out–cross-validation error of this fit. A higher error means the direction could not be build from the previous ones, and thus that it is a new direction, rather than a repetition of a previous one.

3.3.4. Diffusion maps for dynamical systems

A popular application using the relevance–extracting capabilities of diffusion maps has been trying to extract the overarching *macroscopic variables* of large dynamical systems. In the spirit of using good representations, in many places in engineering and the natural sciences, properties a system which is very complicated on the small scale (from interactions between millions of atoms) can be described well in terms of larger–scale quantities (pressure, temperature). For example, these applications have been used to extract underlying parameters in dynamical processes, such as identifying frequencies in pixel data from videos and tones in digital music [43–45], or exploring energy surfaces for configurations of large molecules [46].

The knowledge that the system is governed by differential equations, and thus smoothly *temporally coherent*, is incorporated in different ways in the methods. In [43, 44], the metric is built up through explicitly coding temporal correlations into the distance metric. In [45], a *linear contracting observer*, related to the *Koopman Operator* [47–50] is constructed to reconstruct the dynamics on the diffusion maps coordinates. In [46], the diffusion maps coordinates are used to explore parameter sets related to timesteps that take much longer to simulate.

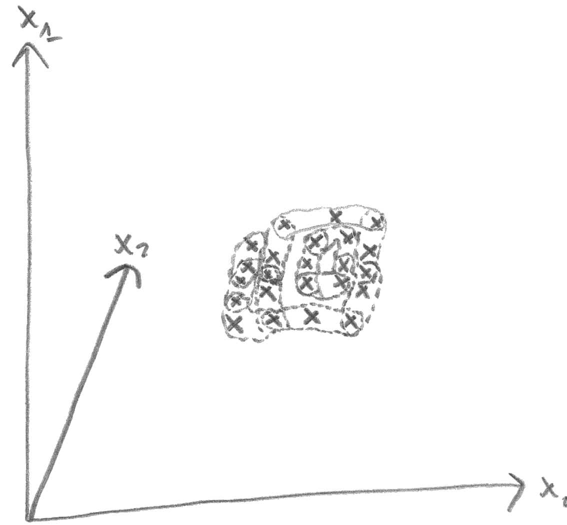


Figure 3.4.: An illustration of the time-lagged or time-delayed variables. Each datapoint is concatenated with the consecutive ones in its time series, here illustrated drawn dotted lines grouping together triples of points. The aggregated distances between triples are then used as the pairwise distances needed for diffusion maps.

3.3.5. Closed observables

Another way of incorporating the dynamics is by expanding each datapoint through the *time-delay-embedding*, that is, incorporating a whole time-series as one datapoint. This way, the distance metric compares differences between whole trajectories rather than single points in time. The theorem behind this, *Taken's theorem* [51, 52], states that observations of a dynamical system are enough to reconstruct the dynamics of it. Berry et al., in [53], demonstrated the efficiency of using diffusion maps exactly for extracting macroscopic variables of dynamical systems. In a paper by Dietrich et al. [1], this was used to build an approximate, low-dimensional version of the dynamics, with the low-dimensional approximate dynamic variables referred to as *closed observables*, as they should be approximated using so much dynamic information that the system is closed on the manifold.

The closed observables are constructed in a three-step process:

- *Prerequisites.* We assume that we have some type of dynamical system (such as described in section 2.1.1), of which we can gather observations $\boldsymbol{y}(t)$ at times $\{t\}$. These can be state or control variable values, or functions thereof. We also have a set of starting variable values $\{\boldsymbol{x}_0\}$, and a set of parameters $\{\boldsymbol{p}\}$, which can be system dynamic parameters, or information about future events. We gather data from the system, so that we have series of observation values $\boldsymbol{y}_k(\boldsymbol{p})$, for different parameters $\boldsymbol{p}$ at a series of timepoints t_k. The t_k are taken to be equidistant and in increasing order, for simplicity.

- *Construction of the time-lagged manifold.* For each starting point and each parameter value, we construct time-lagged variables $\boldsymbol{h}_k(\boldsymbol{p})$ by concatenating weighted observa-

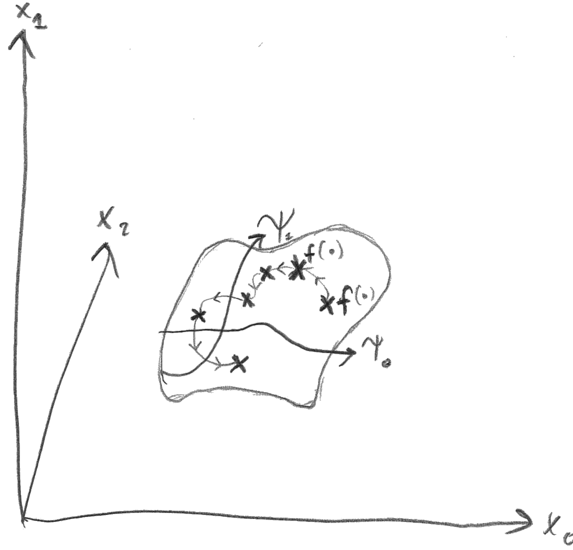


Figure 3.5.: Utilization of the diffusion maps coordinates. To utilize the diffusion maps coordinates in the closed observables framework, three interpolations are necessary. First, one needs to interpolate from the original coordinates and parameters (x_i) to the diffusion maps coordinates (ψ_i). Then, in order to approximate the dynamics, one interpolates the function going from one datapoint's diffusion maps coordinate to the subsequent one (following the arrows). Lastly, values of the functions or variables that are desired are interpolated from the diffusion maps coordinates ($f(\boldsymbol{\psi})$).

tions $e^{-i\kappa}\boldsymbol{y}_{k+i}(\boldsymbol{p})$ for T consecutive timepoints (see figure 3.4)

$$\boldsymbol{h}_k(\boldsymbol{p}) = \left\{\boldsymbol{y}_k(\boldsymbol{p}), e^{-\kappa}\boldsymbol{y}_{k+1}(\boldsymbol{p}), e^{-2\kappa}\boldsymbol{y}_{k+2}(\boldsymbol{p}), \dots, e^{-(T-1)\kappa}\boldsymbol{y}_{k+T-1}(\boldsymbol{p})\right\}$$

T is chosen large enough to capture the dynamics of the system by Taken's theorem, and $\kappa \geq 0$ is chosen appropriately small enough to weigh the more current observations higher, but large enough to not cut off the last values, such that enough points are included. From each starting point and parameter combination, we can thus construct multiple $\boldsymbol{h}_k(\boldsymbol{p})$, as long as $T < \max_k t_k$ (or $\ll$), as the points are equidistant, i.e., t_{k+1} can be seen as a starting point for another T values with the same spacing in t.

- *Construction of the closed observables.* The closed observables are constructed as the condensed representation of the time-lagged variables $\boldsymbol{h}_k(\boldsymbol{p})$, by using the diffusion maps method (see section 3.3.1 above for the method). Using the euclidean metric distance between points, the trajectories are then compared, which by diffusion maps should extract the underlying variables of the dynamics (optimally, as shown in [53]).

- *Dynamics estimation.* By a three-step interpolation, the dimension-reduced variables ψ_i from diffusion maps can be used to approximate the dynamics of the system. Firstly, the relation between starting values and parameters with the diffusion maps coordinates $\boldsymbol{\psi}(\boldsymbol{x}_0, \boldsymbol{p})$ is interpolated through a function $\boldsymbol{\psi}_I(\boldsymbol{x}_0, \boldsymbol{p})$. Secondly, the dynamics in the low-dimensional coordinates $\boldsymbol{G}(\boldsymbol{\psi}_k) = \boldsymbol{\psi}_{k+1} - \boldsymbol{\psi}_k$ is interpolated by a

function $\boldsymbol{G}_I$. Finally, the outputs $\boldsymbol{y}_k(\boldsymbol{p})$ are interpolated by a function $\boldsymbol{y}_I(\boldsymbol{\psi}_k)$. This is illustrated in figure 3.5.

So what is the benefit of this process and three-step interpolation compared with for example interpolating the dynamics on the original variables? As shown in [1], as long as the dimension of the system in the diffusion maps coordinates is lower than that of the original data, the storage requirements for interpolants is heavily reduced. A factor in the order of 10^5, or approximately "the difference between the memory of a smartphone and a supercomputer", is mentioned for an example in the paper.

3.4. Mathematical and computational methods

In the framework of diffusion maps and closed observables, there are many ways to implement several of the parts, and some require special consideration.

3.4.1. Parameter reduction

In order to be able to use the Closed observables algorithm for predicting control outputs, we need to do create the function $\boldsymbol{\psi}_I(\boldsymbol{x}_0, \boldsymbol{p})$ to interpolate from our state variables to the diffusion map coordinates. However, as with much regression, having too many inputs that are not relevant makes our interpolation likely to overfit to irrelevant data. Also, including irrelevant parameters reduces performance with unnecessary calculations, as the volume of our space we are interpolating from grows exponentially the dimension of our input. Thus, it is necessary to reduce the dimensionality of our input by finding the relevant state variables. Of course, we could try and mitigate this problem by sampling cleverly, using for example *sparse grids* [54] (described below), but this would require setting up a sampling grid for maximal efficiency.

This is the classical problem of *feature selection* and *feature extraction*. Since we expect the diffusion maps manifold to be highly nonlinear, we can forgo the classical approaches extracting linear features such as PCA. A simple method that however finds locally linear features is *local regression* [55], which allows us to fit for example a linear function locally, and allow it to change over the dataset. By greedily selecting the features with most explanatory power, or deselecting those which produces the least error when removed, we can obtain the "best" state variables using this.

3.4.2. Sampling

Since we we calculate the diffusion maps coordinates with sampled data, the resulting eigenfunctions and parametrisations we obtain from the method are also only available to us as the values sampled at the same points. In order to have a representation that we can also use for new, unsampled starting and intermediate points, we need to interpolate the results between the samples.

Additionally, whenever we want to use the result from the diffusion maps method, such as when simulating the reduced dynamics and obtaining outputs in the Closed Observables method, we only have this at sampled points as well. If we want this at unsampled points, we need to interpolate again.

Regardless of the way we interpolate and sample, unless we have very specific knowledge of the problem at hand, the interpolation becomes harder the more parameters, or dimensions, that can influence the result. This is referred to as the *curse of dimensionality*.

The same thing applies to our problem of interpolating a (multivariable) smooth function from a multidimensional input space. There are however many ways of mitigating this problem. For example, there are several ways to sample the input domain, where we might be interested in some areas more than others. For the first, areas we have already sampled are unnecessary to sample more (unless there is also uncertainty which needs to be sampled), and there might be some combinations of inputs that are invalid or unreachable in our scenarios, which we thus do not need to sample.

In our test case of approximating the dynamics of the MPC problem in section 2.3.1, the most naive way of sampling would be to simply let the MPC implementation run along the gathered parameter (height) data, and hope that this would adequately sample the whole relevant input space. Since we can control the starting points for the MPC implementation, we could however try different strategies to gain efficiency.

For example, we could try and analyse the space of input parameters, and uniformly sample this space, eliminating the many doubled configurations where the inputs are very similar to each other. Additionally, to make sure we would sample the whole relevant space, we could systematically sample the relevant configurations of the dynamic variables, without having to rely on the dynamics to explore them for us.

An interesting approach for this was explored in [46], where diffusion maps were utilized to find the relevant parameters to the dynamics, and after a run, the parameter sets lying closest to the boundary of the approximated manifold were found, and extrapolated "outwards" using local PCA.

Another method that allows sampling the whole space without having to sample all dimensions fully is using *sparse grids* [54], that build a structured grid that allow for less sampled points, while retaining information for interpolating the solution. This allows, for the same level of error, to have $\mathcal{O}(N\,(\log N)^{p-1})$ sampled points, where N is the number of sample points along each dimension, and p the dimension, instead of $\mathcal{O}(N^p)$ points for a full grid.

3.4.3. Interpolation methods

As we need to use interpolation for the closed observables method, we go through a few methods that could be used.

Nearest-neighbour interpolation approximates the value at a point by, as the name implies, the value of the nearest neighbour. Although this is very simple to implement and to calculate, it only gives piecewise constant approximations, that thus do not use any of the regularity of the solution. The accuracy of the method roughly corresponds with distance between points, so the error for the simple method scales as $\mathcal{O}(N_{\text{tot}}^{-1/p})$.

Linear interpolation approximates the value at a point $\boldsymbol{x} = (x_0, x_1, ...)$ by a, usually only piecewise, linear function whose parameters are decided by the sampled points. The most common approach is to sample the points on a full Cartesian grid, and then weight the contribution of each point based on the position of the point within the grid cell of the point. However, linear interpolation can be applied on any type of grid with some modifications to get the best approximation on that type of grid.

Sparse grids can for example be used for linear interpolation with a hierarchical combination of basis, and other piecewise linear approximations may be done that use the regularity of the function to approximate better. For example, local PCA might provide better approximations of linear models valid for a small region, for example providing a linear model based on the k nearest neighbours, or points within some close region, based on another measure.

Various other variants of interpolation also use the nearest neighbours, but then interpolate based on some other criterion, as "local" variants of the algorithms.

Geometric harmonics [42] is an extension method closely related to diffusion maps, which extends functions defined on a set $\mathcal{D}$ to a larger domain $\bar{\mathcal{D}} \supset \mathcal{D}$, using parts of the diffusion maps framework. This setting is typical for the diffusion maps setting, where $\mathcal{D}$ would be the sampled datapoints and $\bar{\mathcal{D}}$ could be the manifold it is sampled from, or a superset of it.

First, the diffusion maps eigenfunctions $\boldsymbol{\psi}_i$ are extended to $\bar{\mathcal{D}}$ by letting their value be an average of that of the nearby data, weighted by the diffusion maps kernel,

$$\bar{\boldsymbol{\psi}}_i(\bar{\boldsymbol{x}}) = \frac{1}{\lambda_i} \int_{\mathcal{D}} k(\bar{\boldsymbol{x}}, \boldsymbol{y}) \boldsymbol{\psi}(\boldsymbol{y}) \, \mathrm{d}\mu(\boldsymbol{y})$$

with $\bar{\boldsymbol{\psi}}_i, \bar{\boldsymbol{x}} \in \bar{\mathcal{D}}$. Then, the function is built up using the same coefficients as on the diffusion maps eigenfunctions on the original data, but on the extended basis functions

$$f(\bar{\boldsymbol{x}}) = \sum_i \langle f | \boldsymbol{\psi}_i \rangle \, \bar{\boldsymbol{\psi}}_i(\boldsymbol{x})$$

where $\langle \cdot | \cdot \rangle$ is the standard inner product on $\mathcal{D}$, $\langle f|g \rangle = \int_{\mathcal{D}} f(\boldsymbol{x}) g(\boldsymbol{x}) \, \mathrm{d}\mu(\boldsymbol{x})$.

3.4.4. Large eigenvalue computations

As part of the diffusion maps process, an eigenproblem needs to be solved. For a very large amount of datapoints, such as might be needed to fully approximate the whole nonlinear dynamical system, this eigenvalue computation might be too large to be handled by a naive eigenvalue solution. As computing eigenvalues and eigenvectors of an $n \times n$ matrix is typically an $\mathcal{O}(n^3)$ process, there might be some necessary steps to take in order to be able to calculate the solution for large n.

For the first, we realize that the eigenvalue problem can be solved by calculating the eigenvalues of a symmetric matrix (see section 3.3.1).

For the second, while calculating the distances between datapoints, there should be enough points that we deem to far away from each other to directly influence each other in the computation (such that there is no chance for the diffusion process to diffuse between these points). If we set the distance between them to infinity, or respectively the chance to diffuse between the points to zero, the diffusion matrix should turn sparse, if we truly have a low-dimensional manifold in a high-dimensional space, and we have chosen our diffusion length right. Thus, we might be able to use algorithms for finding eigenvalues of symmetric sparse matrices.

In addition, in the case of diffusion maps, we are only interested in the eigenvectors which carry the most significance, which can be found among those with the largest eigenvalue magnitudes. This means we can reduce our search to finding the k largest eigenvalues, where $k \ll n$.

One such algorithm that can do this efficiently is the *Lanczos algorithm* [56], which iteratively builds up a *Krylov* subspace of a random starting vector, and finds the approximate eigenvectors within. It relies on matrix–vector multiplications to build up the subspace, and requires significantly less computation, for most practical cases $\mathcal{O}(kn^2)$. It has many variations. There exist many software libraries that implement these, for example the *ARnoldi PACKage* (ARPACK) [57], and its recent continuation with parallel extensions. ARPACK is included in for example MATLAB, where it is used in the eigenvalue solving function `eigs`.

Part II.

Methodology

4. Algorithm Overview

Just as the theory behind this thesis spans several fields, the implementation of the methods described in Part I also spans several separate parts. So, in order to not lose track of the big picture, we here introduce the whole algorithm in an overview.

4.1. Schematic diagram

Starting from a scenario such as that in section 2.3, we create a dynamical model of the system that encompasses the necessary features. Following the notation in section 2.3.1, we identify the dynamics $\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u})$ of the system, and the constraints $\boldsymbol{r}$ and $\boldsymbol{d}$, and a set of parameters $\boldsymbol{p}$. With this, we proceed to choose a cost function C and optimize for this according to Model Predictive Control, described in section 2.2, choosing necessary discretization parameters and prediction horizon lengths. The time-series data generated for states and optimal controls here is then passed on for analysis by diffusion maps, described in section 3.3. An approximation of the optimal controls is calculated, using the underlying low-dimensional representation of the dynamics obtained. The representation is then used to generate a low-dimensional controller by interpolative methods. Finally, the efficiency of this controller is then validated in the validation framework described in chapter 7.

This overview is summarized in figure 4.1 with references in table 4.1, with a more graphical exposition sketched in figure 4.2. The following chapters will go into more detail on the process. Source code can be found in Appendix C.

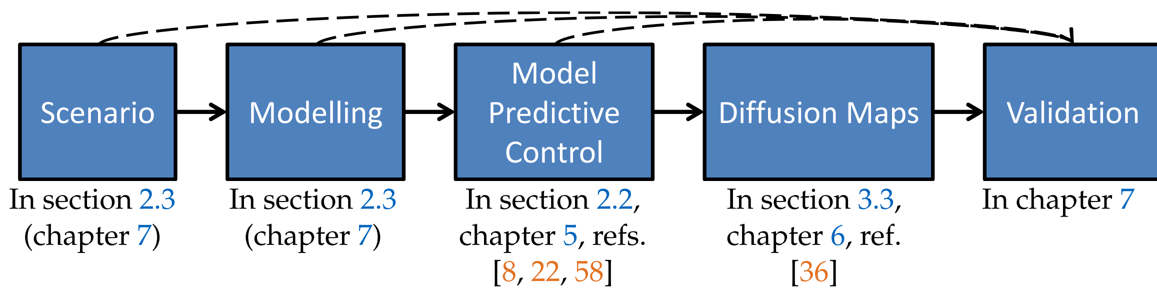
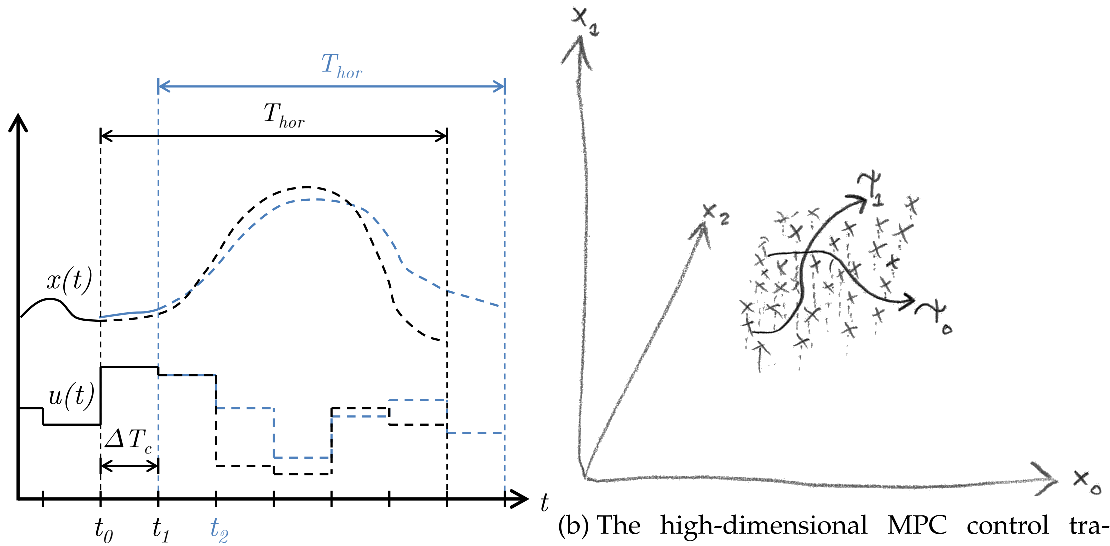


Figure 4.1.: An overview flowchart of the algorithmic process described by this thesis. See text and table 4.1 for more details.

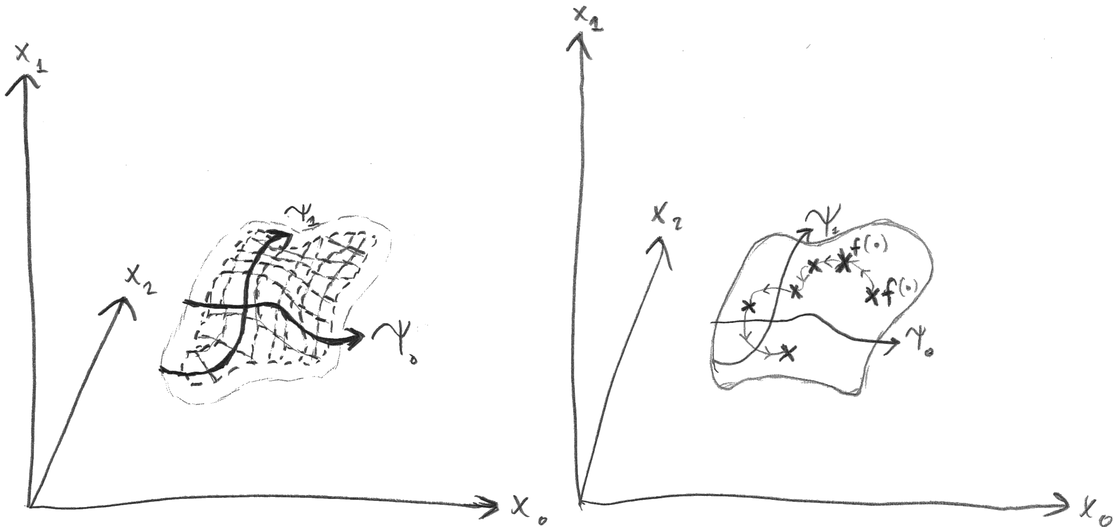
Table 4.1.: The algorithm–relevant subparts of this thesis. See text and figure 4.1 for an overview of the algorithm.

	Theory	Implementation	References
Scenario and Model	Section 2.3	Chapters 5 and 7	
MPC	Section 2.2	Chapter 5	[8, 22, 58]
Diffusion maps	Section 3.3	Chapter 6	[36]
Validation	(Chapter 7)	Chapter 7	



(a) The scenario of choice is modelled, and optimised control values are calculated using Model Predictive Control (section 2.2).

(b) The high-dimensional MPC control trajectories are parameterized on a lower-dimensional manifold by the using diffusion maps, resulting in sampled values for diffusion map coordinates ψ_0 and ψ_1 (section 3.3.1).



(c) Functions between system variables and diffusion maps coordinates as well as describing the diffusion maps coordinates' dynamics are interpolated to non-sampled points to create functions approximating the optimal control (section 3.4.3).

(d) The interpolated approximative functions ($f(\cdot)$) are used to construct a controller that controls the system optimally. The controller is validated in the validation framework (chapter 7).

Figure 4.2.: Illustration of the steps in the algorithm.

5. Practical Model Predictive Control

Just like with any algorithm, the implementation of MPC requires special considerations. In this chapter, the measures taken and the specific implementation of the theory described in chapter 2 are described and discussed. The relevant source code is found in Appendix C.1.

5.1. Height data acquisition

In order to be able to generate realistic scenarios for our model system, we need data from the real world. For the variation of height along a road, there are data to be downloaded from the internet, with varying limitations on its use. For the purposes of this thesis, height data from the Google Maps Elevation API [29] was deemed most useful, since the data is accurate enough, has very good coverage, and above all, is free to use for research and development purposes. In addition, it has a useful API for scripted downloading of data.

The routes that were chosen as scenarios were taken between appropriately situated cities in Germany, Austria, Switzerland and Italy (see table 5.1), with both hillier and flatter parts. The elevation data was then gathered at GPS locations for the roads retrieved from Google's navigation services, split up to provide desired resolution among the points.

A limitation of this approach was that the elevation retrieved ignores the existence of tunnels through for example mountains, instead producing values for the ground level vertically above the tunnel. For roads in many mountainous parts, the end results are thus mostly relatively smooth height profiles, suitable for motorized vehicles, interspersed with parts of extreme height differences up mountain sides. As such height profiles might be infeasible to drive over at all, and also are not relevant for real scenarios, roads with too extreme slope data were filtered out not to be used for simulation.

Destination cities			
Berlin	Frankfurt	Konstanz	Salzburg (Austria)
Cologne	Freiburg	Munich	Bolzano (Italy)
Dresden	Hamburg	Stuttgart	Lugano (Switzerland)
Erfurt	Hanover	Innsbruck (Austria)	Zürich (Switzerland)

Table 5.1.: The different cities between which routes were selected as input to the MPC algorithm. Routes including unrealistically high slopes were excluded.

5.2. Model Predictive Control in pyomo

The module pyomo, as described previously in section 2.2.2, is highly versatile and flexible for defining optimization models. The building blocks of a pyomo model are the same as those used in mathematical language, including sets, parameters, variables, expressions, constraints and objectives, and allowing us to translate the MPC problem of our test case (see section 2.3) with relative ease into something that pyomo can use. They are defined as python classes, allowing the use of python's powerful capacities for data input/output and processing.

The module pyomo is available through the python package index, for example using the tool pip [59]. Version 5.2 of pyomo was used during development with python version 3.5.2, as well as numerical library numpy, version 1.1.1 and scientific library scipy, version 0.17.0.

As mentioned in section 2.2.2, pyomo interfaces with a back-end solver to solve the optimization problems, providing an interface for solver options. For Ipopt, the data transfer proceeds through an intermediate file in the .nl format. To the untrained eye, this looks mainly like assembler code, but it allows for limited amounts of proofreading, in addition to pyomo's own functions.

5.3. Model Predictive Control code organization

The code run the MPC algorithm with pyomo is split up into functional parts, each separated into a python class to facilitate modular decoupling and allow easy extendibility and reusability. The parts consist roughly as follows: a model utility class VehicleModel, in which the model is defined and the interactions with the pyomo model are centralized; a parameter class Params, in which parameters are read and processed; a solver utility class SolverUtil, handling the interaction with the solver; an output class interface OutputInterface, from which subclasses can handle specific desired outputs and plots; and a runner class SimulationManager, which coordinates the simulation and handles initialization and stopping.

The main algorithm, as handled by SimulationManager, can be found in run.py. An outline is given as follows, followed up with a closer explanation of what is going on

in each part.

Algorithm 1: Optimal control strategy generation with MPC

```
input  : Simulation parameters; Height data tabulated with distance
output: Time-series data of a system controlled with MPC
(1) build up and discretize model; initialise values;
(2) fix initial conditions;
    // Main loop
    while distance < end distance do
        if not first iteration then
            increase global distance counter by ΔX_c;
(3)         shift previous solution by ΔX_c to provide initial guess for next solution;
        end
(4)     load and fix height data for current distance grid points;
(5)     initialise derivative variables to their current values;
(6)     solve optimisation problem for all variables;
        foreach output object in registered outputs do
(7)         save or print data from current state;
        end
    end
    foreach output object in registered outputs do
(8)     do final output;
    end
```

Lines 1 and 2, Initialization: The parameters, from a command-line specified `.json` parameter file, are loaded and used to initialize the parameter class `Params`. From these parameter values, accessed through `Params`, the main model class `VehicleModel` (subclassing `BaseModel`, which has basic functionality for extension) is initialized. `VehicleModel` wraps the `pyomo` model, and initializes all the `pyomo` model components: sets, parameters, variables, expressions, (differential equation-) constraints, integrals and objectives (classes `Set`, `Param`, `Var`, `Expression`, `Constraint` and `Objective` of the module `pyomo.core.base` and `Integral` and `DerivativeVar` of module `pyomo.dae` respectively). This is done through specalized `BaseModel` methods, named `_create_variables` and similarly.

The discretization is done through the `pyomo.dae` module.

Lines 3 to 5, (Re-)Setting up: Before solving, the appropriate height values must always be loaded to the model. This is done through methods on the `Params` class, which interpolate and optionally smoothen the height data based on the neighbouring distance–tabulated height values from the height file specified in the parameter file.

After all state variables are initialized, their distance derivatives are also initialized consistently, so that the differential constraints are not violated already at the beginning, thus making the initial guess closer to a correct solution.

Because MPC iteratively solves the same problem with slightly different parameters, it is also very beneficial to keep parts of the old solution as an initial guess to the next. To do this, not only the boundary conditions (at the beginning distance boundary) are shifted with the horizon, but also all the variable values. Since we define our grid to be equidistant, this corresponds to simply letting every variable `a[i] = a[i+Δ]`, for a variable `a` indexed by `i`, where Δ is the number of distance grid intervals corresponding to the distance shift. The last variable values of course have to be initialized to something else, for example the values at the very last grid point in the old solution.

Line 6, Solving: To solve the optimization problem, the class `SolverUtil` wraps the pyomo solver interface.

Lines 7 and 8, Saving and printing output: The interface `OutputInterface` roughly follows the observer in the Model-View-Controller pattern, in the sense that it provides a method to be notified by the controller `SimulationManager` when there is new data in the model `ModelUtil`. Provided are two implementations which register to `SimulationManager`, `SimpleOutputUtil` which prints running status messages, and `SavingOutputUtil` which saves timeseries data and plots or writes it to file after simulation.

5.4. Parameter and method choices

For many of the expressions and equations in section 2.3, parameter values need to be supplied. For some, such as vehicle weight, air drag coefficient, maximum thrust produced by the motor etc., physical justifications or approximations might suffice. For others, such as horizon length, coarseness of discretization and cost weights, physical parameters may provide guidance, but will require careful testing to turn the system into one producing interesting behaviour. In order to provide a uniform sample for comparison, all simulations were done with the same parameter set, which can be found in table A.1 of Appendix A.

For the derivative discretization, a forward finite difference scheme was used, as this turned out to give the fastest convergence with the same accuracy as other schemes. The horizon length and discretization step numbers were found by setting them low enough that it allowed a quick solution, but high enough to be accurate. This was determined by comparing with results from choosing a doubly large horizon length or twice the number of discretization steps.

For the cost function, a weighted sum of the costs presented in section 2.4.1 was chosen, balancing the weights such that the optimal behaviour similar to the heuristics would show, but velocity would not plummet for every uphill, and accelerations would be smooth, but not completely flat.

Some example results are shown in figures 5.1 and 5.2, both along height profiles that are computer-generated (figure 5.1) and from real height data (figure 5.2). There, some examples where MPC produces significantly predictive behaviour can be seen.

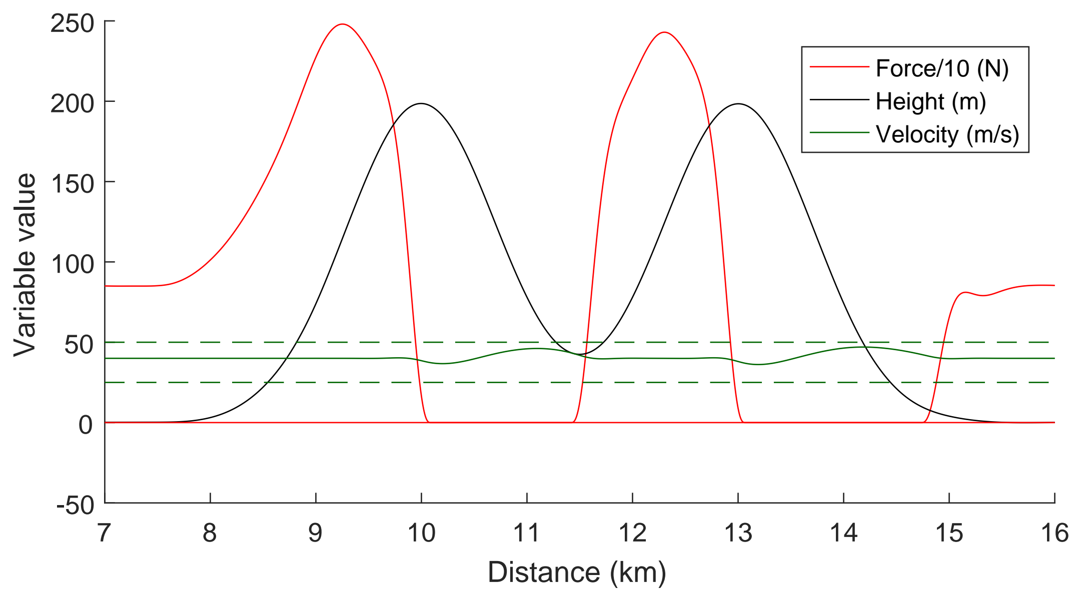
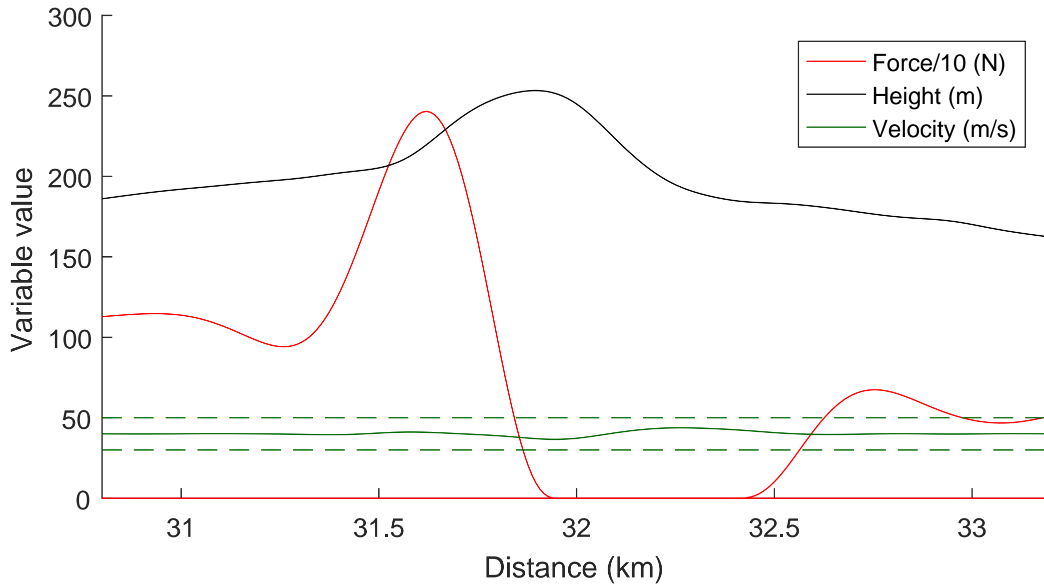
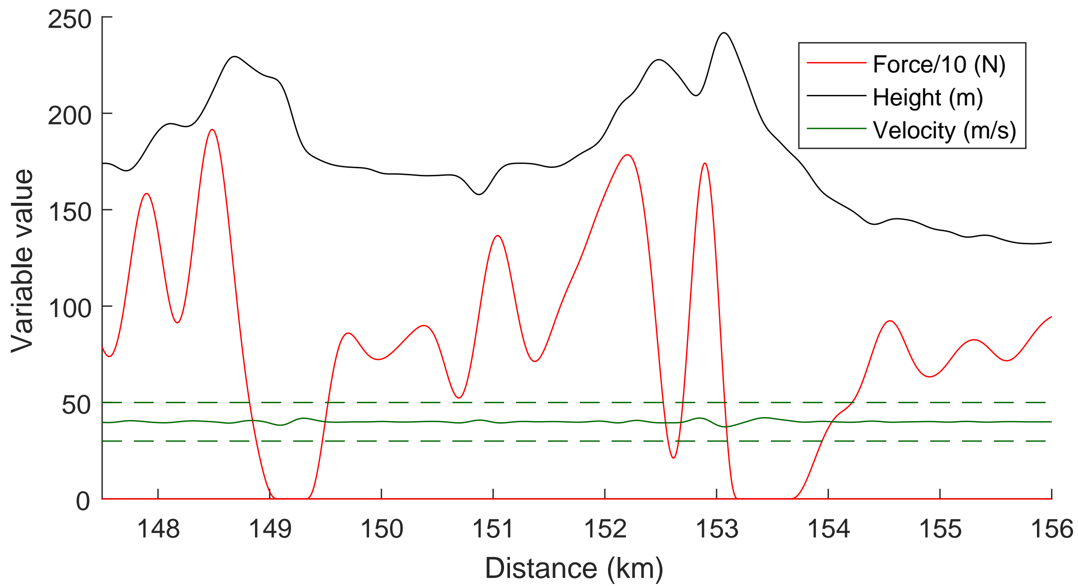


Figure 5.1.: The system's response to two successive hills made up by Gaussian curves of standard deviation 1000 and height 200, with centres 3000 apart.



(a) An example where predictions can save energy. Even before the peak (at approximate distance 32000), the motor force is dropped, resulting in a lower velocity. Velocity is regained as the vehicle rolls down the slope, the motor only reactivating when the slope is too flat to keep the vehicle going at the desired speed. Additionally, no braking is needed to stay below the upper velocity threshold.



(b) More example output where the predictive power can save energy. Map data ©2017 GeoBasis-DE/BKG (©2009), Google.

Figure 5.2.: MPC results for the distance between Stuttgart and Zürich. Map data ©2017 GeoBasis-DE/BKG (©2009), Google.

6. Diffusion Maps Framework

In order to apply the diffusion maps method described in section 3.3.1, there are several practical considerations that need to be made. This is the case whether one chooses to use the diffusion maps framework directly to interpolate function values, as described in section 3.4.3, or when using the closed observables framework described in section 3.3.5. Firstly, as with most data analysis, the results are much improved by some pre- and post-processing of the data, such as reducing data to the most relevant input. Secondly, the diffusion maps algorithm itself leaves several different implementation options, for example the choice of similarity metric and kernel scale. Finally, there is much to consider in the numerical parts of the algorithms themselves. Below, we will try to cover each of these areas and more, that was important in implementing the diffusion maps algorithm. Source code can be found in Appendix C.2.

6.1. Overview of implementation

We start with a condensed overview, roughly following figure 6.1:

- *Prerequisites*. Simulation data with the desired dynamics, i.e. controlled optimally by MPC, organized so that it is retrievable in *a)* contiguous time series and *b)* according to the values of scenario parameters.
- *Closed variables*. From the chosen output/control variable(s), closed observables are constructed as described in section 3.3.5.
  - First, time-lagged variables are created by concatenating temporally subsequent values for each datapoint.
  - Then, the standard diffusion maps algorithm is run on the time-lagged variables as described in section 3.3.1.
  - The variables most relevant for tracking the diffusion maps coordinates are selected, as outlined in section 3.4.1.
  - Three different interpolating functions are created: one interpolating from the original datapoints with the relevant parameters to diffusion maps coordinates, one forming the dynamics by interpolating from the diffusion maps coordinates of each datapoint to that of the next, and one that interpolates from the resulting diffusion maps coordinates to the relevant output.

6.2. Preprocessing

In the scenarios run using the MPC algorithm described in chapter 5, some states and behaviours were sampled more than others, as expected for real data. For example, there

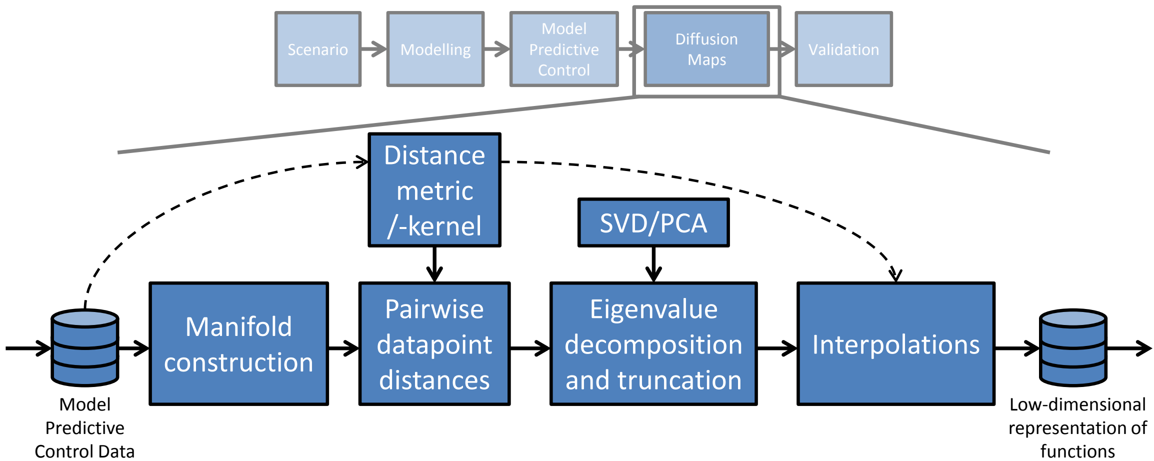


Figure 6.1.: An overview of the algorithmic framework for approximation of the MPC results using diffusion maps.

is more road corresponding to flat or only negligibly inclined distances than every different kind of hill. Thus, the different types of desired behaviours were very unevenly sampled.

As the diffusion maps algorithm relies on parameterizing the differences between the behaviours, it does not behave optimally when most data is the same, even when normalized for sampling density. In many cases, an extreme behaviour may be seen so different from the rest that the diffusion maps coordinates only characterize this one behaviour. This isolation of a behaviour can be partly alleviated by making the diffusion maps kernel size bigger, as this scales down the distances between points. The downside to this treatment is that it also makes the other points seem more similar, and the method might therefore not be able to discern between different behaviours among close points. A promising solution to this may be *variable-width kernels* [60], where the kernel size is varied depending on the sampling density, in an iterative process. This results in wide kernels where the sampling density is low, and narrow kernels where the sampling density is high. Thus, both patterns giving rise to large and small differences may be parametrized. This approach will be target for future work.

A part solution to this overprioritization of extreme points is to filter out the possible extremes, in our physical scenario characterized by trajectories including extreme slope values. However, here one needs to be careful; we also want to keep as much information as possible, and filtering out unusual behaviour reduces the difference in the cases included.

Another issue caused by the concentration of similar datapoints together is that the processing of the large amounts of data needed to sample the space starts requiring more memory than that of a standard computer. With datasets of roads being up to 1000km long, and datapoints being spaced at under 50m, one easily has 20000 datapoints just for a single dataset. A similarity matrix between these points then already has $4 * 10^8$ entries, which for double precision (standard in many MATLAB algorithms) corresponds to 3.2GB

memory. This rises quadratically with number of datapoints, so combining all the available data becomes trickier. Although one may assume that many entries in a similarity matrix should be close to zero (for points dissimilar to each other) and thus ignored, this might not always be the case if the kernel size is big, for example due to the reasons above.

As overly subsampling the data may lead to low-density–region datapoints being too far away from each other, another approach was used, iteratively removing the datapoints having closest neighbours, thus deemed least important for the overall structure, with the motivation of this allowing to keep points in low-density regions while still subsampling the regions where the datapoints are much the same, for an overall more uniform density. The algorithm, found in the function `prune_closer_points` in `prune_output_matrices.py` proceeds as follows:

Algorithm 2: Greedy datapoint subsampling

input : an array of datapoints; a similarity measure
output: selection with specified number of subsampled points

while *selected number of points > required* **do**
  find one datapoint with smallest neighbour distance to a still selected point;
  remove this datapoint from selection;

The specific implementation uses a python Priority Queue to keep a list of the entries with the smallest neighbour distances, updating them lazily when the neighbour with the closest distance is already removed from selection.

6.3. Code structure

The implementation of the diffusion maps algorithm was done in MATLAB for fast prototyping and visualisation purposes.

The main diffusion maps algorithm is found in two versions, one using a single, contiguous time series from one simulation run (`dmapsTesting.m`), the other preprocessed, filtered and merged time-lagged points (see section 6.2) from multiple runs (`dmapsTesting_timlag.m`). Below follows the outline for the single-file version, with file names of functions in comments.

Algorithm 3: Diffusion maps with closed observables

input : Time-series data
output: Diffusion maps eigenvalues and eigenvectors

```
// Here starts the main algorithm
construct time-lagged manifold;                              // constructTimeLagged.m
compute pairwise distances between trajectories;             // makeDistMatrix.m
compute kernel width from nearest neighbour distances;
construct kernel matrix;                                     // constructKernelMat.m
normalize kernel matrix;                                     // normalizeKernel.m
calculate diffusion maps eigenpairs;                         // constructDMaps.m
```

To construct the interpolating functions of the closed observables framework, MATLAB built-in functions `scatteredInterpolant` and `fit` were used, since these allowed

quick prototyping and implementation. With `scatteredInterpolant`, one can interpolate from up to three-dimensional data, using *natural neighbour interpolation*, which is G^1 smooth. With `fit`, one can also fit output with smoothening using e.g. LOESS [55], which is beneficial for us when points near each other have differing values. The `fit` function however only accepts two-dimensional input, which again limits its usability.

We will then use our three selected state variables and use `scatteredInterpolant` to interpolate a function from them to the diffusion maps coordinates. We then interpolate the dynamics on the manifold, if it seems two-dimensional using `fit` with LOESS, if it seems tree-dimensional with `scatteredInterpolant`. We then use the same method to calculate the output control variable.

Putting these three interpolation functions after another gives us the required controller, i.e. we can use it to find the upcoming control value using the current state. The results from this can be found in chapter 8.

7. Validation

In order to validate the results of the MPC framework described in chapter 5, as well as those for the diffusion maps approximation described in chapter 6, a framework was developed in Matlab to independently simulate the scenario using the obtained control values and compare the results.

7.1. Matlab implementation

Using the same equations, the test case scenario of a car on a hilly road, as described in section 2.3.1, was implemented in MATLAB. To make the validation closer to a real-world scenario, the MATLAB simulation uses a more accurate integrator, the standard `ode45`, which uses an adaptive step-size explicit Runge-Kutta method of 4th order to integrate the system of ODEs.

In order to incorporate the control calculated by the MPC framework, the saved control values are loaded and used for the simulation, such that when the ODE is integrated in the distance interval $[x_i, x_{i+1})$ corresponding to one control value F_i, that same value for F_i is used, thereby tabulating the control F as a function of distance x.

As a naive reference controller, simple hand-tuned PID-controllers were used. The scenarios used for calculating the MPC controls were simulated for these controllers as well, and the total cost over the distances were compared. Some results from this comparison can be seen in table 8.1, with the full results in Appendix B.

In addition, controllers created using the diffusion maps with closed observables framework were compared with the MPC- and PID- controllers.

A drawback from the comparison in MATLAB is that the derivatives required for the acceleration and motor thrust smoothness costs are harder to evaluate. Thus, they were ignored for the sake of comparison.

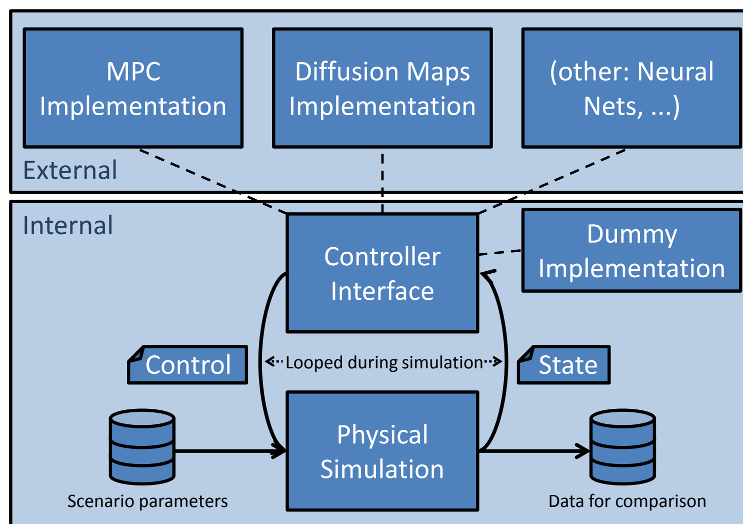


Figure 7.1.: A schematic diagram of the validation framework. The framework is designed such that different control schemes can be used for the same scenario, in order to compare things such as runtime and performance metrics.

Part III.

Conclusion

8. Results and Discussion

In this chapter, we look at and discuss the results obtained from the different parts.

8.1. Model Predictive Control results

The MPC algorithm controls were compared with those of a simple hand-tuned PID controller. As can be seen in table 8.1, the total integrated cost was significantly lower for the MPC controller, especially for those roads that are expected to contain hillier parts. This shows the cost benefits of the MPC algorithm managed to more cleverly plan for changes in load due to inclination, as seen in figures 5.1 and 5.2.

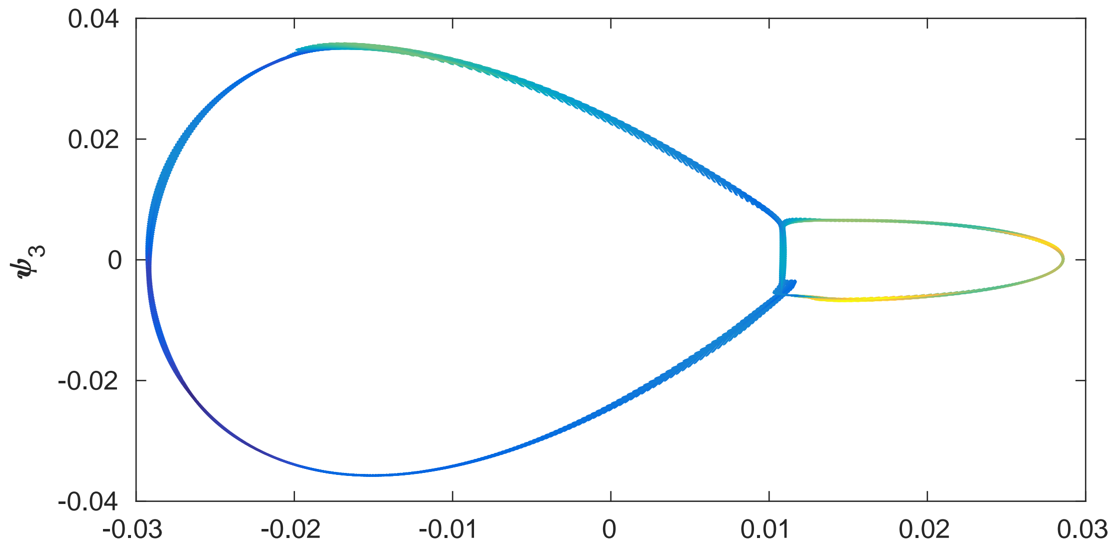
8.2. Diffusion maps results and discussion

Plotting the diffusion maps coordinates against each other, one can see patterns in the dynamics. In figures 8.1a and 8.1b, arrow plots of the gradients were made, colored and sized by magnitude, and one can see a clear spiraling pattern. The pattern is not entirely contained in two dimensions, but might be possible to use for control after smoothing. In figure 8.4, a diffusion maps controller was implemented, and was able to steer along a height curve consisting of five sinusoidals with semi-random coefficients, showing similar energy–saving behaviours to the MPC controller.

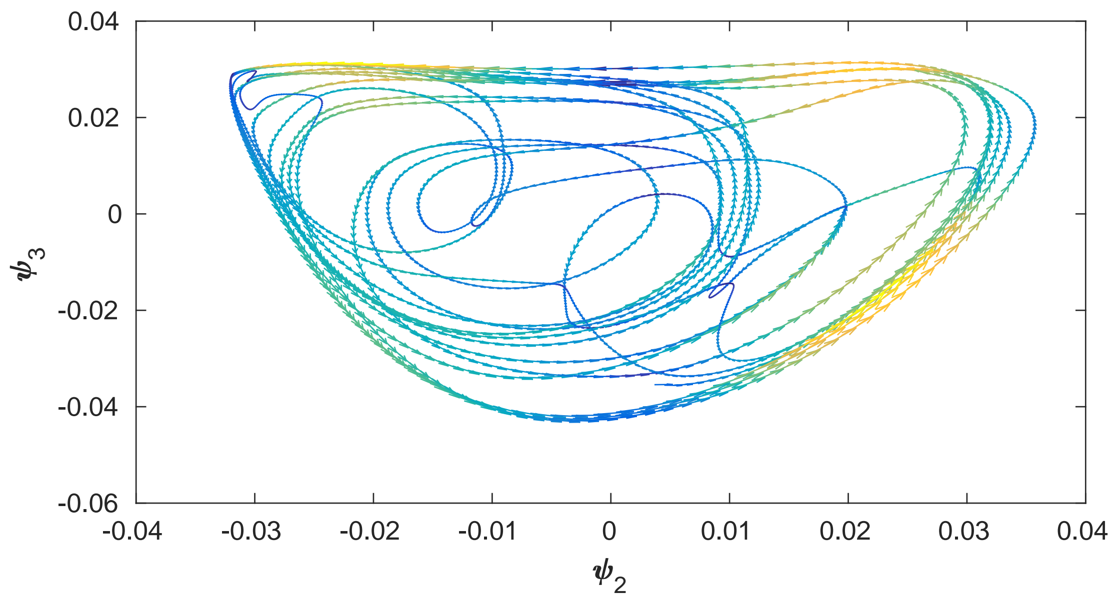
To see if this was generalizable to real data, a similar controller was generated from diffusion maps eigenvectors based on the data between Konstanz and Innsbruck. The controller was then tested on the distance between Konstanz and Innsbruck, which has the first few hundred meters in common with the previous road, but is then different. A small stretch of the simulation can be seen in figure 8.5a, where energy–saving patterns similar to the ones from MPC are seen. Looking at table 8.2, we can see that the cost obtained was actually lower than that of MPC, which can have a number of explanations, the main one most likely being the influence of the acceleration- and motor thrust smoothing costs present in the MPC optimisation but unaccounted for in the tables above. The same controller on an artificial sinusoidal curve performs much worse, as seen in section 8.2.

The worse performance on other types of curves also indicates that the features used to interpolate to the diffusion maps coordinates might not be optimal. In order to find better ones, we might either need to use more variables, or combine the features in another way, i.e. use other types of interpolation. As the current implementation is limited by MATLAB's `scatteredInterpolant` and `fit` functions, with 3 and 2 number of features maximum respectively, we conclude that switching to other methods, such as those outlined in section 3.4.3 might improve performance.

One can also see that the training data heavily influences the results, since the controller trained on the road between Bolzano and Innsbruck performs worse.



(a) Diffusion maps coordinates for a height profile of two harmonically related sinusoidals ($3\lambda_1 = 4\lambda_2$). As is seen in the figure, the simple periodic height curve is easily parameterized by two circles. The color indicates the speed along the manifold, from blue (slow) to yellow (fast).



(b) Diffusion maps coordinates for a height profile of three nonharmonic sinusoidals($3\lambda_1 = 4\lambda_2 = \pi\lambda_3$). The curves no longer overlap, and the space covered by the dynamics is larger. The multiple crossings of curves with different directions indicate that the dynamics might not be two-dimensional anymore.

Figure 8.1.

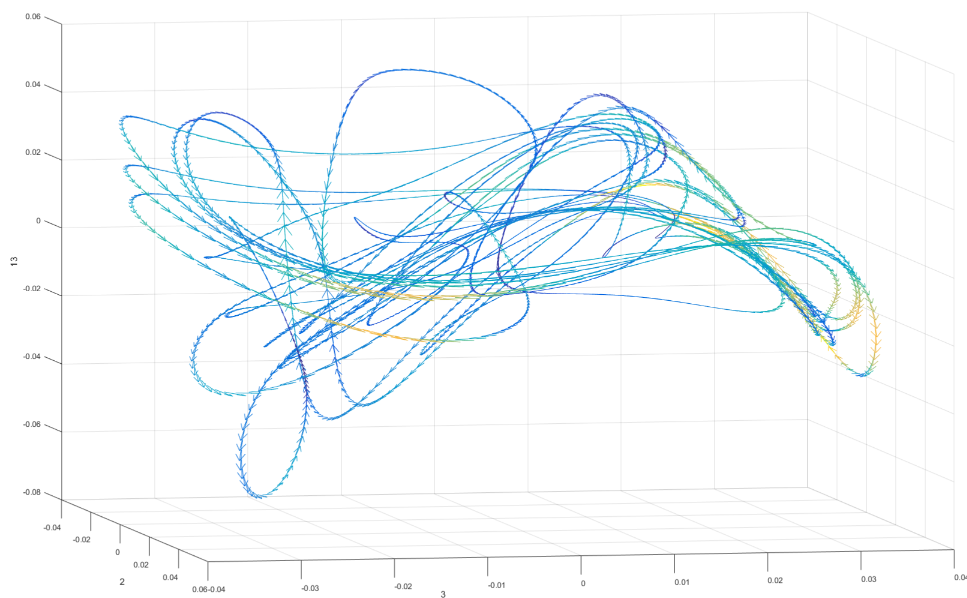


Figure 8.2.: The same data as in figure 8.1b, but with the 13th eigenvector added to the z-axis. Although it is hard to see in the two-dimensional image, the curves no longer overlap; the curves of the dynamic are disentangled. As the height parameter is three-dimensional, it would be expected that three dimensions can describe the manifold.

Cities	Distance (km)	PID cost/m	MPC cost/m	Height gain/loss (m)
Berlin to Hanover	284	0.85862	0.84902	+1365/-1347
Bolzano to Innsbruck	120	1.4062	0.88573	+2741/-2437
Bolzano to Salzburg	295	1.1547	0.86064	+4709/-4549
Dresden to Berlin	190	0.85634	0.8441	+781/-863
Innsbruck to Zuerich	81	1.1084	0.99943	+6201/-6370
Konstanz to Innsbruck	235	1.1455	0.97851	+5250/-5089
Konstanz to Salzburg	364	0.94662	0.84886	+3447/-3431
Konstanz to Stuttgart	171	0.98086	0.84570	+2068/-2234
Konstanz to Zuerich	68	0.8956	0.85265	+602/-607
Munich to Berlin	574	0.96322	0.84062	+5940/-6420
Munich to Bolzano	273	0.69066	0.84531	+3863/-4115
Munich to Salzburg	142	0.89217	0.8422	+1189/-1274
Munich to Stuttgart	231	0.91683	0.83792	+1907/-2174

Table 8.1.: Samples of the results comparing the MPC controller to a PID controller, using a cost balancing energy usage and keeping the velocity. Worth noting is that the improvement for the MPC controller is higher for roads that could be expected to be more mountainous. For example, the controllers perform almost equally well between Berlin and Hanover, whereas the comparable distance between Bolzano and Salzburg gives the MPC controller much better performance. All results can be seen in table B.1 of Appendix B.

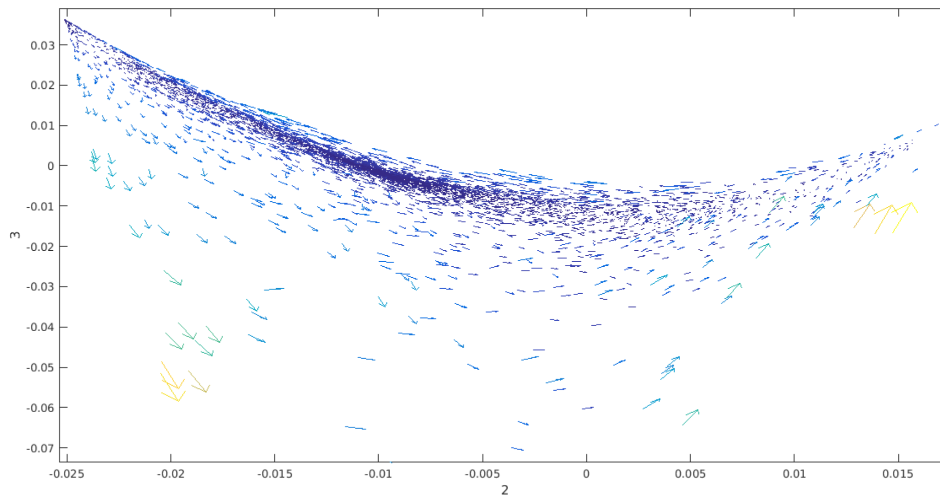


Figure 8.3.: The parameterized diffusion maps manifold for a combination of 4 different simulations using different real height data. Here, the periodic structure is not so clear anymore. One can see a concentration of datapoints in some areas of the manifold, even though most in the high-density areas have been filtered away. However, there is a clear pattern of circling around an attractor around the darker blue (slower) areas of higher concentration.

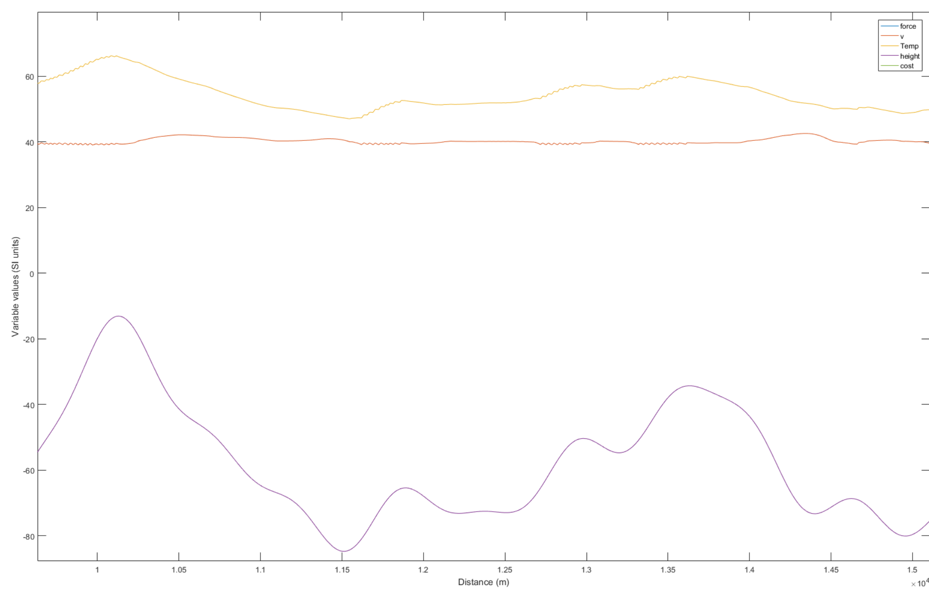
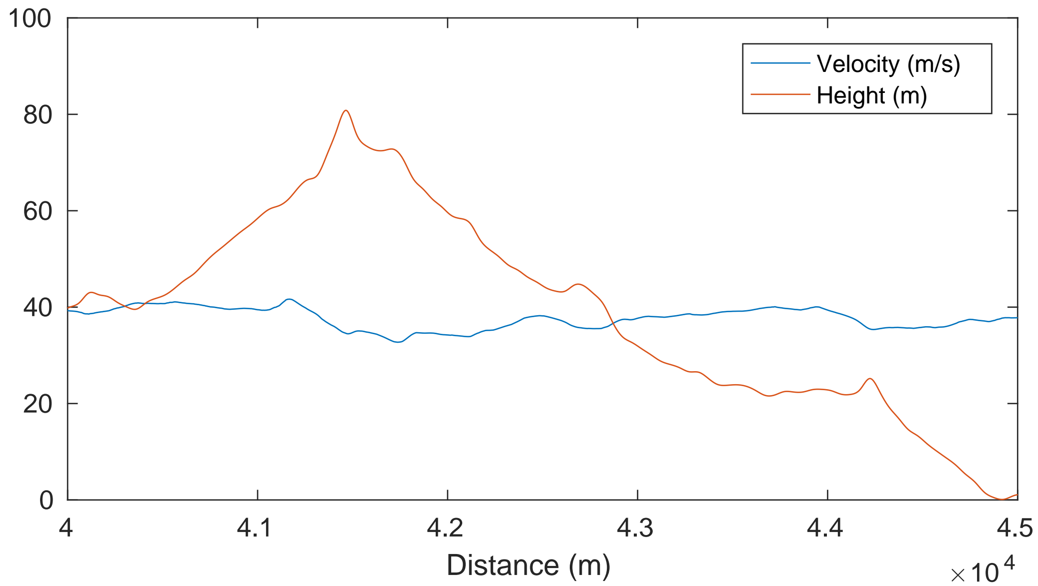
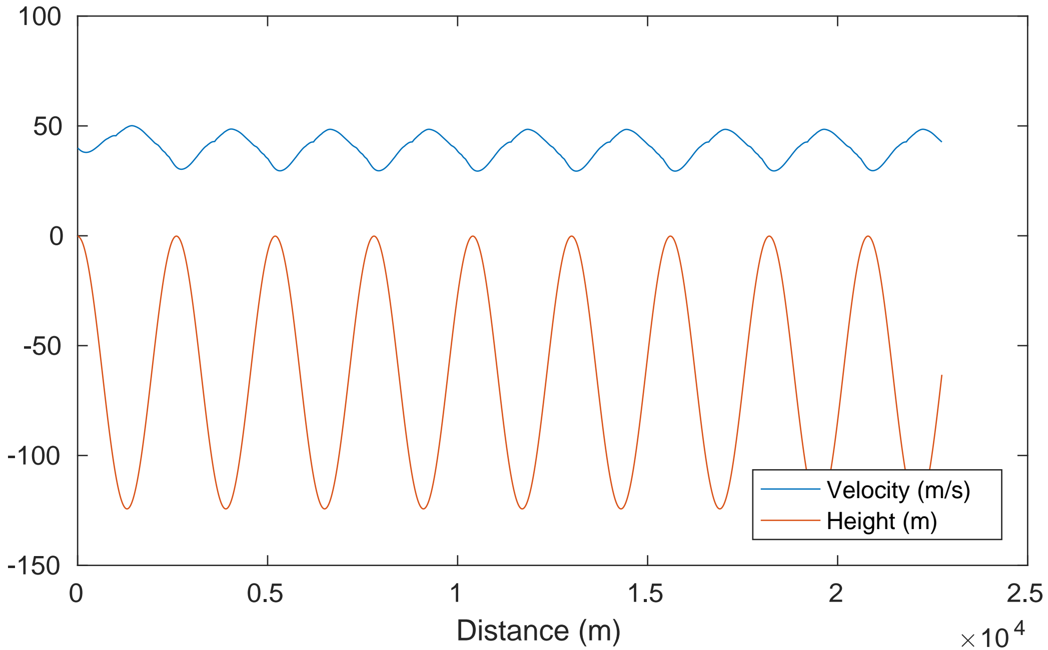


Figure 8.4.: Simulation using a diffusion maps controller as outlined in this thesis. The diffusion maps coordinates of the first, second, and eighth eigendirections are interpolated from the state variables of velocity, slope 250m ahead, and slope 650m ahead. One can see behaviours similar to those of the MPC control, letting velocity increase when going downhill.



(a) Simulation using a diffusion maps controller as outlined in this thesis. The diffusion maps coordinates of the first and second eigendirections are interpolated from the state variables of motor thrust, slope 295m ahead, and slope 500m ahead. Again, one can see the energy–saving behaviours of MPC being approximated. The controller was built from data generated between Konstanz and Zürich, and used on the road between Konstanz and Innsbruck. This shows that the controller generalizes knowledge trained on other data. Map data ©2017 GeoBasis-DE/BKG (©2009), Google.



(b) The same controller on a different height curve, this time an artificial sinusoidal curve with amplitude 65m and wavelength 2600m. The controller fails to control this adequately, and performs almost twice as bad as the PID controller (table 8.2). To improve robustness, one could therefore consider combining the diffusion maps controller with a PID.

Figure 8.5.

Road	Training data	PID cost/m	MPC cost/m	DM cost/m
Konstanz to Zuerich	Konstanz to Innsbruck	0.89563	0.85266	0.82671
Konstanz to Zuerich	Bolzano to Innsbruck	0.89563	0.85266	1.18159
$65\cos\frac{2\pi x}{2600}$	Konstanz to Innsbruck	1.0047	0.9853	2.3191

Table 8.2.: Some results for using the diffusion maps-based controller. As can be seen, the results are highly sensitive to the training data. Some types of roads, which the controller has not seen before, may result in degraded performance, even simple cases of sinusoids.

8.3. Outlook on further work

There are several improvements to be done on the different interpolations used, especially to able to interpolate using more variables. One improvement would be implementing efficient datastructures for nearest-neighbour search, which could much help local higher-dimensional interpolating algorithms. Examples of these include *ball-trees* [61] and *kD-trees* [62], which both work on dividing the space hierarchically to eliminate the most and biggest branches of the trees, and therefore as many candidates as possible, with the fewest comparisons.

Another improvement would be using better smoothing interpolation or fitting. As there are numbers of outliers in the data which cannot easily be filtered away automatically, it would be beneficial to have interpolation methods that are robust to this type of noise, and still be able to operate in more than two dimensions.

To be able to include more data, and obtain more robust control patterns, other eigenvalue solvers could be used, that are more applicable for larger-scale problems. The basic MATLAB implementation is restricted to single-core usage. Parallelizing over several machines would also give the benefit of providing more memory for a bigger kernel matrix.

8.4. Conclusion

In this thesis, a novel method of generating approximations of *Model Predictive Control* (MPC) using diffusion maps with closed observables was proposed and implemented. For this, an MPC framework was developed using the `python` module `pyomo`. Another framework was developed to implement the diffusion maps method and convert the approximations into a controller. Finally, the controllers from both frameworks were compared and validated for a test scenario of a car on a hilly road. Comparisons were made both against each other and against a simpler PID-controller.

The MPC framework was shown to perform better than a simple PID-controller in all cases, as seen in table B.1. The results for the proposed method are also promising, with similar-looking optimizing behaviours as those for MPC seen in the diffusion maps controller, for example in figure 8.5a. In one instance, the overall results were even better than MPC, seen in table 8.2. However, the controller is highly sensitive to which training data is used, and what features are selected for interpolating the control. In order to produce a robust controller, more development is needed, and other methods, such as using variable bandwidth kernels in diffusion maps, and refining the MATLAB interpolation algorithms,

are needed.

Nonetheless, the method already has several benefits compared with similar existing methods. In contrast with explicit MPC [12], the method works well for nonlinear cases, with many variables. Compared with using neural networks to approximate MPC [4, 5], the method is much more interpretable, visualizable, and the sampling error is more controllable. If there are unsampled regions, one can look at the diffusion maps embedding generated, see where the "holes" are, and try to generate more data there.

Towards the overall goal of generating usable optimal control from engineering models, this thesis therefore provides a first step and proof-of-concept. Diffusion maps with closed observables is a promising way to generate approximations of optimal control from MPC, however, more work needs to be done before the method is mature and robust.

Appendix

A. Parameter Values

This chapter contains the parameter values used in the algorithms in chapter 5.

Parameter name	Symbol	Value
Gravitational acceleration	g	9.82ms^{-2}
Vehicle mass	m_{car}	1000kg
Air drag coefficient	c_{air}	0.5kgm^{-1}
Rolling resistance coefficient	c_{roll}	0.005
Velocity limits	$\{v_{\text{low}}, v_{\text{high}}\}$	$\{25, 50\}\text{ms}^{-1}$
Motor power limits	$\{F_{\text{motor, low}}, F_{\text{motor, high}}\}$	$\{0, 2946\}$N
Braking limits	$\{F_{\text{brake, low}}, F_{\text{brake, high}}\}$	$\{-2946, 0\}$N
Horizon length	$x_{\text{end}} - x_0$	5000m
Discretization steps	-	506
Iteration step length	ΔX_{p}	500m
Energy cost coefficient	w_{motor}	0.001
Velocity cost coefficient	w_{vel}	0.001
Acceleration cost coefficient	w_{acc}	0.001
Thrust smoothness coefficient	w_{force}	0.1

Table A.1.: The parameters for the MPC simulations.

B. Model Predictive Control Results

This chapter contains results for the simulations using the Model Predictive Control framework in chapter 5.

Table B.1.: Results for the MPC simulations

Cities	Distance (km)	PID cost/m	MPC cost/m	Altitude gained/lost (m)
Berlin to Hamburg	287	0.85689	0.84740	+1072/-1103
Berlin to Hanover	284	0.85862	0.84902	+1365/-1347
Berlin to Salzburg	719	1.0004	0.85521	+8533/-8139
Berlin to Zuerich	820	1.072	0.85590	+11981/-11609
Bolzano to Innsbruck	120	1.4062	0.88573	+2741/-2437
Bolzano to Salzburg	295	1.1547	0.86064	+4709/-4549
Cologne to Berlin	566	0.93617	0.84832	+4933/-4946
Cologne to Erfurt	359	1.1556	0.85432	+6921/-6775
Cologne to Hamburg	426	0.9319	0.8473	+3798/-3844
Cologne to Hanover	288	0.92428	0.84836	+2755/-2750
Cologne to Zuerich	553	1.0449	0.85734	+8008/-7651
Dresden to Berlin	190	0.85634	0.8441	+781/-863
Dresden to Hamburg	492	0.87915	0.84625	+2887/-2999
Dresden to Hanover	362	0.87148	0.84668	+2079/-2143
Dresden to Innsbruck	611	0.9986	0.85695	+7714/-7255
Dresden to Salzburg	597	0.99966	0.85472	+8018/-7704
Dresden to Zuerich	700	1.0749	0.85525	+11677/-11386
Erfurt to Berlin	298	0.87570	0.84485	+1891/-2048
Erfurt to Dresden	214	0.90511	0.84702	+2312/-2387
Erfurt to Hamburg	386	0.88926	0.84373	+2872/-3061
Erfurt to Hanover	244	0.87977	0.8429	+1950/-2090
Erfurt to Innsbruck	567	1.0308	0.85688	+7144/-6762
Erfurt to Salzburg	550	1.0175	0.85432	+7435/-7198
Erfurt to Zuerich	534	0.14262	0.86082	+10814/-10599
Frankfurt to Cologne	188	0.89446	0.84567	+1904/-1953
Frankfurt to Dresden	458	0.93124	0.84954	+6069/-6052
Frankfurt to Erfurt	259	0.91582	0.85171	+3350/-3255
Frankfurt to Hamburg	491	0.91896	0.84675	+5547/-5641
Frankfurt to Hanover	349	0.91428	0.84693	+4481/-4526
Frankfurt to Salzburg	531	0.99244	0.85478	+6308/-5978
Freiburg to Berlin	789	0.9680	0.84649	+8278/-8514
Freiburg to Cologne	426	0.91022	0.8433	+3184/-3409
Freiburg to Dresden	669	0.9887	0.84733	+8100/-8255

Table B.1.: Results for the MPC simulations

Cities	Distance (km)	PID cost/m	MPC cost/m	Altitude gained/lost (m)
Freiburg to Erfurt	515	0.90607	0.84687	+4545/-4623
Freiburg to Frankfurt	267	0.84769	0.84214	+784/-957
Freiburg to Hamburg	744	0.92870	0.84531	+7228/-7495
Freiburg to Hanover	604	0.91192	0.84492	+5774/-5993
Freiburg to Konstanz	121	0.9170	0.85943	+1564/-1419
Freiburg to Salzburg	564	0.96020	0.85126	+5775/-5617
Freiburg to Stuttgart	197	0.93165	0.85182	+1548/-1573
Hamburg to Hanover	158	0.86131	0.85144	+685/-636
Hamburg to Zuerich	835	1.0593	0.85671	+14587/-14179
Hanover to Salzburg	761	1.007	0.85481	+11389/-11015
Hanover to Zuerich	700	1.0452	0.85573	+12516/-12165
Innsbruck to Salzburg	184	0.86896	0.84045	+1078/-1223
Innsbruck to Zuerich	81	1.1084	0.99943	+6201/-6370
Konstanz to Berlin	781	1.0357	0.84578	+10910/-11288
Konstanz to Dresden	660	1.0407	0.84591	+10349/-10647
Konstanz to Erfurt	496	0.96204	0.84862	+9540/-9760
Konstanz to Frankfurt	356	0.9315	0.84043	+3383/-3697
Konstanz to Hamburg	801	0.11981	0.84528	+12733/-13139
Konstanz to Hanover	661	1.0227	0.84489	+11542/-11900
Konstanz to Innsbruck	235	1.1455	0.97851	+5250/-5089
Konstanz to Salzburg	364	0.94662	0.84886	+3447/-3431
Konstanz to Stuttgart	171	0.98086	0.84570	+2068/-2234
Konstanz to Zuerich	68	0.8956	0.85265	+602/-607
Munich to Berlin	574	0.96322	0.84062	+5940/-6420
Munich to Bolzano	273	0.69066	0.84531	+3863/-4115
Munich to Cologne	565	1.0408	0.84122	+7523/-7988
Munich to Dresden	452	0.96450	0.83975	+5510/-5909
Munich to Erfurt	406	0.96089	0.84088	+4888/-5209
Munich to Frankfurt	388	0.96737	0.83860	+4122/-4538
Munich to Freiburg	419	0.9282	0.84312	+3582/-3826
Munich to Hamburg	777	0.94733	0.8427	+7364/-7873
Munich to Hanover	619	0.99654	0.84232	+9070/-9528
Munich to Innsbruck	161	0.88102	0.85210	+814/-753
Munich to Salzburg	142	0.89217	0.8422	+1189/-1274
Munich to Stuttgart	231	0.91683	0.83792	+1907/-2174
Stuttgart to Berlin	620	0.98879	0.84628	+7218/-7430
Stuttgart to Cologne	362	0.9456	0.84320	+3611/-3806
Stuttgart to Dresden	498	0.99943	0.84685	+6982/-7114
Stuttgart to Erfurt	333	0.91472	0.85242	+6501/-6554
Stuttgart to Frankfurt	200	0.85308	0.84083	+1050/-1199
Stuttgart to Hamburg	638	0.9831	0.84537	+9631/-9874
Stuttgart to Hanover	500	0.99228	0.84499	+8418/-8609

Table B.1.: Results for the MPC simulations

Cities	Distance (km)	PID cost/m	MPC cost/m	Altitude gained/lost (m)
Stuttgart to Salzburg	375	0.95238	0.85321	+3882/-3702
Stuttgart to Zuerich	214	1.04	0.85802	+3023/-2864

C. Source Code

This chapter contains source code used to produce the results of this thesis. As even this appendix was too small to contain all the code, most of it can instead be found at `https://github.com/EWannerberg/AutomaticHeuristicGeneration` (as of 14 July 2017), for convenient usage with the `git` utility. The code here is split up into four main parts: the MPC part,the core diffusion maps part, the closed observables part and the validation part.

C.1. Model Predictive Control

The Model Predictive Control code is split into different classes as described in chapter 5, which roughly correspond to a class each. The main algorithm is run from the `SimulationManager` class in `run.py`.

Listing C.1: `run.py`

```
import model_util as mu
import solver_util as su
import output
import debug


__author__ = 'Erik Wannerberg'


class SimulationManager:
    """
    Class to control the simulation.
    :type _model_obj: model_util.BaseModel
    :type _solver_obj: solver_util.SolverUtil
    :type _output_list: list[output.OutputInterface]
    """
    def __init__(self, model_obj, solver_obj, output_list=list()):
        assert isinstance(model_obj, mu.BaseModel)

        self._model_obj = model_obj  # type: model_util.BaseModel

        assert isinstance(solver_obj, su.SolverUtil)
        self._solver_obj = solver_obj

        if not isinstance(output_list, list):
            assert isinstance(output_list, output.OutputInterface)
            self._output_list = [output_list]
        else:
            assert all(isinstance(o, output.OutputInterface) for o in output_list)
            self._output_list = output_list

        self._error_printer = None

        self.shift_length = 1

```

```
36      def setup(self):
37          """
38          Call all the necessary functions in the right order on the participating objects.
39          :return:
40          :rtype:
41          """
42          import math
43          self._model_obj._create_params()
44          self._model_obj._create_variables()
45          self._model_obj._create_expressions()
46          self._model_obj._create_diffeq_constraints()
47          self._model_obj._create_integrals()
48          self._model_obj._discretize()
49          self._model_obj._create_objectives()
50          self._model_obj._presolve()
51
52          self._solver_obj._set_solver(solver_type=self._model_obj._parameters["solver"],
53
                                        ↪  solver_executable=self._model_obj._parameters["solver_executable"])
54          self.shift_length = math.floor(self._model_obj.parameters["delta_x"]
55                                         / self._model_obj.parameters["episode_length"]
56                                         * self._model_obj.parameters["episode_pts"])
57
58      def set_error_printer(self, error_printer):
59          """
60          Set an outputter to print when something goes wrong
61          :param error_printer: Outputter at error
62          :type error_printer: output.Outputinterface
63          """
64          assert isinstance(error_printer, output.OutputInterface)
65
66          self._error_printer = error_printer
67
68      def run(self):
69          """
70          Run the main simulation loop until completion.
71          """
72          status = self._solver_obj._solve()
73          if not status:
74              if self._error_printer is not None:
75                  self._error_printer.print_output()
76              debug.CheckInstanceFeasibility(self._model_obj.model, 0.001)
77
78          for output_obj in self._output_list:
79              if isinstance(output_obj, output.SavingOutputUtil):
80                  output_obj.set_index_var(self._model_obj.model.x_local)
81
82                  output_obj.add_saved_vars(self._model_obj.model.v,
83                                            self._model_obj.model.braking_force,
84                                            self._model_obj.model.slope,
85                                            self._model_obj.model.height,
86                                            self._model_obj.model.inst_cost,
87                                            self._model_obj.model.motor_force,)
88
89              output_obj.update()
90
91          while ((self._model_obj.model.global_x_0.value
92                  + self._model_obj.parameters["episode_length"]
93                  + self._model_obj.parameters["delta_x"])
94                 < self._model_obj.parameters["end_dist"]):
95
96              self._model_obj.shift_all(test_model.model.x_local, self.shift_length)
97              self._model_obj.model.global_x_0 +=
                 ↪  sorted(self._model_obj.model.x_local)[self.shift_length]
98              self._model_obj._presolve()
```

```
99
100             status = self._solver_obj._solve()
101
102             if not status:
103                 if self._error_printer is not None:
104                     self._error_printer.print_output()
105                 debug.CheckInstanceFeasibility(self._model_obj.model, 0.001)
106
107             for output_obj in self._output_list:
108                 output_obj.update()
109
110         for output_obj in self._output_list:
111             if isinstance(output_obj, output.SavingOutputUtil):
112                 output_obj.print_output(self._model_obj.parameters["output_file"])
113             else:
114                 output_obj.print_output()
115
116
117 if __name__ == "__main__":
118     import sys
119
120     filename = "example/test.json"
121     if len(sys.argv) > 1:
122         filename = sys.argv[1]
123
124     test_model = mu.VehicleModel(filename)
125
126     solver = su.SolverUtil(test_model)
127     outputter = output.SavingOutputUtil(test_model)
128     sout = output.SimpleOutputUtil(test_model)
129
130     runner = SimulationManager(model_obj=test_model,
131                                solver_obj=solver,
132                                output_list=[outputter, sout])
133     runner.setup()
134     runner.set_error_printer(sout)
135     solver._solver.options["max_iter"] = 5000
136     solver._solver.options["bound_relax_factor"] = 0.0001  # is actual constraint
        ↪ satisfaction limit
137     solver._solver.options["dual_inf_tol"] = 0.1
138     solver._solver.options["constr_viol_tol"] = 0.0001
139     runner.run()
140     solver.print_error_summary()
141
142     if test_model.parameters["plot_after_finish"]:
143         outputter.plot_result_graph()
144         sout.plot_result_graph()
```

C.2. Diffusion maps

This section contains the code from chapter 6.
The main script comes in two versions, in `dmapsTesting.m` and `dmapsTesting_timlag.m`, where the compute diffusion maps modes for single-timeseries runs and multiple-timeseries runs respectively. The input files for `dmapsTesting.m` are the ones given as output from the MPC part, whereas the ones for `dmapsTesting_timlag.m` are the ones saved after preprocessing by the functions in `prune_output_matrices.py`.

Listing C.2: `dmapsTesting.m`

```
1 %% Define parameters
```

```
2   dataFile = '../Validation-MPC/heightmaps/Bolzano_to_Innsbruck_res502017
        -06-24-05.09.52.out';
3
4   % dataFile = 'C:/Users/Public/shareVM/cos_5rand.out';
5
6   num_tl_pts = 130;
7   num_eigs = 30;
8   ignore_first = 250;
9   max_allowed_slope = 0.2;
10  takeEvery = 1;
11  tlag_end_attenuation = 0.2;
12  kernel_width_minmedians = 6;
13  % kernel_width_maxmins = 1.5;
14  kernel_width_medians = 2;
15
16  %% Import data
17  dataStruct = importdata(dataFile);
18  data = dataStruct.data(ignore_first:end, :);
19
20  force_col = find(strcmp(dataStruct.colheaders, 'motor_force'));
21  dist_col = find(strcmp(dataStruct.colheaders, 'x_local'));
22  slope_col = find(strcmp(dataStruct.colheaders, 'slope'));
23
24  num_rows = max(size(data));
25
26  %% create filter for slope
27  highslope_cols = abs(data(:,slope_col)) > max_allowed_slope;
28  ignore_rows = zeros(size(highslope_cols), 'logical');
29
30  ignore_width = 40;
31  for i=1:ignore_width
32      ignore_rows(1:(num_rows-i + 1)) = ignore_rows(1:(num_rows-i + 1)) |
            highslope_cols (i:num_rows);
33      ignore_rows(i:num_rows) = ignore_rows(i:num_rows) | highslope_cols (1:(num_rows-
            i + 1));
34  end
35
36  %% construct time-lagged manifold
37  tlag_exponential_factor = -log(tlag_end_attenuation)/(num_tl_pts-1);
38  timeLaggedData = constructTimeLagged(zscore(data(:,force_col)), num_tl_pts,
        tlag_exponential_factor);
39
40  indices = 1:size(timeLaggedData,1);
41  % indices = indices(~ignore_rows(floor(ignore_width/2):(num_rows+floor(ignore_width
        /2)-num_tl_pts)));
42  indices = indices(~ignore_rows(1:(num_rows+1-num_tl_pts)));
43  reducedIndices = indices(1:takeEvery:max(size(indices)));
44
45  %% Execute Diffusion Maps algorithm
46  dist_mat = makeDistMatrix(timeLaggedData(reducedIndices,:));
47  kernel_width = median(min(dist_mat + eye(size(dist_mat))*max(max(dist_mat))))*
        kernel_width_minmedians;
48  % kernel_width = max(min(dist_mat + eye(size(dist_mat))*max(max(dist_mat))))*
        kernel_width_maxmins;
49  % kernel_width = median(dist_mat(:)) * kernel_width_medians;
50  kernel_mat = constructKernelMat(dist_mat, kernel_width);
51  kernel_mat = normalizeKernel(kernel_mat);
52  [ eigvects, eigvals ] = constructDMaps( kernel_mat, num_eigs );
53
54  %% Postprocess and plot
55  figure; plot(dataStruct.data(reducedIndices,dist_col),eigvects(:,[2:10]))
56  grads = create_gradients(eigvects(:,2:30),dataStruct.data(reducedIndices,dist_col));
57
58  if ~(exist('k', 'var') && exist('l', 'var') && exist('m', 'var'))
59      k = 2; l = 3; m = 4;
```

```
60 end
61
62 figure; q = quiver(eigvects(:,k),eigvects(:,l), grads(:,k-1), grads(:,l-1), 3);
63 mags = sqrt(sum(grads(:,[k,l]).^2,2));
64 color_arrows(q, mags)
65 xlabel(['{\bf\psi}_', num2str(k)])
66 ylabel(['{\bf\psi}_', num2str(l)])
67
68 figure; q = quiver3(eigvects(:,k),eigvects(:,l), eigvects(:,m), grads(:,k-1), grads
       (:,l-1), grads(:,m-1), 3);
69 mags = sqrt(sum(grads(:,[k,l,m]).^2,2));
70 color_arrows(q, mags)
71 xlabel(['{\bf\psi}_', num2str(k)])
72 ylabel(['{\bf\psi}_', num2str(l)])
73 zlabel(['{\bf\psi}_', num2str(m)])
```

Listing C.3: dmapsTesting_timlag.m

```
1 %% Define parameters
2 dataFile = 'removed_std_force.csv';
3
4 num_eigs = 30;
5 % takeEvery = 1;
6 reducedIndices = 1:4000;
7 timeLagAttenuation = 0.01;
8
9 %% Import data
10 dataStruct = importdata(dataFile);
11 dataStruct.textdata = char(dataStruct.textdata);
12 dataStruct.colheaders = split(dataStruct.textdata(3:end),','); % starts with '# '
       due to numpy
13
14 dist_col = find(strcmp(dataStruct.colheaders, 'x_local'));
15 force_col = find(strcmp(dataStruct.colheaders, 'motor_force'));
16 id_col = find(strcmp(dataStruct.colheaders, 'dataf_id'));
17 slope_col = find(strcmp(dataStruct.colheaders, 'slope'));
18
19 % ignore_rows = createSlopeFilter(dataStruct.data, slope_col, dist_col, 40, 0.2,
       id_col);
20 data = dataStruct.data(:,:);
21 timeLaggedData = data(:,(max(size(dataStruct.colheaders))+1):end);
22 num_tl_pts = size(timeLaggedData,2);
23 timeLagDecays = exp(log(timeLagAttenuation)/num_tl_pts * (0:(num_tl_pts-1)) );
24 %extract_starts
25 timeLaggedData = timeLaggedData .* (ones(size(timeLaggedData,1),1) * timeLagDecays);
26
27 %% Execute Diffusion Maps algorithm
28 dist_mat = makeDistMatrix(timeLaggedData(reducedIndices,:));
29 kernel_width = 600; %median(min(dist_mat + eye(size(dist_mat))*max(max(dist_mat))))
       *4;
30 kernel_mat = constructKernelMat(dist_mat, kernel_width);
31 kernel_mat = normalizeKernel(kernel_mat);
32 [ eigvects, eigvals ] = constructDMaps( kernel_mat, num_eigs );
33
34
35 %% Postprocess and plot
36 [~,inds] = sort(data(reducedIndices,dist_col));
37 figure; plot(data(reducedIndices(inds),dist_col),eigvects(inds, [2:10]))
38
39 %compute gradients once for each file used
40 % gradients = zeros(size(eigvects(:,2:end)));
41 % for id = reshape(unique(data(:,id_col)),1,[])
42 %     this_id = data(reducedIndices,id_col) == id;
43 %     gradients(this_id,:) = create_gradients(eigvects(this_id,2:end),data(
       reducedIndices(this_id),dist_col));
```

```
44  % end
45
46  if ~(exist('k', 'var') && exist('l', 'var') && exist('m', 'var'))
47      k = 2; l = 3; m = 4;
48  end
49
50  figure; q = scatter3(data(reducedIndices,6), data(reducedIndices, 5), data(
        reducedIndices,4), 5 , eigvects(:,k));
51  % figure; q = quiver(eigvects(:,k),eigvects(:,l), gradients(:,k-1), gradients(:,l-1)
        , 3);
52  % mags = sqrt(sum(gradients(:,[k,l]).^2,2));
53  % color_arrows(q, mags)
54  % xlabel(k)
55  % ylabel(l)
56  %
57  % figure; q = quiver3(eigvects(:,k),eigvects(:,l), eigvects(:,m), gradients(:,k-1),
        gradients(:,l-1), gradients(:,m-1), 3);
58  % mags = sqrt(sum(gradients(:,[k,l,m]).^2,2));
59  % color_arrows(q, mags)
60  % xlabel(k)
61  % ylabel(l)
62  % zlabel(m)
```

Below follow the functions used in the two scripts, in order of usage.

Listing C.4: `constructTimeLagged.m`

```
1   function [ time_lagged_matrix ] = constructTimeLagged( datapoints, num_pts,
        weight_kappa )
2   %CONSTRUCTTIMELAGGED Construct time-lagged variables
3   %   Construct a matrix of time-lagged variables by appending each data
4   %   point by the subsequent points in the first dimension, weighted by
5   %   exp(-weight_kappa * i), where i is the difference in index between the
6   %   point and the subsequent added one
7
8   orig_size = size(datapoints);
9   new_num_rows = orig_size(1)-num_pts+1;
10
11
12
13  time_lagged_matrix = zeros(new_num_rows, orig_size(2) * num_pts);
14
15  for i = 1:num_pts
16      time_lagged_matrix(:,((i-1)*orig_size(2)+1):(i*orig_size(2))) = ...
17          datapoints(i:(new_num_rows + i - 1),:)*exp(-weight_kappa*(i-1));
18  end
19
20  end
```

Listing C.5: `makeDistMatrix.m`

```
1   function [ output_mat ] = makeDistMatrix( input_mat )
2   %MAKEDISTMATRIX Basically squareform(pdist(input))
3
4   output_mat = squareform(pdist(input_mat));
5
6   end
```

Listing C.6: `constructKernelMat.m`

```
1   function [ kernel_matrix ] = constructKernelMat(dist_matrix, kernel_width)
2   %CONSTRUCTKERNELMAT Build the kernel matrix using the gaussian kernel.
3
4   kernel_matrix = exp(-(dist_matrix.^2)./(kernel_width.^2));
```

```
5
6  end
```

Listing C.7: normalizeKernel.m

```
1  function [ normalized_kernel_mat ] = normalizeKernel( kernel_matrix, alpha_exp )
2  %NORMALIZEKERNEL perform normalization
3  % This makes the diffusion maps eigenvectors equivalent to the heat kernels
4  % on the manifold.
5
6  if nargin < 2
7      alpha_exp = 1;
8  end
9
10 density_estimate = sum(kernel_matrix,2);
11
12 multvector = diag(sparse(density_estimate.^(-alpha_exp)));
13
14 normalized_kernel_mat = multvector * kernel_matrix * multvector;
15
16
17 end
```

Listing C.8: constructDMaps.m

```
1  function [ eigvects, eigvals ] = constructDMaps( kernel_matrix, num_components )
2  %CONSTRUCTDMAPS
3  %
4
5  % normalization factor
6  invroot_rowsum = 1./sparse(sqrt(sum(kernel_matrix,2)));
7
8  % symmetrize for computationally more efficient solution
9  symmetric_kernel = diag(invroot_rowsum) * kernel_matrix * diag(invroot_rowsum);
10
11 % solve eigenproblem
12 [vects, vals] = computeEigs(symmetric_kernel, num_components);
13 eigvals = diag(vals);
14
15 % undo symmetric tranformation to get real eigvects
16 vects = diag(invroot_rowsum) * vects;
17
18 % normalize
19 eigvects = vects*diag(sparse(1./sqrt(sum(vects.^2,1))));
20
21 % sort into descending order
22 [eigvals, inds] = sort(eigvals, 'descend');
23 eigvects = eigvects(:, inds);
24
25 end
```

The last file, constructDMaps.m, also uses computeEigs.m for computing the eigenpairs.

Listing C.9: computeEigs.m

```
1  function [ eigvecs, eigvals ] = computeEigs( symm_matrix, num_components )
2  %COMPUTEEIGS Compute eigenvectors and eigenvalues
3  %   current implementation using matlab eigs with optimizations for
4  %   symmetric matrix
5
6  optis.issymm = true;
7  optis.disp = 2;
```

```
8   optis.tol = 1e-8;
9   optis.maxit = 1500;
10  optis.p = min(200, size(symm_matrix,1));
11
12  disp('Starting eigenvalue solving')
13  [eigvecs, eigvals, flag] = eigs(symm_matrix, num_components, 'LM', optis);
14  disp('Eigenvalue solving finished, complete convergence: ' + string(not(boolean(flag
        ))))
15
16  end
```

For processing many datapoints, the data may be pruned using the utility functions of `prune_output_matrices.py`, for example `prune_output_matrices`.

Listing C.10: `prune_output_matrices.py`

```
95  def prune_closer_points(input_data, column_dist_subset=slice(None),
96                          keep_max_number=None, ignore_distances_under=None):
97
98      import scipy.spatial.distance as sd
99      import numpy as np
100     import heapq
101
102     num_inputs = np.shape(input_data)[0]
103     dist_mat = sd.squareform(sd.pdist(input_data[:, column_dist_subset],
104                                       metric='sqeuclidean'))
105
106     # find the closest neighbours to all points
107
108     # add something big to diagonal to not see self as closest
109     biggest_value = np.amax(dist_mat)
110     dist_mat[np.arange(0, num_inputs), np.arange(0, num_inputs)] = biggest_value
111     minimum_dist_indices = np.argmin(dist_mat, axis=1)
112     minimum_dist_values = dist_mat[np.arange(0, num_inputs), minimum_dist_indices]
113
114     if keep_max_number is None or keep_max_number > num_inputs:
115         keep_max_number = 0
116
117         if ignore_distances_under is None:
118             ignore_distances_under = np.percentile(minimum_dist_values, 25,
119                                                    interpolation='nearest')
120     elif ignore_distances_under is None:
121         ignore_distances_under = np.inf
122
123     ignore_distances_under **= 2
124     prune = np.zeros_like(minimum_dist_indices, dtype=bool)
125
126     pqueue = [(minimum_dist_values[i], i, minimum_dist_indices[i])
127               for i in range(num_inputs)]
128
129     # turn into priority queue
130     heapq.heapify(pqueue)
131
132     # remove indices until closest two neighbours are far enough apart
133     while pqueue[0][0] <= ignore_distances_under and len(pqueue) > keep_max_number:
134         current_candidate = pqueue[0]
135         current_index = current_candidate[1]
136         # check if removed already
137         if prune[current_candidate[2]]:
138             # add something big where pruned already to ignore
139             min_neighbour = np.argmin(dist_mat[current_index, :] + prune * biggest_value)
140             new_candidate = (dist_mat[current_index, min_neighbour],
141                              current_index,
142                              min_neighbour)
143             minimum_dist_values[current_index] = new_candidate[0]
144             heapq.heapreplace(pqueue, new_candidate)
145         else:
146             prune[current_index] = True
147             heapq.heappop(pqueue)
148
149     # reset diagonal to 0
150     dist_mat[np.arange(0, num_inputs), np.arange(0, num_inputs)] = 0
151
152     return input_data[np.logical_not(prune), :], dist_mat[np.ix_(np.logical_not(prune),
        ↪   np.logical_not(prune))]
153
```

```
154
```

C.3. Closed observables and interpolation

For generating the functions for interpolating the dynamics of the system, as described in section 3.3.5, some functions were used in addition to the core diffusion maps algorithm. Although it is executed as part of the diffusion maps preprocessing and listed above, `constructTimeLagged.m` (listing C.4) is technically part of this as well.

`create_closedobs_interps.m` creates the three interpolating functions.

Listing C.11: `create_closedobs_interps.m`

```
1  function [ in_fn , dynamic_fn , out_fn ] = create_closedobs_interps( input_mat ,
       diffmap_coord_mat , output_mat )
2  %CREATE_CLOSEDOBS_INTERPS Create the set of interpolating functions
3  %   creates interpolation functions input->dmaps->dmaps+1->output
4
5  in_fn = create_interp_fun(input_mat , diffmap_coord_mat);
6  dynamic_fn = ...
7      create_interp_fun(diffmap_coord_mat(1:end-1,:),...
8          diffmap_coord_mat(2:end,:));
9  out_fn = create_interp_fun(diffmap_coord_mat , output_mat);
10
11
12 end
```

`create_interp_fun.m` creates an interpolating function using the MATLAB class `scatteredInterpolant`.

Listing C.12: `create_interp_fun.m`

```
1  function [ interp_fun ] = create_interp_fun( datapt_mat , values_mat )
2  %CREATE_INTERP_FUN
3  %   create a function that
4  %   interpolates from NxD - matrix 'datapts' to NxV matrix 'values'
5  %   caution: current usage of scatteredInterpolant limits D to 3
6  %
7  % TODO: Provide interpolating function as a parameter for flexibility.
8
9  numpts = size(datapt_mat,1);
10 assert(numpts == size(values_mat,1), 'Not same amount of pts in data and values')
11
12 numdims = size(values_mat,2);
13
14 for i = 1:numdims
15     interpstruct.(char('interp' + string(i))) = ...
16         scatteredInterpolant(datapt_mat, values_mat(:,i), 'natural');
17 end
18
19     function output_vals = interpolation_function(points)
20         output_vals = zeros(size(points,1),numdims);
21         for j = 1:numdims
22             output_vals(:,j) = interpstruct.(char('interp' + string(j)))(points);
23         end
24     end
25
26     interp_fun = @interpolation_function;
27 end
```

C.4. Validation

This is the code of the validation framework.
The main function is `runSingleScenario`, which runs a simulation for a controller and a height map provided.

Listing C.13: `runSingleScenario.m`

```
function [ historyarray ] = runSingleScenario( param_filename, controller,
    status_every )
%RUNSINGLESCENARIO run using controller
%   Detailed explanation goes here

% optional argument
if nargin < 3
    status_every = inf;
    if nargin < 2
        % if no controller specified, use a 50m-PID
        controller = @(state, model) deal(getNextControlPID50m(state, model), state.
            x + 50);
    end
end

filename = param_filename;

params = loadParamsFromFile(filename);
model = initialiseCarModel(params);

currentState = model.state;

next_output = status_every;
historyarray = [];

simulationIsFinished = false;
while ~simulationIsFinished

    [upcoming_control, nextCtrlDist] = controller(model.state, model);
    model = simulationDistanceStepForward(model, upcoming_control, nextCtrlDist);

    historyarray = [historyarray; stateToVec(model.state)];

    simulationIsFinished = model.state.x > model.heightdata.dist_list(end-50);
    simulationIsFinished = simulationIsFinished || model.state.v < 0.01;
    if model.state.x > next_output
        disp(['Validation currently at x = ', num2str(model.state.x)]);
        next_output = next_output + status_every;
    end
end

% utility for saving history
function state_vec = stateToVec(state)
        state_vec = [state.force state.t state.x state.v ...
            state.acc state.Temp state.height state.cost state.slope];
end

end
```

For comparison, PID-controllers were implemented, for example `getNextControlPID50m`.

Listing C.14: `getNextControlPID50m.m`

```
function [ control ] = getNextControlPID50m( currentState, model )
```

```
2  %GETNEXTCONTROLPID Summary of this function goes here
3  %   Detailed explanation goes here
4
5  persistent lastState;
6  persistent inte;
7
8  if isempty(lastState)
9      lastState = currentState;
10     inte = 0;
11 end
12
13
14 steadyStateForce = 950;
15
16 intHalfTime = 40;
17
18 % intWeight = 0.5^(1/intHalfTime);
19 intWeight = 1;
20
21 prop = model.params.v_init - currentState.v;
22 if lastState.t ~= currentState.t
23     diff = (currentState.v - lastState.v)/(currentState.t - lastState.t);
24 else
25     diff = 0;
26 end
27 inte = inte*intWeight + (model.params.v_init - currentState.v)*(currentState.t -
       lastState.t);
28
29 coef_P = 2;
30 coef_I = 0.6;
31 coef_D = 0.1;
32
33 scaling = 200;
34
35 control = steadyStateForce + (coef_P * prop + coef_I * inte + coef_D * diff) *
       scaling;
36
37 lastState = currentState;
38 end
```

In order to utilize the interpolation functions from Appendix C.3, one can turn the functions from listing C.11 into a controller using `getNextControlDM`.

Listing C.15: `getNextControlDM.m`

```
1  function [ upcoming_control, next_control_dist ] = getNextControlDM( current_state,
       model, DM_funs, step_len )
2  %UNTITLED Summary of this function goes here
3  %   Detailed explanation goes here
4
5  if nargin < 4
6      step_len = 50;
7  end
8
9  next_control_dist = current_state.x + step_len;
10
11 input_fun=DM_funs{1};
12 dyn_fun=DM_funs{2};
13 out_fun=DM_funs{3};
14
15 height_250m_pred = model.slopeEq(current_state.x + 250);
16 height_650m_pred = model.slopeEq(current_state.x + 650);
17
18 upcoming_control = ...
19     out_fun(...
```

```
20          dyn_fun(...
21              input_fun(...
22                  [current_state.v, height_250m_pred, height_650m_pred])));
23
24
25  end
```

Bibliography

[1] Felix Dietrich, Gerta Köster, and Hans-Joachim Bungartz. Numerical Model Construction with Closed Observables. *arXiv:1506.04793 [math-ph]*, June 2015.

[2] Eduardo F. Camacho and Carlos Bordons. *Model Predictive Control*. Springer Science & Business Media, January 2013.

[3] Bernhard Haas. Predictive control systems in heavy-duty commercial vehicles. In *IAV Control Systems Conference*, pages 257–264, Renningen, 2012. expert verlag.

[4] J. Gomez Ortega and E. F. Camacho. Mobile robot navigation in a partially structured static environment, using neural predictive control. *Control Engineering Practice*, 4(12): 1669–1679, 1996.

[5] Bernt M. Åkesson and Hannu T. Toivonen. A neural network model predictive controller. *Journal of Process Control*, 16(9):937–946, October 2006. doi: 10.1016/j.jprocont.2006.06.001.

[6] Ronald R. Coifman and Stéphane Lafon. Diffusion maps. *Applied and Computational Harmonic Analysis*, 21(1):5–30, July 2006. doi: 10.1016/j.acha.2006.04.006.

[7] Wilhelmus H. A. Schilders, Henk A. van der Vorst, Joost Rommes, Hans-Georg Bock, Frank de Hoog, Avner Friedman, Arvind Gupta, Helmut Neunzert, William R. Pulleyblank, Torgeir Rusten, Fadil Santosa, Anna-Karin Tornberg, Luis L. Bonilla, Robert Mattheij, and Otmar Scherzer, editors. *Model Order Reduction: Theory, Research Aspects and Applications*, volume 13 of *Mathematics in Industry*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. doi: 10.1007/978-3-540-78841-6.

[8] Janick Frasch. *Parallel Algorithms for Optimization of Dynamic Systems in Real-Time*. Arenberg Doctoral School, 2014.

[9] Lars Grüne and Jürgen Pannek. *Nonlinear Model Predictive Control*. Communications and Control Engineering. Springer International Publishing, Cham, 2017. doi: 10.1007/978-3-319-46024-6.

[10] A. E. Bryson. *Applied Optimal Control: Optimization, Estimation and Control*. CRC Press, January 1975.

[11] L. S. Pontryagin. *Mathematical Theory of Optimal Processes*. CRC Press, March 1987.

[12] Alberto Bemporad, Manfred Morari, Vivek Dua, and Efstratios N. Pistikopoulos. The explicit linear quadratic regulator for constrained systems. *Automatica*, 38(1):3–20, January 2002. doi: 10.1016/S0005-1098(01)00174-1.

[13] MATLAB® Model Predictive Control Toolbox™, September 2016. The MathWorks®, Inc.

[14] Simulink® - Simulation and Model-Based Design, August 2016. The MathWorks®, Inc., Natick, Massachusetts, United States.

[15] Andreas Wächter and Lorenz T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–57, March 2006. doi: 10.1007/s10107-004-0559-y.

[16] Andrew Makhorin and GNU Project. GLPK (GNU linear programming kit). 2008.

[17] Robert Fourer, David Gay, and Brian Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press, November 2002.

[18] IBM ILOG Cplex. 12.2 User's Manual. *ILOG*, 2010. See ftp://ftp.software.ibm.com/software/websphere/ilog/docs/optimization/cplex/ps_usrmancplex.pdf. Accessed 2017-05-16.

[19] Gurobi Optimization Inc. Gurobi Optimizer Reference Manual. 2016.

[20] List of optimization software. https://en.wikipedia.org/w/index.php?title=List_of_optimization_software&oldid=759681428, January 2017. Page Version ID: 759681428. Accessed 2017-03-02.

[21] William E. Hart. Python optimization modeling objects (Pyomo). In *Operations Research and Cyber-Infrastructure*, pages 3–19. Springer, 2009.

[22] Bethany Nicholson, John D. Siirola, Jean-Paul Watson, Victor M. Zavala, and Lorenz T. Biegler. Pyomo. dae: A Modeling and Automatic Discretization Framework for Optimization with Differential and Algebraic Equations.

[23] Federico Lozano Santamaría and Jorge M. Gómez. Framework in PYOMO for the assessment and implementation of (as)NMPC controllers. *Computers & Chemical Engineering*, 92:93–111, September 2016. doi: 10.1016/j.compchemeng.2016.05.005.

[24] Full saving ahead! - Mercedes-Benz. https://www.mercedes-benz.com/en/mercedes-benz/vehicles/trucks/full-saving-ahead/. Accessed 2016-11-22.

[25] Scania Group. Pressroom: Scania Active Prediction. https://www.scania.com/group/en/event/pressroom-scania-active-prediction/. Accessed 2017-02-14.

[26] I-See — Volvo Trucks. http://www.volvotrucks.us/powertrain/i-shift-transmission/i-see/. Accessed 2016-11-22.

[27] MAN Truck & Bus AG. MAN EfficientCruise® - More efficiency in long-haul or distribution transport — MAN Truck Germany. http://www.truck.man.eu/de/en/man-world/technology-and-competence/efficiency-systems/gps-assisted-cruise-control/GPS-assisted-Cruise-Control.html.

[28] Continental Automotive -eHorizon. http://www.continental-automotive.com/www/automotive_de_en/themes/passenger_cars/interior/connectivity/pi_ehorizon_en.html. Accessed 2017-02-14.

[29] Getting Started — Google Maps Elevation API. https://developers.google.com/maps/documentation/elevation/start. Accessed 2017-04-27.

[30] Y. Bengio, A. Courville, and P. Vincent. Representation Learning: A Review and New Perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1798–1828, August 2013. doi: 10.1109/TPAMI.2013.50.

[31] P. Smolensky. Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1. pages 194–281. MIT Press, Cambridge, MA, USA, 1986.

[32] Ruslan Salakhutdinov, Andriy Mnih, and Geoffrey Hinton. Restricted Boltzmann Machines for Collaborative Filtering. In *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, pages 791–798, New York, NY, USA, 2007. ACM. doi: 10.1145/1273496.1273596.

[33] Koray Kavukcuoglu, Marc'Aurelio Ranzato, and Yann LeCun. Fast inference in sparse coding algorithms with applications to object recognition. *arXiv preprint arXiv:1010.3467*, 2010.

[34] Laurens van der Maaten and Geoffrey Hinton. Visualizing Data using t-SNE. *Journal of Machine Learning Research*, 9(Nov):2579–2605, 2008.

[35] Jonathon Shlens. A Tutorial on Principal Component Analysis. *arXiv:1404.1100 [cs, stat]*, April 2014.

[36] S. Lafon, Y. Keller, and R. R. Coifman. Data Fusion and Multicue Data Matching by Diffusion Maps. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(11):1784–1797, November 2006. doi: 10.1109/TPAMI.2006.223.

[37] Stéphane S. Lafon. *Diffusion Maps and Geometric Harmonics*. PhD thesis, Yale University, 2004.

[38] Peter W. Jones, Mauro Maggioni, and Raanan Schul. Manifold parametrizations by eigenfunctions of the Laplacian and heat kernels. *Proceedings of the National Academy of Sciences*, 105(6):1803–1808, December 2008. doi: 10.1073/pnas.0710175104.

[39] Boaz Nadler, Stéphane Lafon, Ronald R. Coifman, and Ioannis G. Kevrekidis. Diffusion maps, spectral clustering and reaction coordinates of dynamical systems. *Applied and Computational Harmonic Analysis*, 21(1):113–127, July 2006. doi: 10.1016/j.acha.2005.07.004.

[40] Carmeline J. Dsilva, Ronen Talmon, Ronald R. Coifman, and Ioannis G. Kevrekidis. Parsimonious representation of nonlinear dynamical systems through manifold learning: A chemotaxis case study. *Applied and Computational Harmonic Analysis*, July 2015. doi: 10.1016/j.acha.2015.06.008.

[41] Lawrence Cayton. Algorithms for manifold learning. *Univ. of California at San Diego Tech. Rep*, pages 1–17, 2005.

[42] Ronald R. Coifman and Stéphane Lafon. Geometric harmonics: A novel tool for multiscale out-of-sample extension of empirical functions. *Applied and Computational Harmonic Analysis*, 21(1):31–52, July 2006. doi: 10.1016/j.acha.2005.07.005.

[43] Or Yair, Ronen Talmon, Ronald R. Coifman, and Ioannis G. Kevrekidis. No equations, no parameters, no variables: Data, and the reconstruction of normal forms by learning informed observation geometries. *arXiv:1612.03195 [nlin]*, November 2016.

[44] Or Yair and Ronen Talmon. Local Canonical Correlation Analysis for Nonlinear Common Variables Discovery. *arXiv:1606.04268 [cs, stat]*, June 2016.

[45] Tal Shnitzer, Ronen Talmon, and Jean-Jacques Slotine. Manifold Learning With Contracting Observers for Data-Driven Time-Series Analysis. *IEEE Transactions on Signal Processing*, 65(4):904–918, 2017.

[46] Eliodoro Chiavazzo, Ronald R. Coifman, Roberto Covino, C. William Gear, Anastasia S. Georgiou, Gerhard Hummer, and Ioannis G. Kevrekidis. iMapD: Intrinsic Map Dynamics exploration for uncharted effective free energy landscapes. *arXiv:1701.01513 [physics]*, December 2016.

[47] Matthew O. Williams, Ioannis G. Kevrekidis, and Clarence W. Rowley. A Data–Driven Approximation of the Koopman Operator: Extending Dynamic Mode Decomposition. *Journal of Nonlinear Science*, 25(6):1307–1346, December 2015. doi: 10.1007/s00332-015-9258-5.

[48] B. O. Koopman and J. v Neumann. Dynamical systems of continuous spectra. *Proceedings of the National Academy of Sciences*, 18(3):255–263, 1932.

[49] Bernard O. Koopman. Hamiltonian systems and transformation in Hilbert space. *Proceedings of the National Academy of Sciences*, 17(5):315–318, 1931.

[50] Marko Budišić, Ryan Mohr, and Igor Mezić. Applied Koopmanism. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 22(4):047510, December 2012. doi: 10.1063/1.4772195.

[51] David Ruelle and Floris Takens. On the nature of turbulence. *Communications in Mathematical Physics*, 20(3):167–192, September 1971. doi: 10.1007/BF01646553.

[52] Floris Takens. Detecting strange attractors in turbulence. In *Dynamical Systems and Turbulence, Warwick 1980*, pages 366–381. Springer, Berlin, Heidelberg, 1981. doi: 10.1007/BFb0091924.

[53] T. Berry, J. Cressman, Z. Gregurić-Ferenček, and T. Sauer. Time-Scale Separation from Diffusion-Mapped Delay Coordinates. *SIAM Journal on Applied Dynamical Systems*, 12(2):618–649, January 2013. doi: 10.1137/12088183X.

[54] Hans-Joachim Bungartz and Michael Griebel. Sparse grids. *Acta Numerica*, 13:147–269, May 2004. doi: 10.1017/S0962492904000182.

[55] William S. Cleveland and Susan J. Devlin. Locally Weighted Regression: An Approach to Regression Analysis by Local Fitting. *Journal of the American Statistical Association*, 83(403):596–610, 1988. doi: 10.2307/2289282.

[56] Cornelius Lanczos. An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. *Journal of Research of the National Bureau of Standards*, 45(4):255–282, October 1950.

[57] R. Lehoucq, D. Sorensen, and C. Yang. *ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*. Software, Environments and Tools. Society for Industrial and Applied Mathematics, January 1998. doi: 10.1137/1.9780898719628.

[58] William E. Hart, Jean-Paul Watson, and David L. Woodruff. Pyomo: Modeling and solving mathematical programs in Python. *Mathematical Programming Computation*, 3 (3):219–260, September 2011. doi: 10.1007/s12532-011-0026-8.

[59] The pip developers. Pip: The PyPA recommended tool for installing Python packages. Accessed 2017-07-13.

[60] Tyrus Berry and John Harlim. Variable bandwidth diffusion kernels. *Applied and Computational Harmonic Analysis*, 40(1):68–96, January 2016. doi: 10.1016/j.acha.2015.01.001.

[61] K. Fukunage and P. M. Narendra. A Branch and Bound Algorithm for Computing k-Nearest Neighbors. *IEEE Trans. Comput.*, 24(7):750–753, July 1975. doi: 10.1109/T-C.1975.224297.

[62] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An Algorithm for Finding Best Matches in Logarithmic Expected Time. *ACM Trans. Math. Softw.*, 3(3): 209–226, September 1977. doi: 10.1145/355744.355745.

Errata

Below follows a listing of minor adjustments and errata corrections, that were made since the time of submission of the thesis.

General corrections

- Table of Contents: Homogenization of indents.
- Section and subsection headings: Homogenization of capitalization.

Chapter 8:

- Table 8.1 (page 62): Correction of the units of column header ”**Distance (m)**” to ”**Distance (km)**”.
- Table 8.1 (page 62): Correction of column header and data interchange for columns ”**PID cost/m**” and ”**MPC cost/m**”.