ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

inspire iwf
Institut für
Werkzeugmaschinen & Fertigung

Automatic extraction of Partially Observable Markov Decision Processes from Simulink models

Oliver Stollmann

Bachelor Thesis
2011

Bastian Migge
PD Dr. habil. Andreas Kunz

Prof. Dr. Konrad Wegener

Abstract

This thesis presents a method for extracting Partially Observable Markov Decision Processes from Simulink models. The aim is to combine the modelling power of Simulink with the optimization advantages of POMDPs. An extraction tool allows engineers to model the system in graphical modelling tools such as Simulink, yet also permits non-myopic optimization of the modelled system by transforming it into a POMDP, an otherwise error-prone manual process. This transformation tool was produced as part of a project to develop an intelligent assistant for Waste-to-Energy plant managers. This text presents the theoretical background of dynamic systems, stochastic processes and Markov Decision Processes. It also introduces Simulink, a graphical dynamic system modelling tool. The contribution of this work is the study of and the development of an extraction algorithm, in MATLAB, that permits the transformation of arbitrary Simulink models into POMDPs. A secondary product of this research is the development of a validation tool, also written in MATLAB, necessary to judge the quality of the extracted POMDP. This thesis ends with an evaluation of the extraction approach and algorithm by studying the quality of a POMDP extracted from a Simulink model of a third-order Butterworth filter. The thesis shows that the extraction of Partially Observable Markov Decision Processes from Simulink models is, in general, possible. Nonetheless more research is required before the extracted Markov Processes can be used for optimization in an industrial setting.

IWF
Institut für Werkzeugmaschinen und Fertigung
Institute of Machine Tools and Manufacturing

POMDP controller for strategic incineration plant production planning

Keywords / Topics: Model Predictive Plant Control, Contorol under Uncertainty, Modelling, Planning

Abstract

This bachelor thesis investigates the application of a POMDP controller in the field of incineration energy production plants. To model a plant exemplarily as a Markov Decision Process (MDP), the control actions and cost as well as uncertain observations are identified. To evaluate the POMDP controller, the resulting strategy is compared to corresponding results of different predicitive control techniques.

Introduction

The ICVR works in the field of assistance systems, helping the user to make decisions based on simulating and forecasting dynamic systems under uncertainty. As part of the KTI project “WtE – Commercial Optimizer” we develop a controller that suggests operating instructions to the energy plant manager. The automatically generated non-myopic strategy is calculated using predictive control on the dynamic model of the plant and the energy markets. This enables to optimally map the production potential of the plant onto the market needs.

Content

The thesis shows the benefits and drawbacks of using POMDPs in strategic plant control. Comparing the POMDP controller with other stochastic model predictive control methods weights the benefits of considering uncertainty in the strategy versus the high computational efforts. This work will exemplarily show the potential value of POMDP controller in the field of industrial control.

One potential application is the guidance in splitting the produced thermo dynamical energy into the electricity grid and the district heating. The optimal economical strategy depends on production capacity and the market prices and penalties, and is not necessarily congruent to the optimal physical utilization.

Work packages

- Identify control actions of incineration plants
- Develop a MDP model of the financial reward and the system dynamics of the power plant
- Identify observations that indicate the hidden system state of the MDP (sensor model)
- Apply a POMDP planner to determine the optimal production policy offline to a Simulink model
- Compare the outcoming strategy with other (stochastic) model predicive control methods
- Find simplifications to enable online (re)planning for model adaption at runtime
- Document and present results

Requirements

We are looking for students, who would like to work at industrial driven projects in small research teams. Although skills in systems engineering, modeling (Simulink) and programming (C) are welcome, most important is the will to break into the given problem, develop solutions, and solve problems pro actively. Fulfilling the common academic processing is mandatory. You will document and present your results to the institutes’ members in a 20 minutes mid-term and final session.

Information & Administration

Bastian Migge, ETH Zentrum – CLA G19.1, bastian.migge@iwf.mavt.ethz.ch

PD Dr. habil. Andreas Kunz , ETH Zentrum – CLA G9, kunz@iwf.mavt.ethz.ch

inspire icvr
innovation center virtual reality

Acknowledgment

I would like to first thank Bastian Migge for his support. His knowledge of and experience with POMDPs provided me with a valuable mentor in the field. Additionaly his constant feedback on my presentations and this text have greatly improved its quality and focus.

I also thank Thomas Nescher for his valuable critique during the final presentation of this thesis. It instigated the creation of the validation tool.

Finally I would like to thank all members of the IWF's ICVR for their support with general and administrative problems.

Plagiarism Statement

Statement regarding plagiarism when submitting written work at ETH Zurich

By signing this statement, I affirm that I have read the information notice on plagiarism, independently produced this paper, and adhered to the general practice of source citation in this subject-area.

Information notice on plagiarism:

http://www.ethz.ch/students/semester/plagiarism_s_en.pdf

Place/Date	Signature

Contents

Abbreviations

1D One-Dimensional

2D Two-Dimensional

3D Three-Dimensional

TP Transition Probability

MDP Markov Decision Process

POMDP Partially Observable Markov Decision Process

SISO Single-Output/Single-Input

IO Input/Output

CPU Central Processing Unit

RAM Random-Access Memory

List of Figures

List of Tables

Introduction

This bachelor thesis was completed at the Institute of Machine Tools and Manufacturing of the Swiss Federal Institute of Technology. The goal of this work was the study of the feasability of and the development of an extraction tool that can transform a dynamic probabilistic non-linear Simulink model into a Partially Observable Markov Decision Process. The Institute of Machine Tools and Manufacturing's Innovation Center Virtual Reality has developed a number of POMDP-driven controllers, solving problems such as the parallax effect caused by the thickness of touch-screen displays.

This thesis occured as part of a large research project financed by the Swiss Commission for Technology and Innovation and ABB Switzerland. Project partners also include two Waste-to-Energy pilot plants and two smaller Swiss universities. The goal of the project is the development of a Waste-to-Energy plant-manager assistant using POMDP-control (similar research has already shown promise [BS]). In order to develop a POMDP-based controller, the existing Simulink models must be converted to POMDPs, which is exactly the goal of this thesis.

During the course of this research a prototype extraction algorithm was developed as well as a validation tool, required to assess the quality of the transformed dynamic system description. This thesis presents the theoretical foundation of this work in chapter 3. The methodology behind the developed extraction tool and the validation tool is then introduced in chapter 4. Concrete implementation details for both the extractor and the validator are provided in chapter 5. An evaluation of the extraction algorithm is presented in chapter 6 and a discussion thereof in chapter 7. Finally a conclusion and an outlook are given in chapter 8.

Motivation

Engineers successfully model complex physical systems using graphical modelling software. The visual nature of graphical models allows an easier transfer of process expertise to a computer representation of complex systems. Nonetheless these tools have drawbacks. Rule based representations of dynamic systems are hard to solve. Complex differential equations require intelligent solvers and computing power.

Contrastingly, dynamic systems modelled as Markov Processes provide a simply solvable, yet extremely unintuitive representation. Non-trivial Markov Decision Processes and even more so Partially Observable Markov Decision Processes cannot be designed using first-principle approaches such as those used to construct most graphical differential-equation-based models. However, once represented as such, Markov Processes are an extremely powerful tool for non-myopic optimization.

The motivation of this work is to provide a tool that can transform models produced by engineers using graphical modelling tools into a system representation that can more easily be used for non-myopic optimization, without the need for complex solvers and raw computing power. A successful automatic transformation tool would allow engineers to use the tools best suited to them, whilst still allowing for the non-myopic optimization of complex dynamic systems without the need for advanced and computationally-costly mathematical solvers. The following two paragraphs quickly present two different use cases of this representational transformation.

The first example is of an extremely complex non-linear dynamic system: a Waste-to-Energy plant. Waste-to-Energy plants provide an interesting optimization problem. The complex intertial incineration process must be managed optimally by taking into account two different cost/reward sources, the time- and temperature-dependent district heating system and the extremely dynamic electricity market. Using Partially Observable Markov Decision Processes for the optimization of such a complex system would provide two main advantages. Firstly, the non-myopic nature of MDP- and POMDP-based optimization allows for a farther optimization horizon, especially useful for the optimization of such an intertial plant. Secondly, the decoupling of the reward model from the system dynamics permits dynamic on-line optimization, that constantly takes into account changing market conditions, without re-simulating the complex dynamic system.

The second example is of a much simpler dynamic system: a window blind controller. The number of windows in a house and the costs do not allow for expensive simulation-software-based model predictive control systems. Optimization using Markov Decision Processes may, on the other hand, be possible using cheap embedded devices with vector processors. Optimization using Markov Decision Processes only requires matrix and vector multiplications, making optimizatin on cheap specialized embedded devices possible. Using an expensive graphical modelling tool, such as Simulink, engineers could model the window blind system and its associated reward system, then transform the system into a Markov Decision Process and finally provide thousands of homes with cheap and *smart* window blind controllers.

Background

The following chapter provides an introduction to the theoretical foundation of this work. The extraction of *Partially Observable Markov Decision Processes* from *Simulink* model requires an understanding of *dynamic systems*, *stochastic processes*, specifically *Markov Chains*, *Markov Decision Processes* and *Partially Observable Markov Decision Processes*, and finally *Simulink*. This chapter introduces each of these concepts or tools, providing examples where helpful. An understanding of these key concepts will facilitate the understanding of the next two chapters, *Methodology* and *Implementation*.

3.1 Dynamic Systems

This section covers deterministic and randomized dynamic systems in theory and gives a few intuitive examples of dynamic systems.

Dynamic systems are systems whose development over time can be described by a single or a set of mathematical equations. Although these equations may not always be symbolically solvable they describe the system's dynamics perfectly. Examples include the time-varying temperature of an object as it is placed in the oven or the voltage across an inductor.

A common example of dynamic systems is the ideal one-dimensional mathematical pendulum seen in Figure 3.1. By considering the forces on the pendulum or it's energy it is quite simple to deduce the following differential equation to describe the system [TM07]:

$$0 = \frac{g}{l} \cdot sin(\phi(t)) + \ddot{\phi}(t)$$

With the small-angles assumption

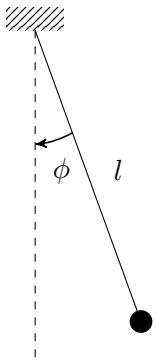

Figure 3.1: *1-dimensional mathematical pendulum*

$$sin(\phi(t)) \approx \phi(t)$$

the solution of the differential equation is

$$\phi(t) = \hat{\phi} \cdot sin(\sqrt{\frac{g}{l}} \cdot t + \phi_0)$$

where $\hat{\phi}$ is the semi-amplitude and ϕ_0 the phase at time $t = 0$. This equation describes the simplified dynamic system perfectly for all times.

An interesting property of the pendulum system is that given the same initial state and excitation (eg. initial angle at 20° and speed 0), the system will always respond the same way as time progresses. This property is called determinism and guarantees that given the same excitation and initial state the system will always develop identically over time. Systems that posess this property are called *deterministic dynamic systems* and differ greatly from the opposed *randomized dynamic systems*.

Randomized dynamic systems are dynamic systems with an element of *randomness*. The consequence of this is that the same initial conditions and excitation do not guarantee an identicial system response. A trivial example of a discrete randomized dynamic system is the following:

$$\begin{aligned} q[n+1] &= q[n] + x[n] + e[n] \\ y[n] &= q[n] + x[n] \end{aligned}$$

where $x[n]$ is the input, $e[n]$ is white noise and $y[n]$ the output at time n. If this system is provided with the same input signal twice, the resulting output signal is likely to be slightly different. This is caused by the inherit randomness of a white noise input.

The above example touches on an important point in the field of signal theory and system dynamics. The pendulum example also differs from the white noise example because the former is defined in *continuous time* whilst the latter is defined in *discrete time*. A continuous time system is defined for any time value $t \in T$, where T is the system's time space. A discrete time system is, on the other hand, only defined on a subset of the time line T, at discrete times $n \in N \subset T$.

A simple example to illustrate this point is the comparison of the continuous and the discrete sine functions:

$$\begin{aligned} y(t) &= sin(t), \\ y[n] &= sin(n) = sin(n \cdot T_s). \end{aligned}$$

Here $y[n]$ is the discrete-time version of the continuous time system $y(t)$. $y[n]$ is produced by *sampling* $y(n)$ with the sampling frequency $f_s = \frac{1}{T_s}$. This means that whilst $y(t)$ is defined for all times t, $y[n]$ is only defined for the times

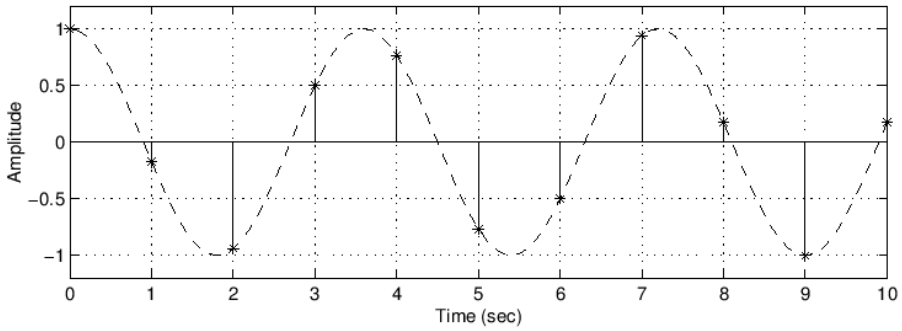

Figure 3.2: *Sampling of a continuous sine signal with sampling rate: $\frac{1}{2}$Hz [III]*

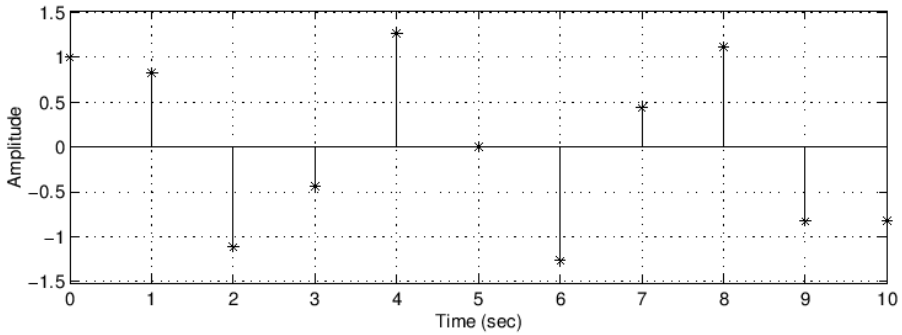

Figure 3.3: *Discrete signal produced by sampling a continuous sine signal with sampling rate: $\frac{1}{2}$Hz [III]*

$$
\begin{aligned}
& n \in N \\
& where \\
& \quad N = n \cdot T_s \, \forall \, (n \in \mathbb{Z}) \\
& \quad = [-\infty \cdot T_s, -(\infty - 1) \cdot T_s, \cdots, -1 \cdot T_s, 0, -1 \cdot T_s, \cdots, (\infty - 1) \cdot T_s, \infty \cdot T_s].
\end{aligned}
$$

It is easy to see here that a discrete time system contains less information than a continuous time system. Figure 3.2 shows a sine curve sampled every second (sample points are marked by stars). Figure 3.3 shows the resulting discrete signal. The information density difference between the orignal continuous signal and the sampled discrete signal becomes obvious. Nonetheless discrete time signals are prevalent because digital systems can only deal with digital (ie. discrete) signals.

The last important property of dynamic systems is the notion of *state*. A system's state is the smallest set of internal and/or external values that represent the entire state of the system. This means that a system's state completely describes that systems condition at a certain time. Coming back to the pendulum example it is clear that the entire system's condition can be described by two variables, the current angle $\phi(t)$ and the current angular velocity $\dot{\phi}(t)$. A definition of these two values at time t completely define the system's condition in past and future times $t \pm (\tau \in \mathbb{R})$. The set of all possible states that a system may find itself in is defined as the *state space* of that system.

Deterministic dynamic systems and *randomized dynamic systems* are two extremely useful mathematical constructs and serve as the basis of the more advanced theory of the next few sections.

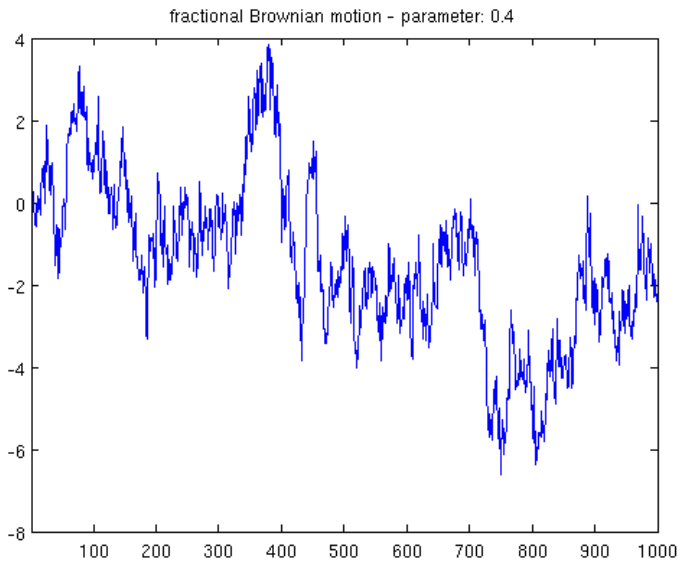

Figure 3.4: *Fractional Brownian motion with Hurst parameter of* 0.4

3.2 Stochastic Processes

A stochastic process is a set of random variables indexed by a parameter. If the indexing parameter is time and the random variables represent possible states then a random process describes a randomized dynamic system that reaches certain states at certain times. The formal definition of a time-indexed stochastic process is a collection $(X_t : t \in T)$ on a probability space where t is the time-index and X_t a random variable on the state space S.

A number of properties allow a more detailed classification of random processes: if the index set T is countable the process is *discrete* and if it is not countable the process is *continuous*, if the state space X is finite the process has a *finite state space* and if the random variable X_t represents values from a countable set the process values are *discrete* and otherwise *continuous*.

A common example of stochastic processes is Fractional Brownian motion as seen in Figure 3.4.

3.2.1 Markov Chains

A Markov Chain is a stochastic process that describes the dynamics of a probabilistic system by defining the conditional transition probabilities $Pr(X_{n+1} = x|X_n = x_n)$ between members of a finite or infinite state-space. Markov chains posess the Markov Property

$$Pr(X_{n+1} = x|X_1 = x_1, X_2 = x_2, ..., X_n = x_n) = Pr(X_{n+1} = x|X_n = x_n),$$

which states that the current conditional transition probabilities are dependent only on the system's current state.

If a Markov Chain's state-space is *finite* the transition probabilities between states can be represented in a *transition probability matrix* where the probability of going from state i to state j, $Pr(X_{n+1} = j|X_n =$

i), is equal to the (i, j) element of the matrix:

$$Pr = \begin{pmatrix} Pr(X_{n+1} = 1|X_n = 1) & Pr(X_{n+1} = 2|X_n = 1) & \cdots & Pr(X_{n+1} = N|X_n = 1) \\ Pr(X_{n+1} = 1|X_n = 2) & Pr(X_{n+1} = 2|X_n = 2) & \cdots & Pr(X_{n+1} = N|X_n = 2) \\ \vdots & \vdots & \ddots & \vdots \\ Pr(X_{n+1} = 1|X_n = N) & Pr(X_{n+1} = 2|X_n = N) & \cdots & Pr(X_{n+1} = N|X_n = N) \end{pmatrix}$$

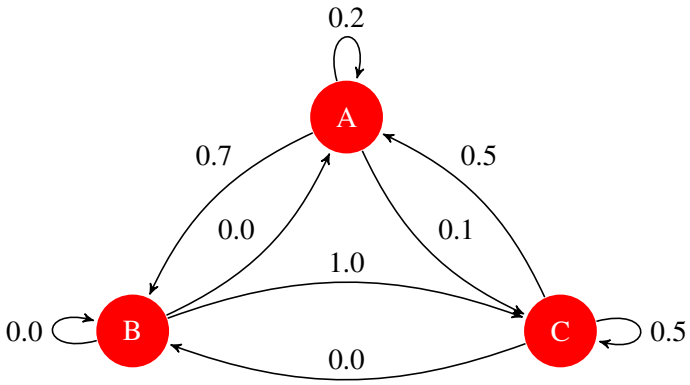

Figure 3.5: *Three-State Markov Chain*

Figure 3.5 shows an example of a three state Markov Chain. The transition probabilities matrix is

$$Pr = \begin{pmatrix} 0.2 & 0.7 & 0.1 \\ 0 & 0 & 1 \\ 0.5 & 0 & 0.5 \end{pmatrix},$$

and the state space S contains only three states:

$$S = \begin{pmatrix} A & B & C \end{pmatrix}.$$

Looking at this transition probability matrix or Figure 3.5, we can, for example, deduce that if the system is currently in state A the probability of transitioning to state B in the next step is 0.7, the probability of transitioning to state C is 0.1 and the probability of remaining in state A is 0.2. Interestingly, because the second row only contains a single non-zero value, which must thus be equal to 1, the transition from state B is entirely deterministic, meaning that it will always be the same.

3.2.2 Markov Decision Processes

Markov Decision Processes (MDPs) are an extension of Markov Chains and describe *controllable* probabilistic dynamic systems. They are defined as a four touple (S, A, P, R) with:

- S: set of states,
- A: set of actions,
- P: conditional transition probabilities,
- R: rewards.

MDPs are used to model probabilistic systems that can be influenced through decision-taking. These decisions are represented as actions and have a direct influence on the transition probabilities of the system. This idea of actions influencing transition probabilities can also be interpreted as *systems with uncertain actions*. Transition probabilities are now three-dimensional and depend not only on the current state, but also on the action being taken; $Pr(X_{n+1} = x_n | X_n = x, a_n = a)$ is the probability of going to state x_n in the next step given that the system is currently in state x and action a has been chosen. Although *next step* is usually associated with a change in time (ie. an advance on the time line), it can, in fact, mean sequential change in any kind of sequential process.

Because MDPs are not only used as a representational form of dynamic systems, but also as an optimization tool, it introduces the notion of rewards. Every transition probability is paired with a reward, or cost (negative reward), value; $R(s, s', a)$ is the immediate reward (scalar) that the system receives when it transitions from state s' to state s given that action a was chosen. Note that rewards are not inherently probabilistic, but an MDP models a *probabilistically rewarding system* indirectly through the stochastic nature of the transition probabilities.

Markov Decision Processes are used to optimize decision making. The combination of a system description and an associated reward model allows the computation of an optimal decision to take in a given situation. The aim of this optimization is to produce a *policy*, $\pi(s)$, that defines exactly which action should be taken, if the system finds itself in a certain state, to maximize the overall reward.

The computation of an optimal policy requires the definition of *optimality*. In most cases optimization simply aims for the maximization of rewards (or minimization of costs) over a certain decision span. This optimization goal is defined in a so called reward function, the most common of which is the *expected discounted total reward* (infinite horizon),

$$\sum_{t=0}^{\infty} \gamma^t R_{a_t}(s_t, s_{t+1}),$$

with:

- γ: discount factor, where $\gamma \in (0, 1]$,
- $R_{a_t}(s_t, s_{t+1})$: reward received in $t + 1$ for taking action a_t from state s_t at time t.

The *expected discounted total reward* seen above is one of the four traditional reward functions, defined in [SB10] as: the finite criterion, the γ-discounted criterion , the total reward criterion and the average reward criterion. The *expected discounted total reward* represents the idea that the decision maker values the total sum of rewards, but prefers rewards received earlier to rewards received later, ie. *rewards are discounted over time*. This reward function has a valuable property of being non-myopic. It takes into account not only the reward received in the next step, but looks far into the future taking into account that actions that are highly rewarded in the short-term may in fact be the wrong decision because of their effect in the long-term.

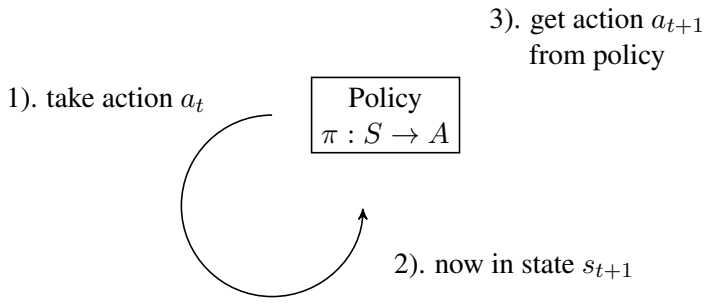

Figure 3.6: *MDP control loop*

Given an MDP and a reward function, an optimal policy can be computed. The process of computating such a policy, so-called *planning*, can be undertaken using either *linear programming* or, more commonly, *dynamic programming* (value- and policy-iteration). An in-depth analysis of the different policy computation algorithms is out of the scope of this text, but is covered in detail by the field's literature [Put94][SB10]. The result of the policy computation is, as described above, a policy $\pi(s) : S \to A$ that maps each element of the set of states S to an action $a \in A$ and thus provides a decision maker with an *optimal decision* to take when the controlled system finds itself in a certain state.

Although control and optimization are not the motivation of this work it is important to look at the use of MDPs as control and optimization tools. Figure 3.6 shows the MDP's computed policy's position in the decision-taking control loop of an abstract controller. The controller has taken some decision at time t. He then received an information as to what state s_{t+1} the system is in now, at time $t + 1$. He then uses the policy $\pi : S \to A$ to determine the optimal action a_{t+1} to take at this time $t + 1$. This loop ensures that the system is *controlled in an optimal way*.

3.2.3 Partially Observable Markov Decision Processes

A Partially Observable Markov Decision Process (POMDP) is a further extension of a Markov Decision Process, the difference being that the decision maker can no longer perfectly observe the entire system state, but must instead deal with partial observations when making decisions. It is formally defined as a six-tuple (S, A, O, T, Ω, R) with:

- S: set of states,
- A: set of actions,
- O: set of observations,
- T: conditional transition probabilities,
- Ω: conditional observation probabilities,
- R: rewards.

An observation is any system output that the decision maker can *observe*. MDPs assume that the decision maker has the ability to *see* what state the system is currently in, whereas POMDPs make no such assumption, relying instead on partial observations. This approach more realistically models reality where systems are rarely completely observable. The notion of actions and rewards remains the same with POMDPs.

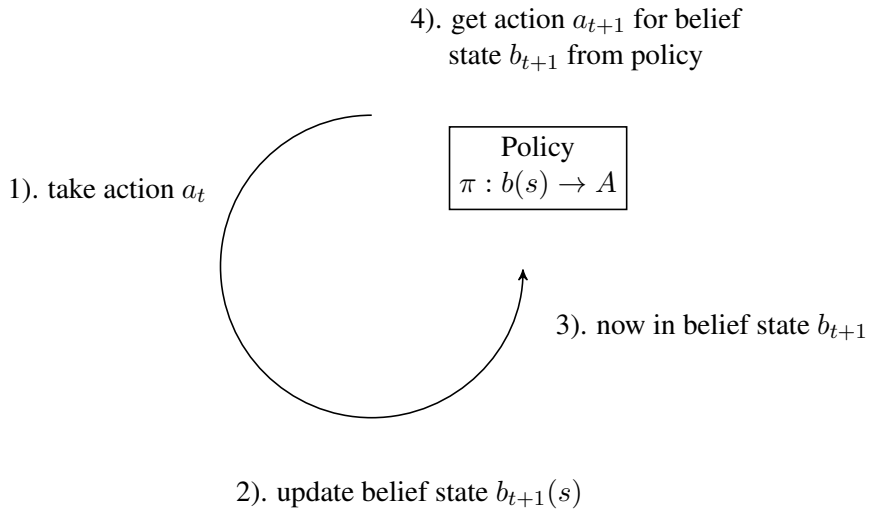

Figure 3.7: *POMDP control loop (offline planning)*

Optimization using POMDPs is an order of magniture more complicated than with MDPs [SB10][Lit96]. The product of an optimal planning procedure is no longer a policy $\pi : S \rightarrow A$, but a policy $\pi : P(S) \rightarrow A$, that maps probability distributions over the state-space to actions. Because exact knowledge of the system's state is no longer available, the controller must maintain a belief-state, a probability distribution over the state-space, representing the probabilities of the system currently being in a certain state. Figure 3.7 shows the traditional control loop of an abstract POMDP based controller. After action a_t is taken at time t the controller must update his belief-state for time $t+1$, meaning he must update the distribution of probabilities that the system will be in a given state s_{t+1} at time $t+1$. The update makes use of both the transition probabilities $T(s_{t+1}|s_t, a_t)$ and the observation o_{t+1} the controller receives at time t+1. The belief-state at time $t+1$,

$$b_{t+1}(s) = F(o_t, a_t, b_t(s)),$$

is then used to query the policy for the action a_{t+1} to take at time $t+1$ (function F is defined in [SB10]). It is immediately obvious that the policy produced by a POMDP planner is now 3-dimensional, as opposed to the 2-dimensional policy an MDP produces. Instead of mapping from finite *states* to *actions* the POMDP policy must map *distributions over states* (ie. belief states) to actions. This adds an enormous computational cost to planning using POMDPs. Solving POMDPs (ie. producing a policy) is often intractable because of the amount of necessary calculations. Therefore much research has been done with alternative, mostly approximative, methods [Han98][Lit96].

The above cost is based on the fact that the policy must be able to map all possible belief-states to actions. In order to overcome this computational cost POMDPs are sometimes used for *online planning*. In this case the policy is not computed in advance for all possible belief states, but rather computed at each time step for a single belief state, the one the system is in when the controller must make a decision. This approach is introduced in [SB10], covered in detail in [RPPCD08] and unfortunately outside the scope of this text.

Although POMDP control is associated with high computational costs it is nonetheless an approach with merit. The distinction between system state and observations reflects the reality of most complex systems and the resulting policies reflect this realism by taking into account that what is observed may differ from what is.

3.3 Simulink

Simulink is a commercial modelling and simulation tool developed by MathWorks. It is an industry standard in the field of control engineering. Models are created graphically in a block-based user interface and simulation results can be easily be analysis with plots or mode advanced tools. Additionally a large number of internal and third-party libraries further simplify modelling, especially in specialized fields such a music or aerospace.

Simulink is also tightly integrated in MATLAB, MathWorks' commercial numerical computation tool, which is widely used in academia and industry in many different fields. Simulink runs inside the MATLAB environment and can thus easily be controlled, tested or extended using the MATLAB programming language.

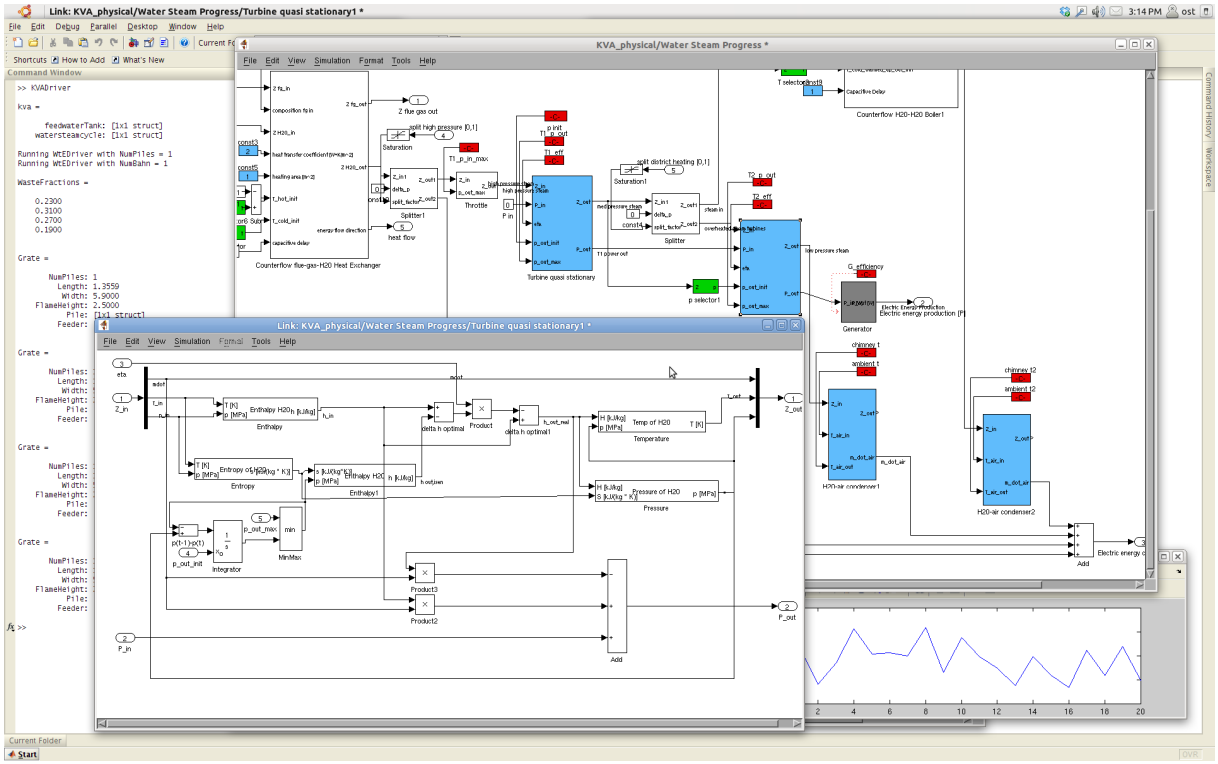

Figure 3.8: *Screenshot of Simulink*

Simulink is especially useful for engineers because no programming knowledge is required even for modelling complex dynamic systems (eg. power plants). Default configuration options and an entirely graphical interface hide most implementation details (simulation details, solver types, etc). A graphical modelling interface is also useful because it allows a more intuitive understanding of the model, unlike the large mathematical matrices used in Markov Chains, Markov Decision Processes or Partially Observable Markov Decision Processes. Figure 3.8 shows an example of the MATLAB environment, a Simulink model and plots of simulation results.

Because this work deals extensively with Simulink a short introductory example may be helpful. Figure 3.9 shows the pendulum model from section 3.1 (see Figure 3.1) implemented in Simulink through a series of integrators, a sin function and two contant parameters, the gravitational acceleration g and the length of the pendulum l. The initial phase is set to zero and an initial condition can be defined in either

of the two integrators. For this simulation the angular velocity was used as an initial condition and set to $1\frac{rad}{s}$. Figure 3.10 shows two plots of the system response once with a gravitational constant of $9.8\frac{m}{s^2}$ and once with a gravitational constant of $19.6\frac{m}{s^2}$.

As can be seen in this example Simulink provides a very intuitive platform for modelling and simulating dynamic systems. Although the above example model is entirely deterministic, it is trivially easy to add randomness to deterministic models in Simulink.

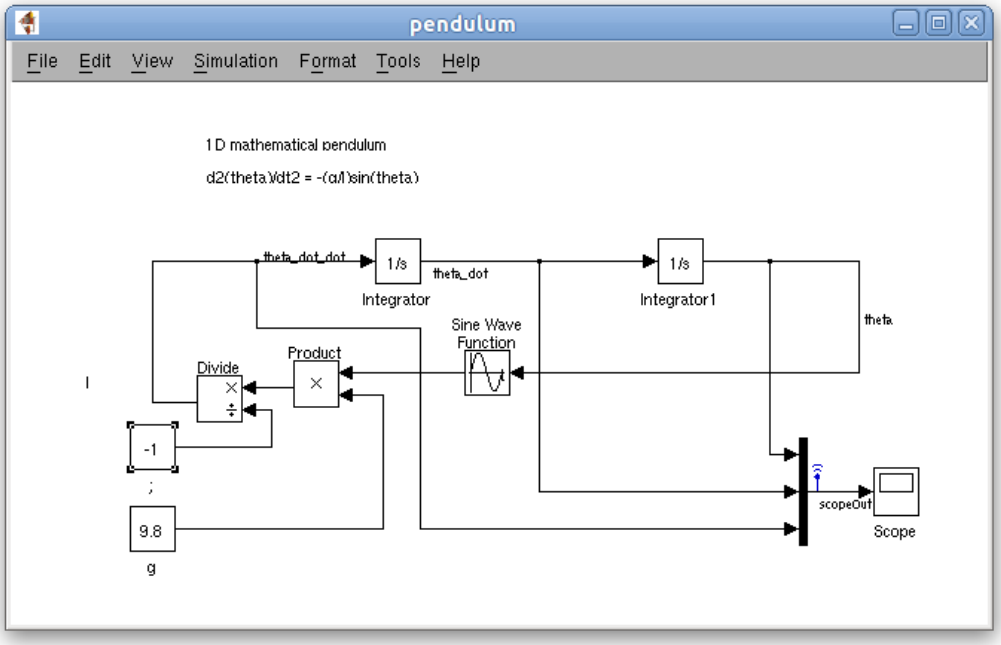

Figure 3.9: *1-dimensional mathematical pendulum model*

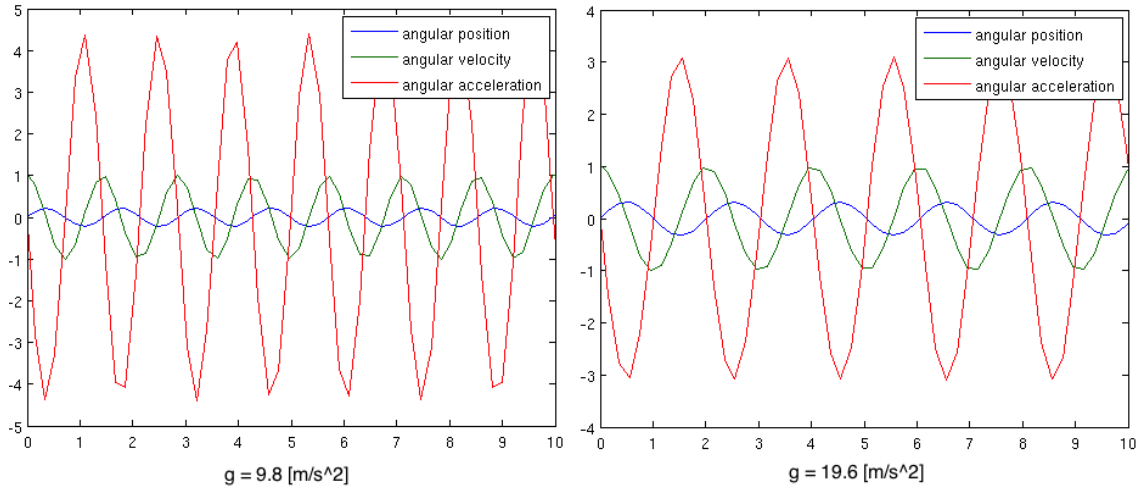

Figure 3.10: *1D mathematical pendulum response*

Methodology

This chapter documents the ideas and approaches behind the implementation of the *extraction algorithm* and the *validation*. It provides a high-level overview of the ideas and the problem/solution tuples that defined the final implementation.

4.1 Extraction

Simulink models represent dynamic systems in a number of different ways. Systems can be described through a graphical representation of differential equations (see example in section 3.3). Systems can also be described by transfer functions or state machines. Simulink offers many different representational forms. Almost all of these representational forms have in common that they represent *rules* of some sort. When asked to simulate these systems Simulink solves these *rules* in real time and produces the system response. MDPs and POMDPs are much simpler constructs that do not require real-time solvers because the system dynamics is represented as simple state transitions probabilities. Given an MDP or a POMDP representation of a dynamic system, the simulation thereof is merely a question of random sampling.

In order to build these transition probability matrices the *extractor* simulates and observes the given Simulink model multiple times and with different inputs to build up the POMDP's transition probability matrix. In parallel the *extractor* also observes the Simulink model's reward and observation outputs and incrementally builds the reward matrix and the conditional observation probability matrix. The following sections go through the extractor's different functions and discusses them from a high-level point of view.

The chapter ends with two concrete examples that aim to facilitate the understanding of this approach by using real, albeit simple, models and actual numerical values to extract Partially Observable Markov Decision Processes.

4.1.1 Approach

The approach chosen for the extraction of Partially Observable Markov Decision Processes from Simulink models is a simple one. If a model is simulated a large enough number of times (see section 4.1.9) from the same source state and given the same action, the transition probability for reaching states in the next step given the source state $s_t = s$ and the action $a_t = a$,

$$Pr(s_{t+1} = s' | s_t = s, a_t = a),$$

can be extracted simply by counting the number of times certain sink states, s_{t+1}, were reached and normalizing the count vector. This is exactly what the extraction algorithm does for every possible source state given every possible action. In parallel, reward and observation outputs are observed to build the reward model and the conditional observation probabilities described in more detail in sections 3.2.2 and 3.2.3. The example in section 4.1.13 presents this process in more detail.

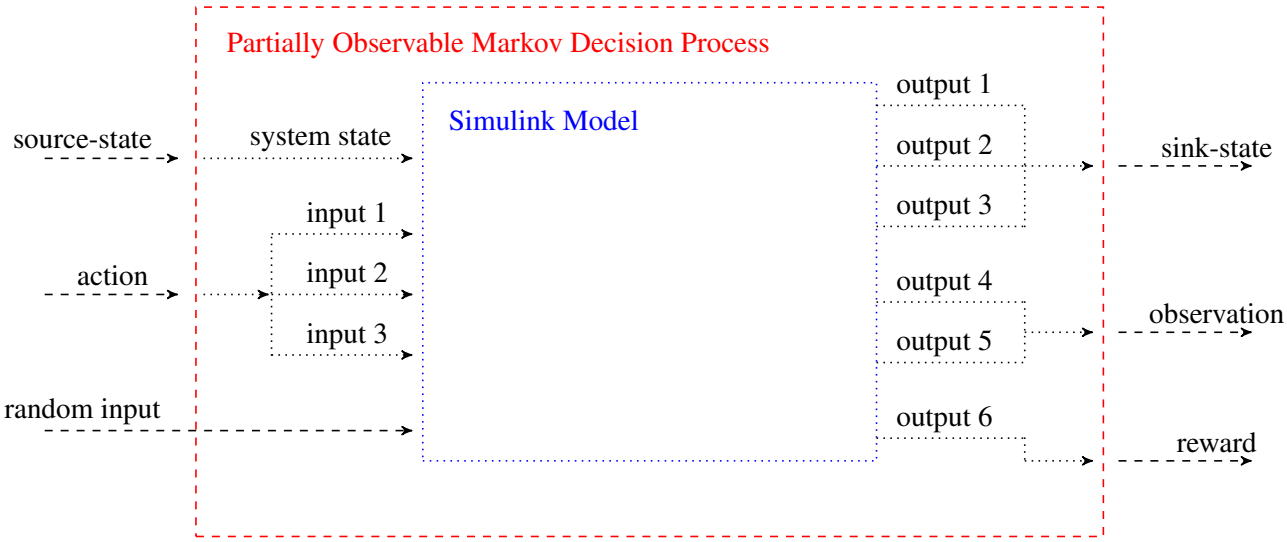

Figure 4.1: *Simulink/POMDP Interfaces*

Given a simple input/output model, boundaries for the inputs and boundaries for the outputs, the extractor will extract a POMDP by simulating the model and observing its response. This extraction involves producing actions from input value boundaries, identifying states and building a state space, identifying observations and building an observation space and handling simulation errors and states outside the permitted bounds.

4.1.2 Simulink and POMDP Interface

Using the schematic in figure 4.1, this section aims to provide an overview of the interfaces between the Simulink model, the extraction algorithm and the Partially Observable Markov Decision Process.

As discussed in section 4.1.1, the extractor works by providing the Simulink model with inputs and observing it's outputs. In order to transform the Simulink model into a POMDP model, Simulink's notion of inputs, outputs and system states must be mapped to the POMDP's base constructs, namely, states, actions, observations and rewards. Figure 4.1 and the rest of this section describe the required transformations between Simulink and POMDP constructs.

On the left side of figure 4.1 one can observe that the extraction algorithm provides three inputs. Firstly it provides a source state and an action and secondly a random number input. The source-state and the action are POMDP constructs, whilst the random input is simply the model's randomness provider (see section 5.1.4). Because the Simulink model is based on individual inputs and not on actions, the extraction algorithm must convert the given action into a set of valid Simulink inputs before feeding these to the model. Because the action set is created from the cartesian product of all permitted input values, the action to input transformer simply splits the action into its individual inputs (see section 4.1.7 for more details about the action set). Similarily, whilst the POMDP works with state indices, the Simulink model must be provided with an actual Simulink simulation state object. These simulation state objects are saved whenever new states are discovered and the extraction algorithm then merely feeds them back to the Simulink model when the discovered states serve as a source states.

The individual Simulink outputs and their POMDP equivalents can be seen on the other side of figure 4.1. For the outputs the transformation is inverted, as Simulink outputs must be transformed into POMDP

states, observations and rewards. The first n_s outputs are state outputs, meaning that they identify the state the system is in (see section 4.1.3). n_s is an extraction configuration parameter that defines the number of state outputs (see figure 6.5 for an example of a model prepared for extraction and figure 5.1 for an example configuration, showing amongst other things the n_s setting). The next n_o outputs are observation outputs, meaning that they define the observation state. n_o is also a configuration parameter that must be defined before an extraction occurs. The last output is interpreted as the immediate transition reward (see section 3.2.2) and is taken as-is to build the reward model.

The conversion between POMDP values and Simulink values occurs before and after *every* simulation as the POMDP's conditional transition probability matrix, conditional observation probability matrix and reward model are gradually built.

4.1.3 System State and System Output

As described in section 3.1 a system's state is defined as the smallest possible set of internal and/or external values that represent the system's condition at a certain time. This notion of state in dynamic systems maps exactly onto the idea of states of stochastic processes. Unfortunately an extraction based on applying different inputs to a system and observing it's outputs does not provide *state information* in the above sense, because the relationship between system output and system state is not biconditional. This distinction entails the biggest simplification made by the product of this work, the POMDP extractor.

The extractor does not distinguish between system output and system state as it makes the assumption that *if two system responses are equal the two states of the system are equal*. This simplification was made for two reasons. Firstly, although Simulink offers a way of saving a simulated system's internal state, it does not provide a simple way of comparing two system states. Secondly this simplifications greatly decreases the size of the extracted POMDPs state space. Chapter 7 touches upon the difficulty associated with extremely large state spaces.

A short example may make this rather large simplification clearer. The mathematical pendulum example (see Figure 3.1) introduced in section 3.1 provides a good basis. As discussed in section 3.1 the pendulum system's condition can be described by only two variable, the angular position $\phi(t)$ and the angular velocity $\dot{\phi}(t)$ of the pendulum. Knowing these two values, the system's past can be reconstructed and it's future predicted. If a POMDP extraction of this system were configured with only the angular position $\phi(t)$ as an output, the extractor would interpret the two states,

$$x_{1,t=\tilde{t}} = \left(\phi_1(\tilde{t}) = \tfrac{\pi}{4},\ \dot{\phi_1}(\tilde{t}) = 0.8\ [\tfrac{rad}{s}]\right)$$
$$x_{2,t=\tilde{t}} = \left(\phi_2(\tilde{t}) = \tfrac{\pi}{4},\ \dot{\phi_2}(\tilde{t}) = -1.3\ [\tfrac{rad}{s}]\right),$$

as equal because only the values $\phi_1(\tilde{t})$ and $\phi_2(\tilde{t})$ would be compared. With this simple example the dangers associated with ignoring the ramifications of this simplification become strikingly obvious. The pendulum in state x_1 will in the future swing in one direction, whilst the pendulum in state x_2 will swing in the opposite direction, clearly a different development, yet deemed equal by the extraction algorithm. In this case the problem could be overcome by defining a second output, the angular velocity $\dot{\phi}(t)$, that would then make the relationship between system state and system output biconditional. With such a configuration the extractor would no longer deem x_1 and x_2 to be equal system states and thus produce a more realistic POMDP.

A more detailed analysis of the ramifications of this simplification can also be found in chapter 7.

4.1.4 Output Discretization

Even though the extraction's source model may be a *continuous* system, the POMDP extractor produces a *discrete* POMDP. In order to extract a discrete POMDP from a continuous model the output values must be discretized, or more accurately quantitized. A simple example would be the sine function system,

$$y(t) = sin(x(t)),$$

where, even though the sine functions codomain is limited, an infinite number of outputs exist. This stems from the fact that the number of values in the range $[-1, 1] \in \mathbb{R}$ is infinite in a continuous system. In order to produce a finite state space, the POMDP extractor must thus convert the continous outputs to discrete outputs. This process is sometimes referred to as *binning*, whereby ranges of continuous values are lumped together into a single discretized value.

The quantitization approach chosen for the POMDP extractor is based on *rounding*. For each configured output i, the configuration must also contain a rounding parameter $n_i \in \mathbb{Z}$, which is then used to round Simulink output values to the nearest multiple of 10^{n_i}. This means that given a rounding parameter of 2, the values $x_1 = 23.1$, $x_2 = 3043$, $x_3 = 100000$ and $x_4 = 100001$ would be discretized to $\hat{x}_1 = 0$, $\hat{x}_2 = 3000$, $\hat{x}_3 = 100000$ and $\hat{x}_4 = 100000$, respectively.

The quantitization parameters directly influence the size of the POMDP's state-space and this effect is discussed in more detail in section 4.1.8.

4.1.5 Output Boundaries

The extractor's output is a *finite* POMDP. Simulink models may, on the other hand, have infinite domains. A simple example would be the following system,

$$y(t) = t,$$

which obviously has a codomain of the same size as the domain, so potentially infinite. In order to produce a POMDP with a *finite* state space, the extractor requires boundary values for each defined output. Each output value must be given both a minimum and a maximum value that it may not exceed. During the extraction, simulation outputs, and their associated state, are discarded if any one of the output values are outside the defined boundaries.

This approach does however present a problem. During simulations from a single source state given a single action the system response may sometimes lie within and sometimes outside the boundary. This problem is illustrated in table 4.1, which contains the results of 10 fictitious simulations from the same source state and using the same action. The stochastic dynamic system sometimes responds with values within and sometimes with values outside of the defined output boundaries. Simply ignoring simulations that result in states outside of the defined boundaries would not accurately reflect the system's dynamics.

The solution used by the extraction algorithm is the use of two fictitious states. The first of these is discussed in section 4.1.6, but the second one represents exactly the above discussed case of reaching a state with values outside the permitted boundaries. This means that the simulation results of table 4.1 can be interpreted accurately by including the probability of reaching the *out-of-bounds state*. The out-of-bounds state is never used as a source state and has a transition probability of 1 of not changing for any action. In this case the probability of reaching the out-of-bounds state is:

$$P(s' = out - of - bounds | s = 22, a = 6) = 0.3.$$

Simulation Run	Source State	Action	Sink State
1	22	6	68
2	22	6	67
3	22	6	122
4	22	6	out-of-bounds
5	22	6	68
6	22	6	68
7	22	6	out-of-bounds
8	22	6	out-of-bounds
9	22	6	67
10	22	6	68

Table 4.1: *Simulation with out-of-bounds sink states*

The reward gained for reaching the out-of-bounds state must be configured manually in the extrator's parameters.

4.1.6 Simulation Errors

Similar to the above discussed problem of reaching states outside the permitted boundaries, simulations may also sometimes simply result in errors. Unacceptable input values or combinations thereof, model instability or failed assertions may sometimes lead to simulation errors. Because of the probabilistic nature of the extractor's source models, it is also possible that only a subset of a set of simulations from the same source state and using the same action results in errors. Similarily to the problem of reaching out-of-bounds states, the occurence of errors cannot simply be ignored, but must rather be reflected in the transition probabilities, observation probabilities and reward model of the extracted POMDP.

The solution to this problem is the introduction of a fictitious *error state*. This state does not ever serve as a source state for simulations, but is used as sink state for simulations that fail due to errors. As with the out-of-bounds state discussed in section 4.1.5, the error state has a transition probability of 1 not to change for any action. As such, simulation errors can accurately be accounted for in the POMDPs transition probabilities, observation probabilities and reward model.

The reward associated with reaching the fictitious error state is part of the extractor's configuration parameters.

4.1.7 Inputs and Actions

MDPs and POMDPs model *controllable* dynamic systems, where a decision taker can influence the system's development over time. Simulink models usually have a single or multiple inputs that are sampled at every time-step and used as input for the given system. In order to extract transition probabilities that

depend on the action chosen by the decision maker, simulations must be observed for each of the possible actions. This means that for every state the system may find itself in, simulations must be run with every possible action a decision maker may choose to take. In order to guarantee that the MDP or POMDP will be able to represent the dynamics of the system correctly this means that every possible permutation of permitted input values must be used during the extraction.

A simple example of this can be made with the *ideal gas law* system,

$$p \cdot V = n \cdot R \cdot T,$$

where $p\ [Pa]$ is the pressure, $V\ [m^3]$ is the volume, $n\ [mole]$ is the mole quantity, $R = 8.314\ [J \cdot K^{-1} \cdot mol^{-1}]$ is the universal gas constant and $T\ [K]$ the temperature. Although this system is not dynamic — it does not change over time — it is sufficient in this context.

If a conversion of this system to an MDP or a POMDP were necessary with the following configuration,

- - inputs: pressure p, volume V, mole quantity n,
- - output: temperature T,

the action set of an MDP or a POMDP would be defined as the cartesian product of all input sets. If the maximum of each input x were defined as x_{max} and it's minimum as x_{min}, the action set would be:

$$\begin{aligned}
A &= \Pi \times \Lambda \times \Gamma \\
&= \{(p, V, n) \mid p \in \Pi,\ V \in \Lambda,\ n \in \Gamma\}, \\
where& \\
\Pi &= [p_{min}, (p_{min} + \pi), (p_{min} + 2 \cdot \pi), \cdots, (p_{min} + (N-1) \cdot \pi), p_{max}] \\
\Lambda &= [V_{min}, (V_{min} + \lambda), (V_{min} + 2 \cdot \lambda), \cdots, (V_{min} + (N-1) \cdot \lambda), V_{max}] \\
\Gamma &= [n_{min}, (n_{min} + \gamma), (n_{min} + 2 \cdot \gamma), \cdots, (n_{min} + (N-1) \cdot \gamma), n_{max}] \\
\pi &= \frac{p_{max} - p_{min}}{N_p - 1} \\
\lambda &= \frac{V_{max} - V_{min}}{N_V - 1} \\
\gamma &= \frac{n_{max} - n_{min}}{N_n - 1}
\end{aligned}$$

with N_i being the number of different input values required between the maximum and minimum values of input x_i (see section 4.1.13.2 for an example). The cardinality of A (ie. the number of actions $a \in A$) is

$$|A| = \prod_{i \in I} N_i.$$

An example action set with numeric values could be

$$A = \begin{pmatrix} (p = 0.0, V = 0.0, n = 0.1) \\ (p = 0.0, V = 0.0, n = 0.3) \\ (p = 0.0, V = 0.0, n = 0.5) \\ \vdots \\ (p = 4.3, V = 2.0, n = 0.9) \\ (p = 4.3, V = 3.0, n = 0.1) \\ (p = 4.3, V = 3.0, n = 0.3) \\ \vdots \\ (p = 9.9, V = 9.0, n = 0.5) \\ (p = 9.9, V = 9.0, n = 0.7) \\ (p = 9.9, V = 9.0, n = 0.9) \end{pmatrix},$$

$$\begin{aligned} where = & \\ p_{min} &= 0.0,\ p_{max} = 9.9,\ \pi = 0.1,\ N_p = 100 \\ V_{min} &= 0.0,\ V_{max} = 9.0,\ \lambda = 1.0,\ N_V = 10 \\ n_{min} &= 0.1,\ n_{max} = 0.9,\ \gamma = 0.2,\ N_n = 5 \end{aligned}$$

In this case the number of actions comes to

$$|A| = \prod_{i \in (p,V,n)} N_i = N_p \cdot N_V \cdot N_n = 100 \cdot 10 \cdot 5 = 5000.$$

The action set produced in this fashion is then used by the extraction algorithm, described above in sections 4.1.1 and 4.1.9.

4.1.8 State Discovery

Although the quantitization of an extraction's legal output values completely defines the extracted POMDP's state space, the extractor does not create the n-dimensional state space in advance. The reason for this lies in the difference between the *potential* and the *reachable state space*.

The *potential* state space of a POMDP is defined by the output boundaries and the discretization parameters. The following short example, with two outputs a and b and boundaries,

$$\begin{aligned} a_{min} &= 0.0 \\ a_{max} &= 1.0 \\ b_{min} &= 1000 \\ b_{max} &= 2000, \end{aligned}$$

illustrates this point. Given these boundaries and discretization parameters the extractor's discretization function may produce the following discrete value buckets for each output,

$$a \in [0.0, 0.5, 1.0]$$
$$b \in [1000, 2000],$$

meaning that the system could reach the following six states:

$$x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{pmatrix} = \begin{pmatrix} a = 0.0 & b = 1000 \\ a = 0.0 & b = 2000 \\ a = 0.5 & b = 1000 \\ a = 0.5 & b = 2000 \\ a = 1.0 & b = 1000 \\ a = 1.0 & b = 2000 \end{pmatrix}.$$

This is the POMDP's *potential state space*, meaning that an extraction may reach all those states but no more. The *reachable* state space is however often much smaller. The limiting factor is the dynamic system itself. Certain systems may never reach some of the states in their potential state space. It may, as a matter of fact, even reach only a tiny subset of it. As such, the extractor is implemented as a *discovering* system that continuously discovers new states and thus builds up its state space gradually.

The only negative consequence of this difference between the *potential* and the *reachable state space* is that only a maximum value prediction can be made of the number of simulations required for an extraction. Experience has, however, shown that the ratio of reachable states to potential states can vary greatly, reducing the value of these maximum time predictions. Such a prediction may be ten years although in reality an extraction may only take a few weeks.

4.1.9 Probabilistic Simulations

Markov Processes model probabilistic systems. In order to extract the correct probabilities from the source model, the extractor must run a large enough number of simulations with the same source state and the same action. The number of simulations required to extract the accurate probabilistic nature of the system depends on the system and its inherit randomness. This value cannot be automatically deduced from a system without knowledge of its internal dynamics and must thus be defined as part of the extractor's configuration parameters. The discussion in chapter 7, discusses the effect of different numbers of simulation runs on the quality of the extracted Markov Process in representing a source model's actual nature.

4.1.10 Time Steps and Decision Epochs

Computer simulations occur in artificial time. Although a chemical reaction may take hours, the simulation thereof could take seconds. In order to correctly represent the development of time in relation to the development of a dynamic system, computer simulation use a relative measure of time, so-called

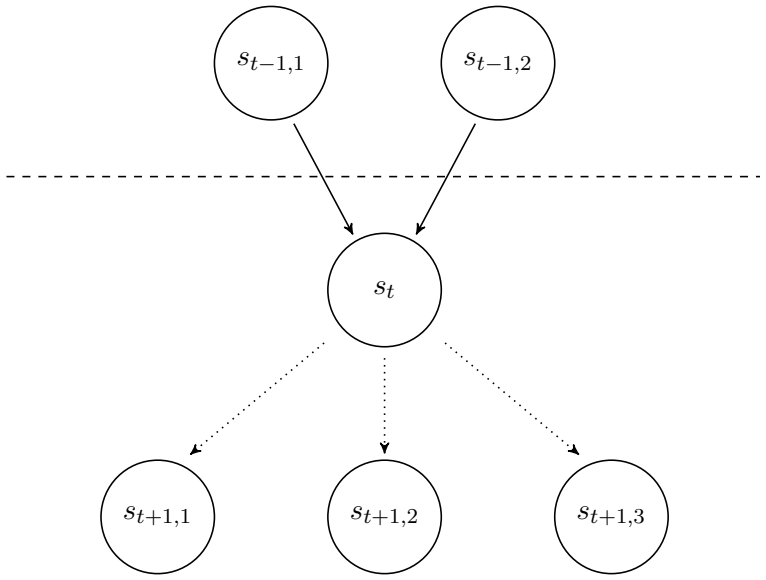

Figure 4.2: *Markov Property*

time steps. The conversion of time steps to real time depends on the model. Certain models may model a minute of real time as a single time steps, whilst other may choose to represent only nano seconds as time steps. This conversion ratio is an inherit property of the model and remains constant for all parts therof. This allows scientists to map simulated changes to changes in reality.

Markov Decision Processes represent time as *decision epochs*. The transitions defined in the conditional transition probabilities matrix introduced in section 3.2.2 occur over a single epoch. Depending on the time granularity of an MDP or a POMDP, a decision epoch may represent a minute, an hour or any other real time value. Section 4.1.11 discusses in more detail the relationship between the Markov property and the real time length of an artifical decision epoch.

The extraction of MDPs or POMDPs from Simulink models thus requires a conversion between *time steps* and *decision epochs*. The approach to this conversion is simple, yet its ramifications significant (see section 4.1.11). The extactor required a simple integer conversion value between time steps and decision epochs. The most granular extraction occurs with a conversion rate of 1, whereby a single decision epoch represents a single simulation time step. Greater conversion values mean that multiple time steps represent a single decision epoch in the extracted POMDP.

Although the implementation of this conversion is simple the conversion ratio can strongly influence the dynamics of the extracted system.

4.1.11 Markov Property and Model Time Lag

The Markov Property was introduced in section 3.2.1. It states that a Markov Chain's transition probabilities at time t depend only on the state at time $t-1$. Formally it is defined as:

$$Pr(X_{n+1} = x | X_1 = x_1, X_2 = x_2, ..., X_n = x_n) = Pr(X_{n+1} = x | X_n = x_n).$$

Figure 4.2 shows an MDP or POMDP development over three *epochs*. The system may have been in

state $s_{t-1,1}$ or state $s_{t-1,2}$ at time $t-1$. It then progresses to state s_t at time t and finally transitions to either state $s_{t+1,1}$, $s_{t+1,2}$ or $s_{t+1,3}$. The Markov property takes action at time t, marked by the dashed line, by guaranteeing that the transition probabilities for the state transition between times t and $t+1$ are not affected by whether the system was in state $s_{t-1,1}$ or $s_{t-1,2}$ at time $t-1$; the transition probabilities depend only on the current state s_t.

This property has some effect on the extraction of MDPs and POMDPs from Simulink models. In fact, it produces a minimum epoch length for an extracted Markov Process. If the output or state of a Simulink model at time t depends on the model's state at time $t-\tau$, τ is the minimum length of an extracted Markov Process's epoch. This stems from the Markov Process's Markov Property. If the Markov Process is to correctly represent the dynamics of the system, the epoch must be longer than the maximum time lag inherit in the system.

This limitation must be considered when specifying the ratio between simulation time steps and decision epochs touched upon in section 4.1.10. In order to correctly transfer the system dynamics of the Simulink source model to the extracted Markov Process *the decision epoch must be at least as long as the model's maximum time lag*, otherwise the system's dependence on historical state would not be correctly represented.

A simple example is the following discrete-time system defined as a difference equation:

$$\begin{aligned} y[t] &= \sum_{k=1}^{5} y[t-k] + \sum_{k=2}^{4} x[t-k] \\ &= y[t-1] + y[t-2] + y[t-3] + y[t-4] + y[t-5] + x[t-2] + x[t-3] + x[t-4] \end{aligned}$$

This system depends on past output values as well as past input values, ie. is it not *memoryless*. The maximum time lag is 5 as the system output at time t depends on the output value at time $t-5$. Coming back to the Markov Property, this means that the minimum conversion ratio between an extracted Markov Process's decision epoch and a simulation time step is $1 = \frac{5\ time\ steps}{1\ decision\ epoch}$.

This limitation does not hinder the correct representation of a source system's dynamics as a Markov Process, but it does limit the time granularity of the resulting stochastic process. Not taking model time lags into account when defining the time-step/decision-epoch ratio will however lead to an incorrect transformation and must thus be kept in mind when defining extraction parameters.

4.1.12 Example 1: Inputs and Outputs

The purpose of this section is to provide an example that will present the simulation, observation and extraction process, especially in terms of simulation inputs and simulation outputs, in a more concrete way. This IO interface has already been introduced in section 4.1.2 and this example builds upon that introduction.

4.1.12.1 Model

The model used as an extraction base is depicted in figure 4.3. Two things are noticable. Firstly the model is deterministic. No random numbers are used, meaning that the model is referentially transparent, *given the same inputs and source state it will always respond in the same way*. Secondly, the model is source-state-independent. The lack of integrator, time-delay or other state dependent blocks means that the

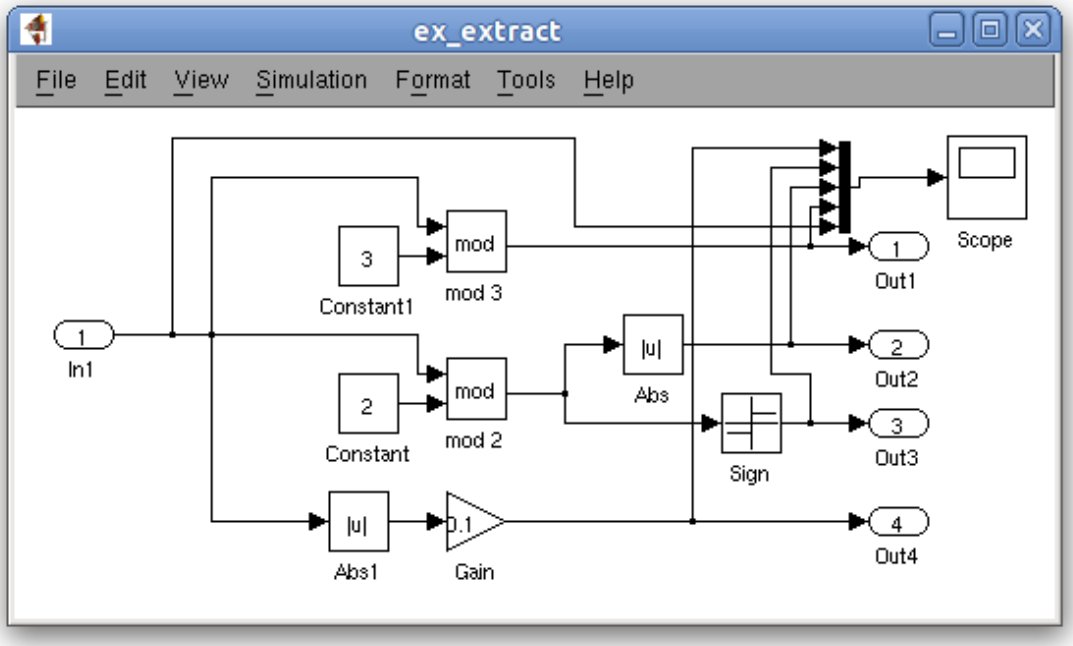

Figure 4.3: *Simulink Model for Example 1*

output of the model is entirely independent of the system's current state, it depends only on the actions. Nonetheless, this simple model can serve as an example that depicts the simulation process.

The model can also be defined formally,

$$
\begin{aligned}
y_1(t) &= u(t)\ mod\ 3 \\
y_2(t) &= |u(t)\ mod\ 2| \\
y_3(t) &= sgn(u(t)\ mod\ 2) \\
y_4(t) &= \frac{1}{10}|u(t)|
\end{aligned}
$$

where $u(t)$ is the input value at time t and $y_1(t)$, $y_2(t)$, $y_3(t)$, and $y_4(t)$ are the four outputs at time t, visible on the left and right side of figure 4.3, respectively.

4.1.12.2 Inputs and Actions

The actions space is quite small and taken as predefined (see the second example in the next section for a closer look at how the action space is built):

$$A = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{pmatrix} = \begin{pmatrix} u = -2 \\ u = -1 \\ u = 0 \\ u = 1 \\ u = 2 \end{pmatrix}$$

4.1.12.3 Outputs

The mapping of the POMDP actions to Simulink inputs and Simulink outputs to POMDP constructs, dicussed in section 4.1.2, is defined as:

- 'In 1': POMDP action value 1
- 'Out 1': POMDP state output 1
- 'Out 2': POMDP observation output 1
- 'Out 3': POMDP observation output 2
- 'Out 4': POMDP reward output 1

This means, that Markov Process states represent the value of the first Simulink output, that the observation is a combination of the second and third output and, finally, that the reward is obtained from the last Simulink output.

4.1.12.4 State Space

In order to keep this example simple, no output value boundaries have been defined. As will become obvious, these are not necessary for this example.

4.1.12.5 Simulations

After the action set has been defined, the inputs mapped to actions and the outputs mapped to either state, observation or reward outputs, the extraction can begin. Because of the deterministic nature of this model, each simulation with the same input will always lead to the same outputs. This is why, this example runs but a single simulation for each source-state/action tuple. In reality hundreds of such simulations would be completed in order to observe the model's stochastic behaviour (see section 4.1.9). The simulations presented here are also source state independent. As discussed in section 4.1.12.1 above, the model's output does not depend on the source state, meaning that the current state of the system does not influence the output.

Table 4.2 shows the inputs used for, and the results produced by all simulations that were run during the extraction.

Source State	Action	Inputs	Outputs (y_1,y_2,y_3,y_4)	State	Observation	Reward
n/a	a_1	$(u = -2)$	$(-2, 0, 0, 0.2)$	$(y_1 = -2)$	$(y_2 = 0, y_3 = 0)$	$(y_4 = 0.2)$
n/a	a_2	$(u = -1)$	$(-1, 1, -1, 0.1)$	$(y_1 = -1)$	$(y_2 = 1, y_3 = -1)$	$(y_4 = 0.1)$
n/a	a_3	$(u = 0)$	$(0, 0, 0, 0)$	$(y_1 = 0)$	$(y_2 = 0, y_3 = 0)$	$(y_4 = 0.0)$
n/a	a_4	$(u = 1)$	$(1, 1, 1, 0.1)$	$(y_1 = 1)$	$(y_2 = 1, y_3 = 1)$	$(y_4 = 0.1)$
n/a	a_5	$(u = 2)$	$(2, 0, 0, 0.2)$	$(y_1 = 2)$	$(y_2 = 0, y_3 = 0)$	$(y_4 = 0.2)$

Table 4.2: *Example 1: simulation results*

Source State	Action	Sink State	Observation	Reward
n/a	a_1	s_5	o_1	0.2
n/a	a_2	s_4	o_3	0.1
n/a	a_3	s_1	o_1	0.0
n/a	a_4	s_2	o_2	0.1
n/a	a_5	s_3	o_1	0.2

Table 4.3: *Example 1: interpreted simulation results*

4.1.12.6 Interpretation

Given the above simulation results, the state space can now be defined as,

$$S = \begin{pmatrix} s_1 = 0 \\ s_2 = 1 \\ s_3 = 2 \\ s_4 = -1 \\ s_5 = -2 \end{pmatrix},$$

and the discovered observation space as,

$$O = \begin{pmatrix} o_1 \\ o_2 \\ o_3 \end{pmatrix} = \begin{pmatrix} o_{1,1} = 0, o_{1,2} = 0 \\ o_{2,1} = 1, o_{2,2} = 1 \\ o_{3,1} = 1, o_{3,2} = -1 \end{pmatrix}.$$

Table 4.3 shows how, using these newly discovered spaces, the simulation results from table 4.2 can now be reinterpreted differently, based only on state space and observation space indices and on a scalar reward value.

4.1.12.7 Extraction

Using these interpreted results, the transition probability matrix of the sytem can be computed. One way of looking at the 3D transition probability tensor is to simply look at it as a set of 2D matrices indexed by the action. The Markov Decision Process can then be represented as a simple Markov Chain (section 3.2.1) for each action. The following matrix, is the transition probability of the extraced system for action a_4:

$$Pr(s'|s, a_4) = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

All rows are equal because the transition probabilities are *source state independent* and each row contains only a single non-zero value because the system's transitions are not probabilistic, but *deterministic*. The transition probability matrices for the remaining actions are available in appendix A.1.

The results in table 4.3 also allow the extraction of the condition observation probabilities. These are presented in the matrix below. *Note, that the observation probabilities can be represented in different ways. In the following matrix, they are presented as a probability of being in a certain state given a certain observation. This can obviously be turned around the other way. In the matrix below, the columns are indexed by state and the rows by observations, meaning that the probability of being in state s_3 if observation o_1 was observed is defined as the third element of the first row.*

$$O(s|o) = \begin{pmatrix} \frac{1}{3} & 0 & \frac{1}{3} & 0 & \frac{1}{3} \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

The reward matrix is extracted simply by storing the received reward, the source state, the sink state and the action. Querying for rewards is then simply a question of requesting the (i, j, k)-th element of the matrix where i is the source state index, j the action index and k the sink state index.

4.1.13 Example 2: Transition Probability Extraction

The aim of the following example is to anchor the previously touched upon abstract concepts in reality. All steps of the simplified transition probability extraction process are shortly explained to support the more theoretical descriptions of the previous two chapters.

4.1.13.1 Model

The extractor's source model in this example is a black-box SISO (single input, single output) model. The extractor has no knowledge of the underlying dynamics or the system's domain and codomain.

4.1.13.2 Extraction Configuration

The following configuration options have been chosen for this extraction:

- input: u
- input boundaries: $u_{min} = 1$, $u_{max} = 2$
- input granularity: N_u = 2;
- output: y
- output boundaries: $y_{min} = 0$, $y_{max} = 10$
- output granularity (discretization): N_y = 0
- time-step/decision-epoch conversion ratio: $10\ time\ steps\ =\ 1\ decision\ epoch$
- number of probabilistic simulations: $n_{psim} = 10$;

4.1.13.3 Action Space

Given the input boundaries and input granularity, the action space is produced as follows:

$$A = \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} u = 1 \\ u = 2 \end{pmatrix}$$

4.1.13.4 Simulations

This example does not contain the results of all simulations. Only a subset thereof is shown. Table 4.4 shows the system outputs observed for n_{psim} simulations from state s_7 using action a_2. Of the 10 simulations, three produced output values outside the permitted ranges, a single simulation produced an error and the other 6 simulation produced output values within the boundaries. The fifth column shows the discretized output values. Section 4.1.4 explains this process in more detail. Finally the last column shows the states these output values or results were mapped to.

4.1.13.5 Conditional Transition Probability Extraction

Using the simulation results from table 4.4, the conditional transition probabilities from source state s_7 given action a_2 can now be computed. The *simulation error state* is defined as s_1 and the *out-of-bounds state* as s_2, producing the following transition probabilities:

$$\begin{aligned} Pr(s'|a_2, s_7) &= (Pr(s_1|a_2, s_7), Pr(s_2|a_2, s_7), Pr(s_3|a_2, s_7), \cdots, Pr(s_{|S|}|a_2, s_7)) \\ &= (0.1, 0.3, 0.0, 0.0, 0.2, 0.1, 0.3, 0.0, 0.0, 0.0, \cdots, 0.0), \end{aligned}$$

where $|S|$ is the cardinality of the state space, ie. the total number of states.

Simulation Run	Source State	Action	Output Value	Discretized Output Value	Sink State
1	7	2	n/a	n/a	simulation-error
2	7	2	3.4	3	7
3	7	2	3.2	3	7
4	7	2	10.6	11	out-of-bounds
5	7	2	2.4	2	5
6	7	2	2.9	3	7
7	7	2	-1.2	-1	out-of-bounds
8	7	2	-1.3	-1	out-of-bounds
9	7	2	1.9	2	5
10	7	2	6.4	6	6

Table 4.4: *Example 1: simulation results*

4.1.13.6 Observation Probabilities and Rewards

The observation probabilities and the rewards are extracted in the exact same way, simply by observing specified reward and observation outputs. Because of the similarity of these extraction processes, they are not presented in more detail in this example.

4.1.13.7 Result

After running all required simulation from all *discovered* states the *extractor* will have produced a POMDP reflecting the dynamics of the source system.

4.2 Validation

In order to quantify the value of this *tranformational* approach, a validation tools is necessary. The extraction of Partially Observable Markov Decision Processes from Simulink model is simply a transformation between different dynamic system descriptions, a transformation from a rule based system description to a probabilistic state transitional system. Ideally such a transformation would produce a perfect representation of the source model. Unfortunately assumptions and simplifications (see sections 4.1.3, 4.1.4 and 4.1.11) may have a derogatory effect on the quality of the system representation. The *validator* aims to help identify the qualitative shortcomings of an extraction product. Using visual descriptive statistics tools such as box plots and simple correlation analysis, the validator provides a user of the extractor with means to verify the accuracy of the extracted dynamic system.

Chapter 6 presents an actual extraction result using the *validator* to compare the respones of the original system and the extracted system.

4.2.1 Approach

The *validator* measures qualitative difference between two dynamic system descriptions by comparing the responses of the two systems to idential stimulation. The approach of analysing the response of a dynamic system to well-defined stimulation comes from the field of control, as this method can often provide insight into the system's behaviour. Linear time-invariant system are, for example, completely defined by their impulse response, ie. the reponse to other input signals can be deduced from the impulse response. Common stimulation signals include the Dirac Delta function, the Heaviside step function, sine functions or white noise.

This approach is based on the assumption that similar responses to identical stimulation implies similar system descriptions. Given that the *extractor* simply transforms a model described in one way (Simulink) to a model described in another way (Markov Process) the response of these two systems should be similar given similar stimulation.

The difference between the responses of both systems may thus be an accurate measure of the qualitative deficit of the Markov Process description.

4.2.2 MDP/POMDP Simulator

In order to compare system reponses, the validator requires the ability to simulate Markov Processes. Given a fully defined MDP or POMDP and a decision array (containing an action for each decision epoch), a simulator must produce a response of the system as a historical series of states the process reached.

Therefore the validator includes an MDP simulator. Given an initial state and action set tuple the simulator samples as follows: at each epoch, a uniformly distributed number in the range $[0, 1]$ is taken, the state the system will transition to in the next epoch is then chosen by comparing the uniform random number to the transition probability's cumulative distribution. This sequential sampling and transitioning loop continues until all actions have been taken and the historical state transitions is returned as the simulation output.

4.2.3 Methodology

This section describes the methodology behind the validator. Because of the probabilistic nature of the systems studied in the context of Markov Processes, validation cannot rely on only individual simulations. In order to take into account the random differences between system responses, Monte Carlo simulations are required. Accurately comparing the responses of a Simulink model and an MDP model thus entails comparing response distributions instead of individual responses.

Box plots offer a way of representing distributions of values visually and are thus the validator's main output. A box plots takes a series of values and represents these values using a median mark, a box containing values between the 25th and 75th percentile, whiskers traditionally encompassing approximately 99.3% of the data (if normally distributed) and finally individual outliers far of the norm.

Figure 4.4 shows two example box plots, both produced with only one data series. The box plot on the left was produced using a range of only four values taken from the standard normal distribution and it is immediately obvious, that four values do not suffice for a clear representation of the normal distribution. Contrastingly, the box plot on the right was produced using one thousand random number taken from the

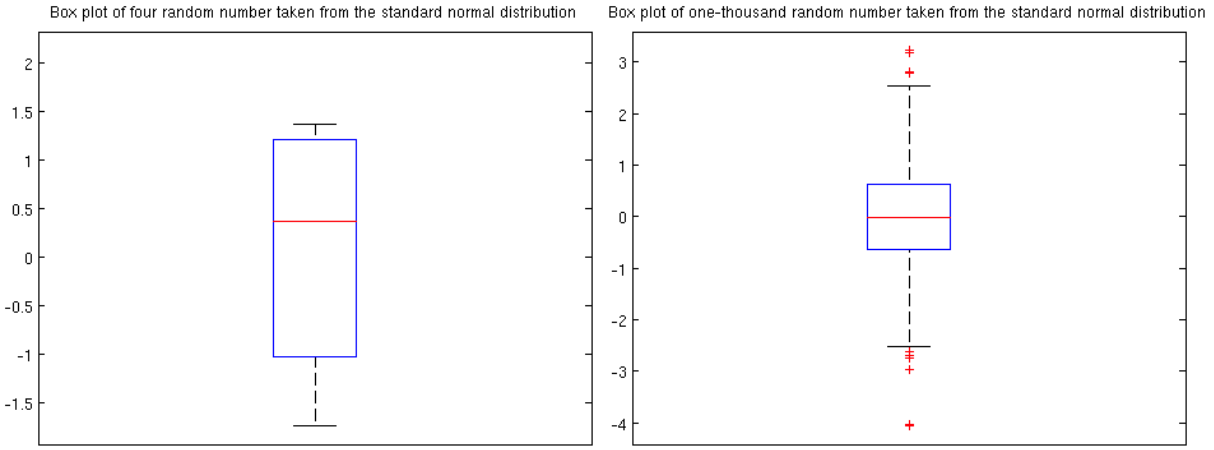

Figure 4.4: *Example box plots*

same distribution. The law of large number dictates that the median should be nearly exactly zero and the variance symmetric; this is clearly the case here. This short example shows that box plots are a good tool for the visual representation of probabilistically distributed data.

The validator produces two sets of two box plots. Using a series of values for *each* epoch, a box plot is created to show the distribution of those values at each epoch. Each set of box plots, shows distributions of results of Simulink simulations in the upper plot and distribution of results of the Markov Process simulation in the lower plot. The two sets differentiate in that the first produced set compares the system responses in the real number space, whilst the other set compares the system responses in the state space.

A simulink simulation produces values in the real number space, whilst a Markov Process simulation occurs in the state space. In order to compare these two results, the Markov Processes states must be mapped to real values, or the real values of the Simulink simulation must be mapped into the state space. The validator does both. After running multiple simulations of both the Simulink model and the Markov Decision Process, the validator converts the Simulink simulation output series to state transition series by searching the Markov Process's state space and finding the states that match the scalar values, and it converts the Markov Process simulation's state transition series into a series of real values simply by using the state definitions' rounded output value (see section 4.1.4). The validator now has both the simulation result as an array of consecutive values as well as an array of consecutive states. Using both these descriptions, it produces box plots representing the distribution of both simulations' outputs and states.

Beside the box plot, the validator also produces a plot showing the correlation of the simulation result distributions in state space. It does this by calculating, for each epoch, the correlation between the state distribution vectors of the Simulink model's system response and the Markov Process's response.

Finally the validator also produces two plots using the mean output value (real numbers space) and the mean state-index (state space) of both the Simulink and the Markov Process simulations at each epoch. Although this plot provides valuable information, it must be considered carefully, as it no longer takes into account the variances of the responses, merely their means.

4.2.4 Limitations

This validation approach does have certain limitations. Mainly it is based on the assumption that similar responses imply similar system dynamics. Although this can be said of linear time-invariant systems, it may not be the case for other types of models. This must be kept in mind when assessing the quality of an extracted model. A number of other limitations exist.

Firstly the validator can only be used with single-output models. The response difference is not computed in the continuous domain, but rather in state space, meaning that the outputs of simulations of the Simulink models are mapped into the Markov Process's state space and then compared. In order to assess the variance of the two systems' responses, the state space must then be ordered. Unfortunately ordering only supports one-dimensional values (n-dimensional spaces cannot easily be ordered).

Additionaly a comparison of a continuous model with a discrete model requires the discretization of the former, meaning that the quality of the transformed model can only be judged as far as the discretization permits. Rough discretization parameters may hide some of the source model's underlying dynamics, yet still not be correctly recognized by this validation tool. The quality of its assessment is limited by the roughness of the transformation's discretization.

Finally the comparison is also limited by the conversion ratio between simulation time steps and Markov Process epochs (see section 4.1.10). Because this ratio may be greater (but never less) than 1, the dynamics of the system between epochs cannot be compared to the response of the Markov Process. This means that a comparison is only possible at every epoch, even though the original model may show significant non-linear behaviour between these time steps. The validator can thus not provide a qualitative assessment of the differences between epochs.

Implementation

The following chapter describes the implementation of both the *extractor* and the *validator*, building upon the methodology introduced in chapter 4. Because of the limited scope of this text and detailed comments in the source code and the configuration files, the following sections will only provide a high-level overview of the implementation, going into more detailed discussion only for a number of rather interesting aspects.

5.1 Extractor

The extractor is implemented in the MATLAB programming language using an object-oriented approach. The choice of the MATLAB programming language stems from the fact that it is strongly integrated with Simulink, introduced in section 3.3. The entire code base encompasses approximately 2000 lines of code including comments. The following sections introduce both the main extraction class, the simulation class, miscellaneous utility classes and functions and the format of an extraction configuration script. Finally a difficulty with Simulink's random number generators is discussed and the solution presented.

5.1.1 Extration Class

The *extraction* logic, ie. the transformation of simulation results into conditional state transition probabilities, is implemented as a single class, namely 'POMDPExtractor'. The 'POMDPExtractor' calls upon a simulation object and other helper functions. It exports a relatively simple interface, containing both a simple single-call extraction function,

`extract()`,

as well as the following individual functions:

- `create_error_state()`,
- `create_out_of_range_state()`,
- `prepare_action_space()`,
- `initial_discovery()`,
- `sim_loop()`,
- `fill_error_tps()`,
- `fill_out_of_range_tps()`.

As mentioned in the introduction to this chapter, only certain, more interesting, parts of the implementation will be covered, as the rest is explained in the source code comments. In the following the class data structures, the initial discovery and the main simulation loop are shortly discussed.

5.1.1.1 Data Structures

The 'POMDPExtractor' class contains, amongst other things, data structures for the conditional transition probability matrix, the conditional observation probability matrix, the reward model, the state space and the observation space.

The transition probability matrix is implemented as a three-dimensional tensor, indexed by positive integers, accessed with the following call:

```
TP(source-state-index,action-index,sink-state-index).
```

The reward model is an identical three-dimensional tensor data structure, whereas the observation probabilities are stored in a standard matrix, also indexed by positive integers, the state-index as well as the observation-index.

The state space and the observation space are implemented as structure arrays of the following schema,

```
state_space = struct( ...
            'outputs', {}, ...
            'init_state', {}, ...
            'been_source', {} ...
            );

observation_space = struct( ...
            'outputs', {} ...
            );
```

where 'outputs' are arrays of quantitized values representing model outputs (eg. temperature, pressure, flux, etc), 'init_state' contains the Simulink internal simulation state object and 'been_source' is a boolean flag used to query the state space for states not yet used as simulation source states. The observation space struct contains a single field, an array of discretized model output values.

After every simulation, a 'save_state' function receives the simulation model's output values, the Simulink internal system state object and the current state space and, after discretizing the output values, checks whether this state has already been discovered, if yes, disgarding the data and if not, adding the newly discovered data to the Markov Process's state space. The newly added state will then be used as a source state in a later simulation loop. A similar function takes a simulation's observation outputs and updates the observation space accordingly.

5.1.1.2 Initial Discovery

The initial discovery, touched upon in section 4.1.8, produces the extractor's initial state space. Before a single simulation is run, the extractor has no knowledge of the state space that will be incrementally built as the extraction progresses. In order to provide the main simulation loop with source states to begin simulating with, the initial discovery function simulates the source model for every action in the action space without specifying an initial system state, letting Simulink fall back to its default intial system state. The results of these simulations are not used to extract the transition probabilities, the observation probabilities or the reward model, they are merely used to build an initial state space to be used in the main simulation loop.

5.1.1.3 Main Simulation Loop

Following the initial discovery, the extractor enters the main simulation loop. Every iteration of this loop runs all required simulations from a single source state. The source state is chosen by querying the state space data structure for states that have not yet been used as source states for extraction simulations.

Within this loop the entire action space is again looped through and probabilistic simulations (see section 4.1.9) are run for each source-state/action pair, producing a result similar to the one given in the extraction example in section 4.1.13. Using these simulation results, a single row in the three-dimensional state transition probability matrix is computed, as well as entries in the observation and the reward model updated.

The main simulation loop ends when all states in the state space have been used as source states, ie. when the entire conditional transition probability matrix, the entire conditional observation probability matrix and the entire reward matrix have been computed.

5.1.2 Simulation Class

The 'Simulation' class provides the extractor with a simple interface for simulating Simulink models. Amongst other helper functions it exports the following 'parallel_simulations' function:

```
function [par_sim_out] = parallel_simulations( ...
                             model_name, ...
                             a_t, ...
                             a_u, ...
                             num_steps, ...
                             num_runs, ...
                             catch_exception, ...
                             a_init_state)
```

This function provides the backbone to all of the extractor's simulations. It provides a simplified interface to MATLAB's parallel computing system.

Given a model name, a time step array, inputs for each simulation at each time step, initial states and a number of other configuration arguments, the 'Simulation' object will solve scoping problems, handle exceptions and finally run parallel simulations providing each simulation with the correct inputs and initial state. The simulation results are returned as a cell array containing simulation outputs and simulation state or in the case of simulation errors, the error description (for debugging). A more detailed discussion about running parallel Simulink simulations in MATLAB is available in appendix A.2.

MATLAB provides a decoupled interface for running parallel simulations, meaning that job control adjusts dynamically to the number of available cores. A helpful consequence of this is that extractions will run on both single-core and multi-core machines without modification, the latter case, however, providing much faster results.

5.1.3 Extraction Configuration

A discussed in the previous chapters and sections the extractor requires a certain level of configuration, such as the number of inputs, discretization parameters, etc. Figure 5.1 provides an example of a

```
mp = POMDPExtractor();

mp.model_name = 'rand_mod';
mp.input_ranges = [[0,1];[0,1]];
mp.output_ranges = [0,2];

mp.number_of_inputs = 2;
mp.rand_input = 1;

mp.rand_input_type = 'normal';
mp.rand_input_config = struct( ...
                            'mean', 0, ...
                            'variance', 0.1 ...
                            );


mp.number_of_state_outputs = 1;
mp.number_of_observation_outputs = 1;

mp.epoch_time_steps = 10;

mp.action_space_granularity = 10;
mp.state_space_granularity = [-1];
mp.observation_space_granularity = [-1];

mp.extract_rewards = 1;
mp.is_deterministic_model = 0;

mp.probabilistic_extraction_num_runs = 500;

mp = mp.extract();
```

Figure 5.1: *Example extraction configuration*

complete configuration and extraction-calling script.

5.1.4 Model Randomness

Simulink randomness blocks produce pseudo-random numbers using seed values. The advantage of seed values is that they can be used to reproduce identical pseudo-random number sequences multiple times. The disadvantage of this approach is that an algorithm based on observing the random behavior of a model by simulating this model multiple times, will not notice any random behavior unless the seed values are changed for every simulation.

Although Simulink offers a relatively simple way of updating seed values, this approach does not work when simulating model with a specific inital state (saved from previous simulations). Simulations using past system state objects seem to ignore newly set seed values, opting instead for the seed value saved in the past simulation's system state object.

In order to overcome this problem, the 'POMDPExtractor' offers its own random number generator. This approach guarantees, that new random values will be available for each simulation. These random number are available as an input to the model, effectively replacing Simulink's random number blocks. The custom random number generator can be configured in the extraction parameters (uniform generator or normal generator, mean and variance values, etc). An example configuration can be seen above in section 5.1.3.

This approach does however have a drawback. The random number generator produces values for each time step, whereas a variable-step solver may require values between time steps. The consequence of this is that, if so required, the randomly generated values are interpolated between time-steps. This must be considered, as it may have adverse effects on the simulation quality.

5.2 Validator

The validator is implemented as a single function with the following signature:

```
function compare(model_name, ...
                 mdp, ...
                 mdp_init_state, ...
                 mdp_actions, ...
                 num_sims, ...
                 time_steps_per_epoch)
```

It takes the name of a pre-configured Simulink model, a Markov Decision Process, an index for the Markov Process's initial state, a vector containing action indices for each epoch, the number of required Monte Carlo simulations and the epoch to time step conversion ratio (see section 4.1.10), as arguments and finally produces the comparison metrics discussed in section 4.2.3. The following few paragraphs describe the validator's implementation in a bit more detail.

Given these parameters the validator first simulates the pre-configured Simulink model the required number of times, storing the output values for later analysis. The model must be pre-configured in the sense that its inputs do not come from the validator, but must rather be implemented as Simulink blocks within the model itself. This approach greatly simplifies the validator's implementation and because of Simulink's large library of signals, all the standard stimulation functions, such as unit steps, impulses and sine functions are easily available. In addition to this, simulation parameters such as start-/end-time must be configured to match the simulation Markov Processes simulation (see section 4.1.10).

Secondly the validator simulates the Markov Process by sampling random numbers and simulating the state transitions for a certain amount of time. The chosen actions are given as a parameter and must match the input signal used by the Simulink model. This may limit the use of complex input signals such a sine signals unless the Markov Process's action space is detailed enough to represent such a complex input signal.

Following the simulations of both the Simulink model and the Markov Process, the 'compare' function converts the Simulink model simulations' results to the Markov processes state space and converts the Markov Process' simulation results to continuous values by replacing the states by the discretized output values they represent (ie. the rounded values, see section 4.1.4 for more details). The results of all the simulations are then available both in the state space as well as in the continuous number space (actual output values).

Finally the comparison function produces the plots described in section 4.2.3, using MATLAB's built-in statistical functions (box plot, correlation coefficients, etc). An example of the produced output can be see in the results discussion in chapter 6.

Evaluation

The aim of this chapter is to critically analyse the usefulness and accuracy of the developed extractor. Using a Simulink model, a Markov Decision Process is extracted and then the responses of both systems to two different stimulation signals are compared. This chapter serves as an evaluation of the approach developed for this thesis.

6.1 Setup

In the following, the evaluation experiment setup is described. The test model, the stimulation signals and the model's Simulink implementation are introduced.

6.1.1 Test Model

The model used for this result analysis is a third-order Butterworth filter, chosen mainly because of it's interesting overshoot properties. The filter is modeled using the transfer function [Per91]

$$H(s) = \frac{1}{s^3 + 2s^2 + 2s + 1}.$$

Section 6.1.3 presents the Simulink implementation of this filter.

6.1.2 Stimulation Signals

For each experiment a different signal was used. Firstly a constant input signal and secondly a scaled step signal. These two signals $u_1(t)$ and $u_2(t)$ are defined as:

$$u_1(t) = 0.6667,$$
$$u_2(t) = \begin{cases} 0.6667, & t < 10 \\ 4, & t \geq 10 \end{cases}.$$

Figures 6.1 and 6.2 show the standard response of the filter to these inputs, as well as the input signals themselves. Note that even though $u_1(t)$ is defined here as a constant signal, it is in fact a step signal, with step time $t = 0$ and step value 0.6667. This effect occurs because there is no way to let the Simulink model start with a past signal value. Nonetheless it will be referred to here as the constant signal, because of the relatively small difference between the initial state and the step.

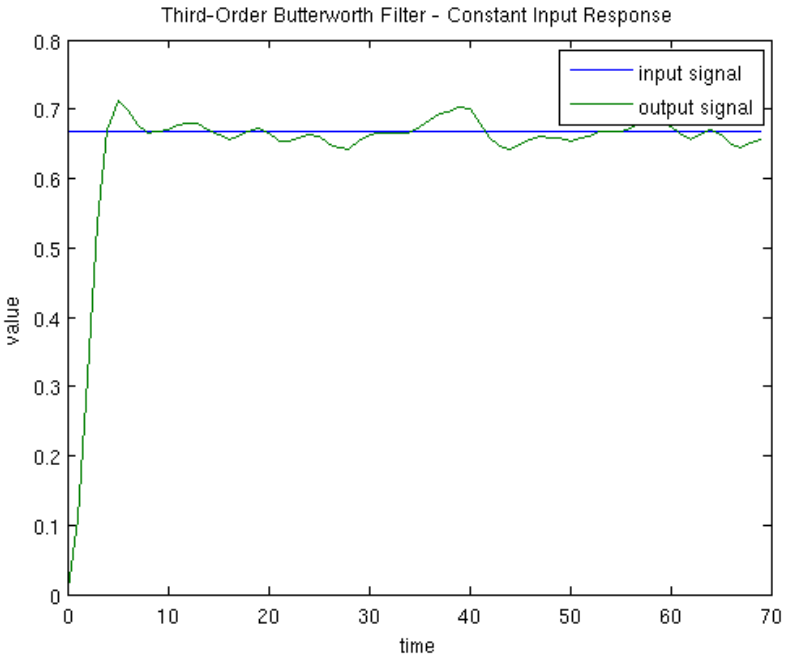

Figure 6.1: *Response of Butterworth-Filter system to constant input*

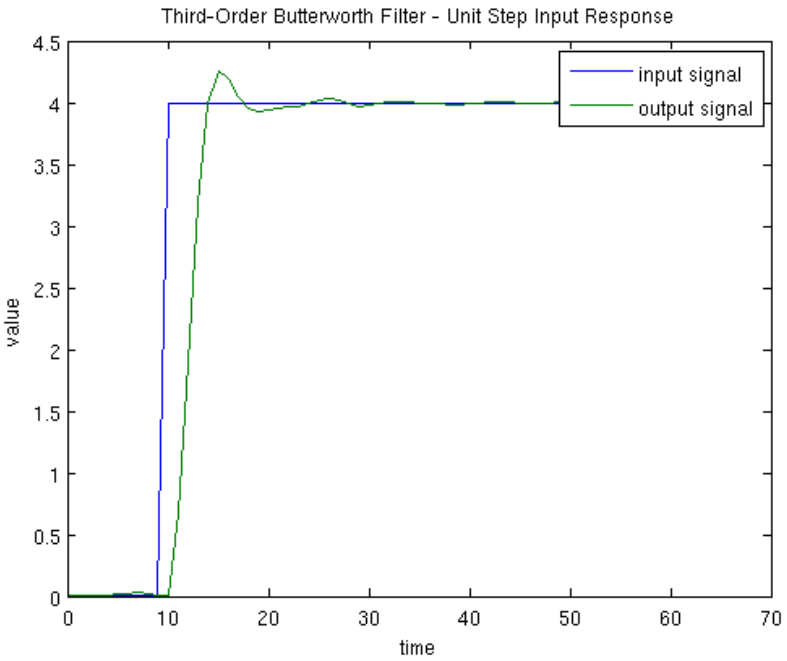

Figure 6.2: *Response of Butterworth-Filter system to a scaled step input (step at t=10)*

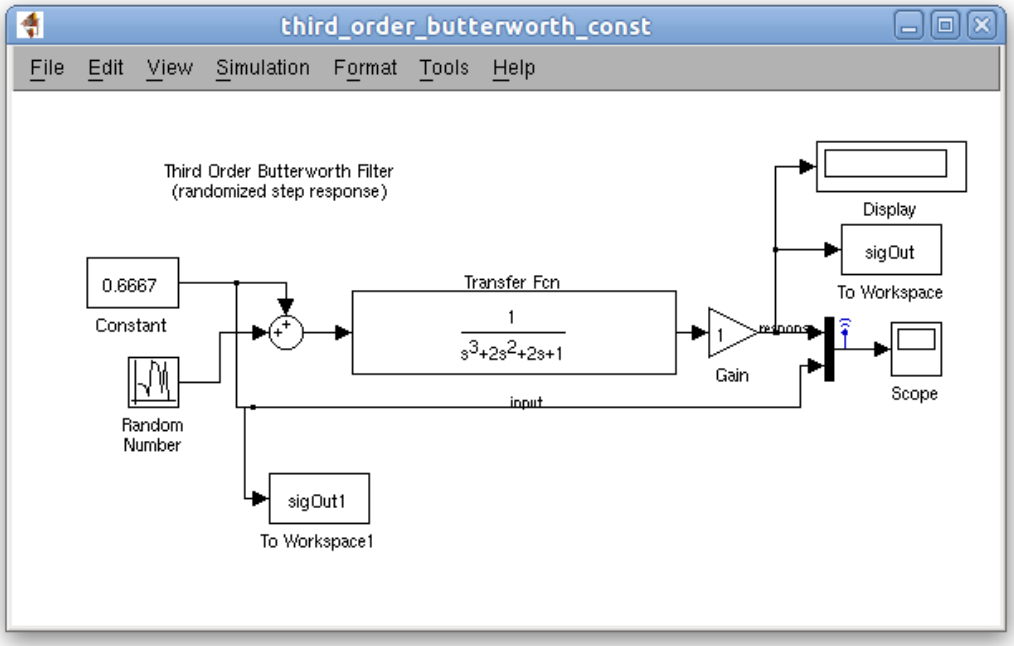

Figure 6.3: *Implementation of a Butterworth-Filter in Simulink with a constant input*

6.1.3 Simulink Models

The filter was implemented three different times as a Simulink model, once for each input signal and once for the extraction. Figures 6.3 and 6.4 show the model for input signals $u_1(t)$ and $u_2(t)$, respectively. Figure 6.5 shows the model as it was prepared for the extraction (with root level inputs and outputs).

6.2 Result

In order to evaluate the usefulness of the extraction approach presented in this thesis, the test model was transformed into a Markov Decision Process. As described in section 5.1.3, a number of choices must be made. The configuration of this extraction is relatively simple because of the model's simplicity (single input, single output). The following parameters defined the extraction:

- input: u, $u_{min} = 0$, $u_{max} = 6$
- input granularity: $N_u = 10$
- random input: normally distributed with mean $\mu = 0$ and variance $\sigma^2 = 0.001$
- output boundaries: $y_{min} = 0$, $y_{max} = 10$
- output discretization: $n_1 = -2$
- probabilistic simulation, number of runs: n_{psim} = 20

The extraction took approximately 40 hours on a dual-core HP workstation with 2 GB of RAM. 634 states were discovered during the extraction, meaning that the transition probability matrix is of dimen-

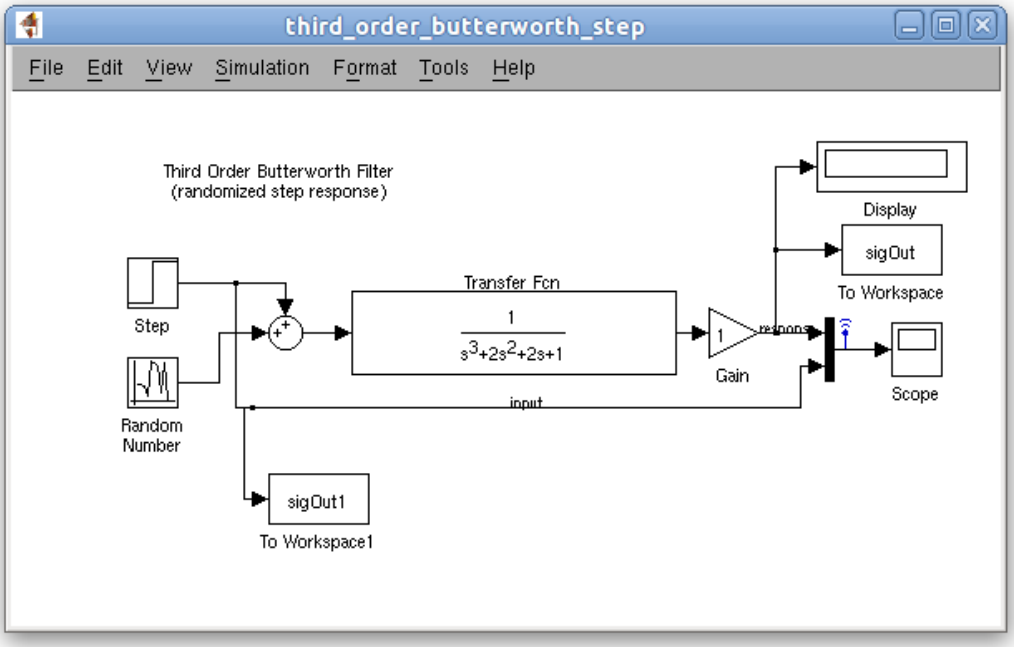

Figure 6.4: *Implementation of a Butterworth-Filter in Simulink with a scaled step input (step at t=10)*

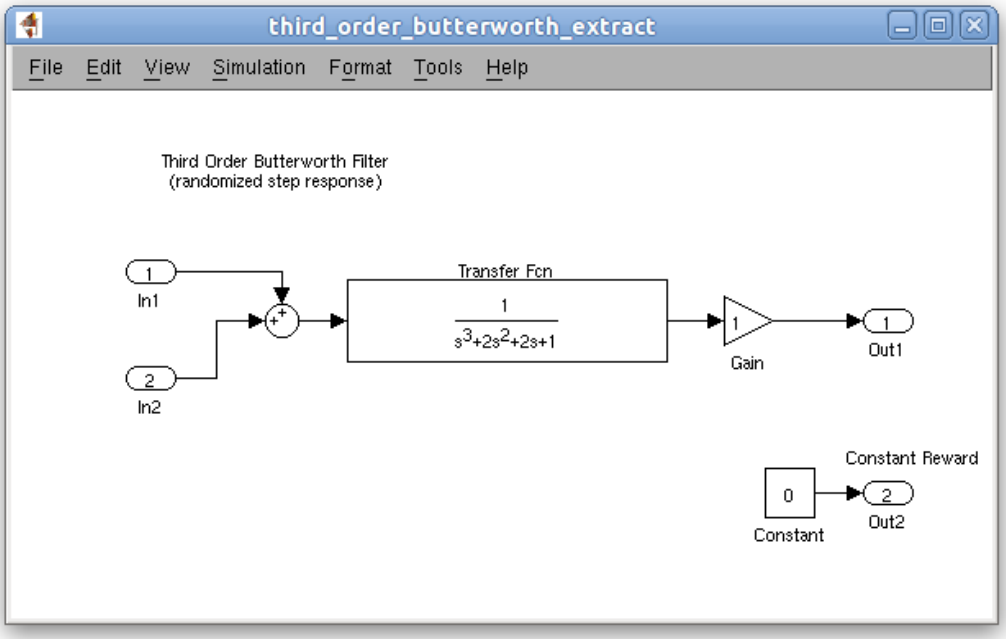

Figure 6.5: *Butterworth-Filter Simulink model prepared for MDP extraction*

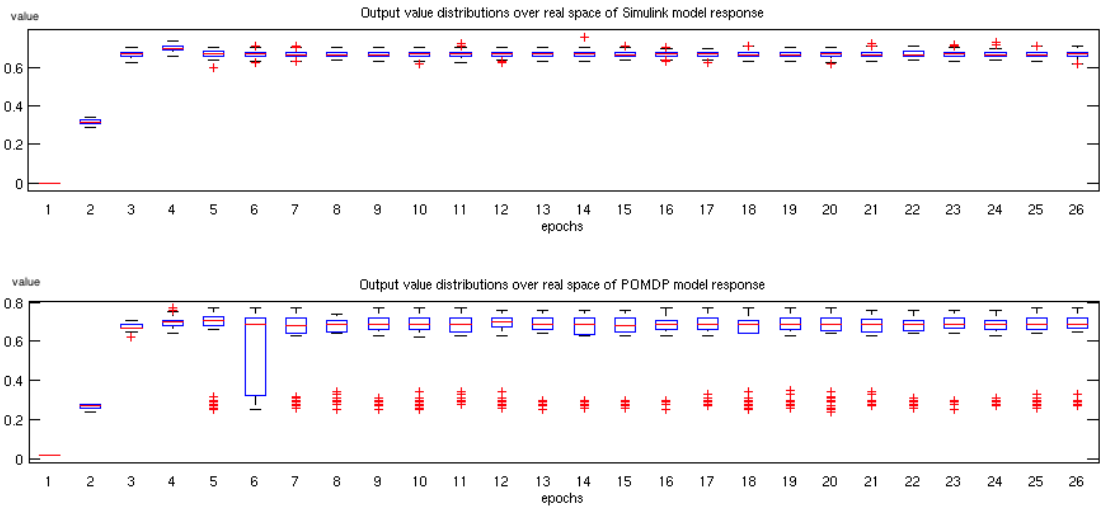

Figure 6.6: *Box plots of output value distributions of Simulink and MDP responses to signal* $u_1(t)$

sion $636 \times 636 \times 10$ (the two extra states are the *out-of-bound state* and the *error state* discussed in sections 4.1.5 and 4.1.6). The next section presents the extractor's evaluation.

6.3 Evaluation

Two validation comparisons were made, comparing the response of the Simulink model and the response of the Markov Decision Process to both the stimulation signal $u_1(t)$ and $u_2(t)$. The 'compare' function ran 100 simulation of both the pre-configured Simulink models and the Markov Decision Process. The resulting plots and a short analysis thereof can be found the next two sections.

6.3.1 Response to Constant Signal

In this section, the response differences between the Simulink and the MDP model will be discussed.

Figures 6.6 and 6.7 show the response distributions in both the real number space and the state space. Note that, as described in section 4.2.3, the conversion of the Simulink model's scalar output values to the Markov Process's state space provides an analytical base for evaluating the effect of the discretization (see section 4.1.4) on the extracted system's behaviour. In this case it is immediately obvious, that the discretization does not have a strong influence on the accuracy of the response, as the Simulink responses look almost exactly similar in both the real space and the state space and the MDP's responses also look equally distributed in both spaces. These box plots do however show that the MDP's responses are distributed with a much higher variance than those of the Simulink model. Additionaly two suprising patterns can be observed. Firstly the MDP system behaves differently at epoch six, where the response variance is much higher than for the the other epochs, and secondly, the MDP's responses have a number of ouliers that remain at much lower values, both in real and in state space.

The correlation plot in figure 6.8 also offers interesting insights. The average correlation between state distributions averages at around 0.4. This value is quite low and probably caused by the combination of

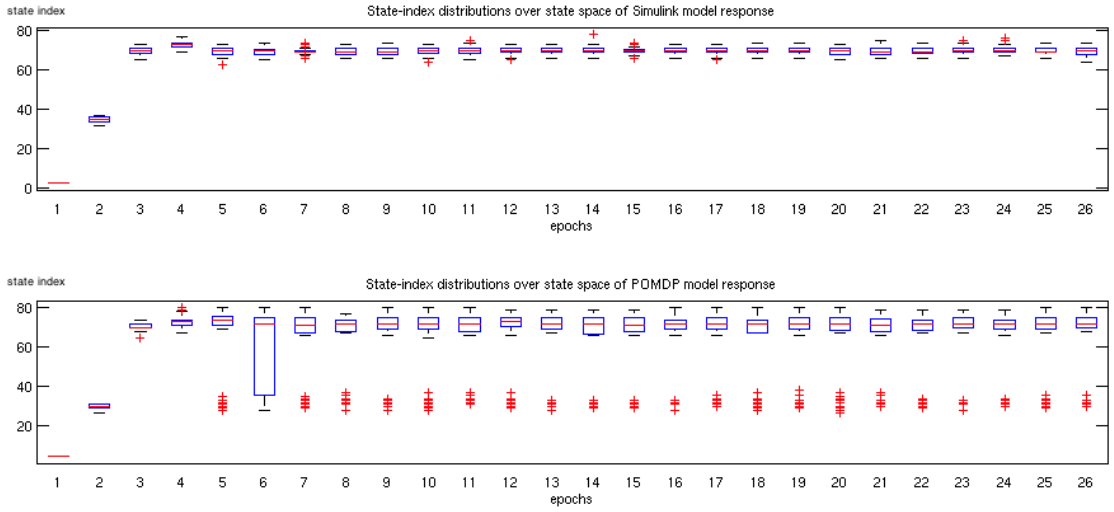

Figure 6.7: *Box plots of state-index distributions of Simulink and MDP responses to signal* $u_1(t)$

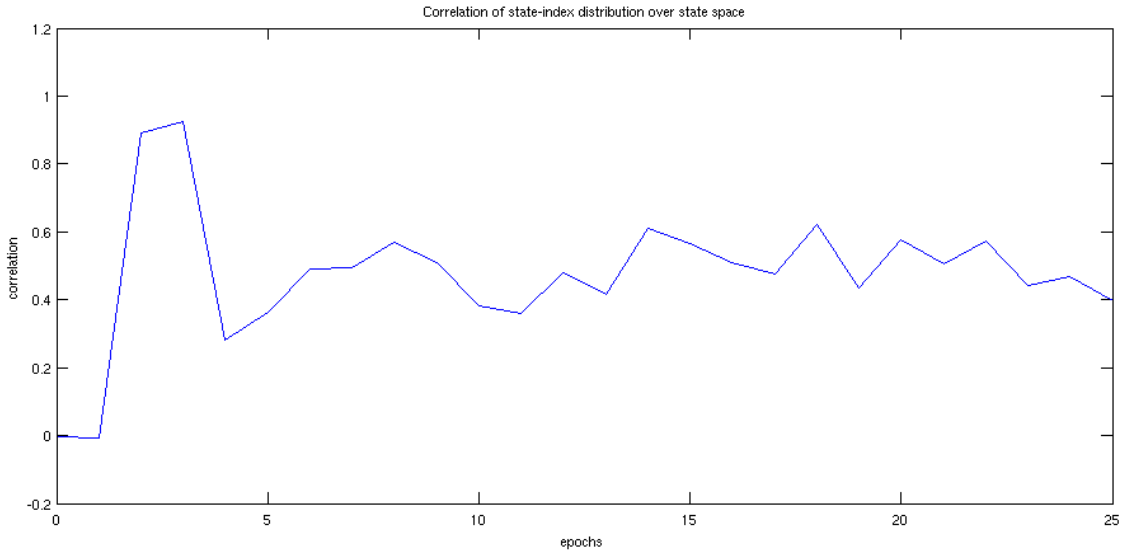

Figure 6.8: *Plot of state-index distributions correlation of the Simulink and MDP responses to signal* $u_1(t)$

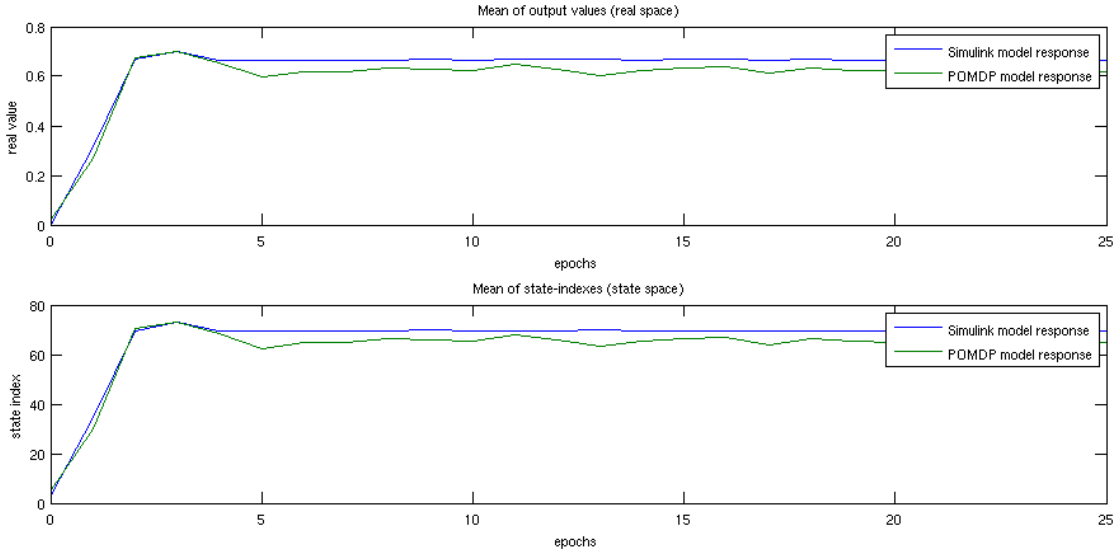

Figure 6.9: *Plot of the real output value mean of Simulink and MDP responses to signal* $u_1(t)$

a large number of states and the rather large variance discussed in the previous paragraph. Interestingly the correlation jumps to much higher values at epochs two and three. This seems to indicate that whilst the correlation is quite low at steady state, it increases when the system radically changes.

Finally, figure 6.9 shows the means of all responses in both state and real space. The plots show a strong correlation of the mean values. This is valuable information because it shows that even though the MDP's responses vary greatly, they do in fact, in average, produce a very similar response to the original model's. The different outputs, both in real and in state space, at steady state should also be noted and will be discussed in chapter 7.

6.3.2 Response to Step Signal

The responses to the step input provide even more insight into the difference between the Simulink and the MDP system description. The next few paragraphs will again discuss the visible differences.

Similar as for the constant input signal, the discretization does not have a visible effect on the system responses. This can be seen by the lack of visible differences between the box plots of the Simulink responses in real and in state space as well as by looking at the differences between the response distributions of the MDP in the two spaces. In this case, the MDP also responds with a much stronger variance, indicating a less accurate system description. The responses do show, that both models react similarily and at the same time to the stimulation signal. Interestingly the MDP system settles at a slightly higher output value before and after the initial step. Interestingly these responses show an effect that was not noticable with the responses to the constant stimulation. The box plots show that the distribution variance only increases after the disturbance. The MDP's response variance before the step (at epoch 10) is suprisingly small. This may indicate a propagation effect, whereby the variance increases with every disturbance, as the resulting uncertainty propagates to later epochs.

Figure 6.12 shows the correlation between the responses' state space distributions for each epoch. The plot shows that the distributions do not correlate at all, except for epoch 13. The extremely low correlation

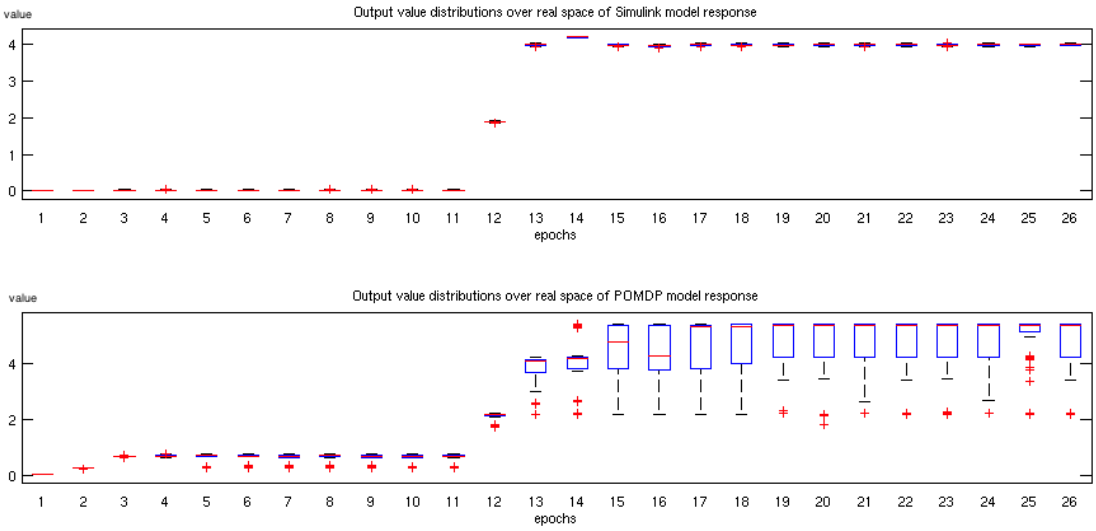

Figure 6.10: *Box plots of output value distributions of Simulink and MDP responses to signal* $u_2(t)$

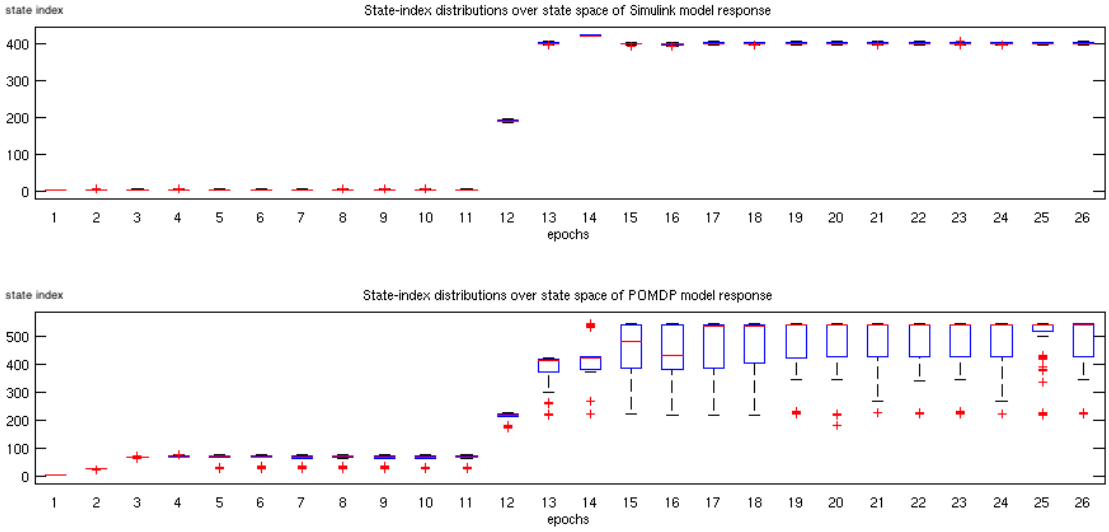

Figure 6.11: *Box plots of state-index distributions of Simulink and MDP responses to signal* $u_2(t)$

will be discussed in more detail in chapter 7, but is likely caused by initial state differences between the models. Similarily to the correlation in the previous section, the correlation also spikes just after the disturbance, indicating that, contrary to steady state, the models' reactions correlate strongly when the system is disturbed.

The plot of real and state output means in figure 6.13 show two interesting effects. Firstly the models seems to diverge even before the disturbance and secondly the systems diverge again after the disturbance. Yet, during the disturbance, the systems react similarily. Although this is discussed in more detail

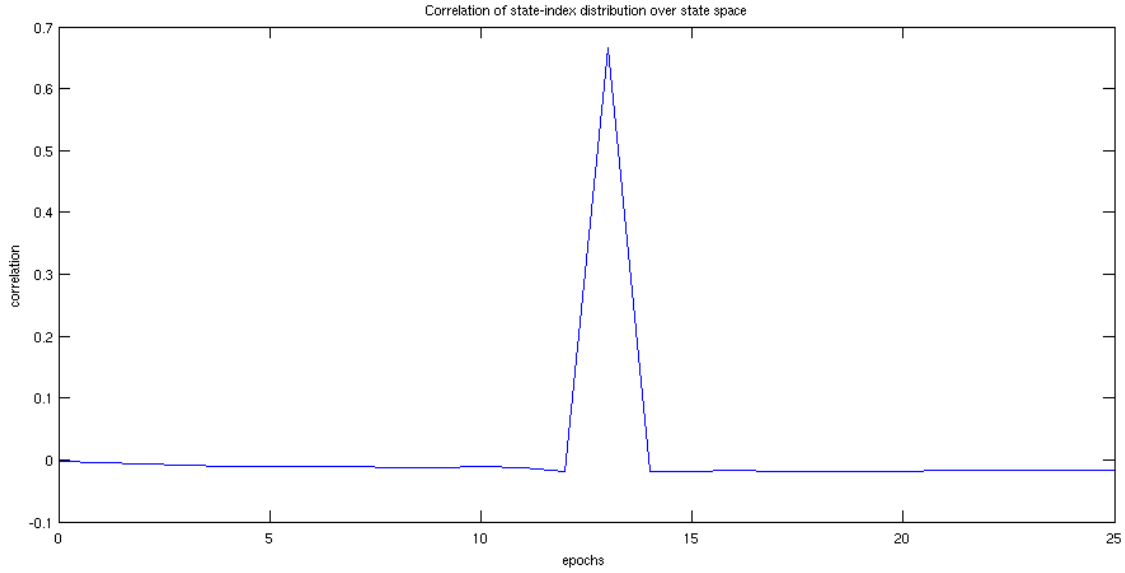

Figure 6.12: *Plot of state-index distributions correlation of the Simulink and MDP responses to signal* $u_2(t)$

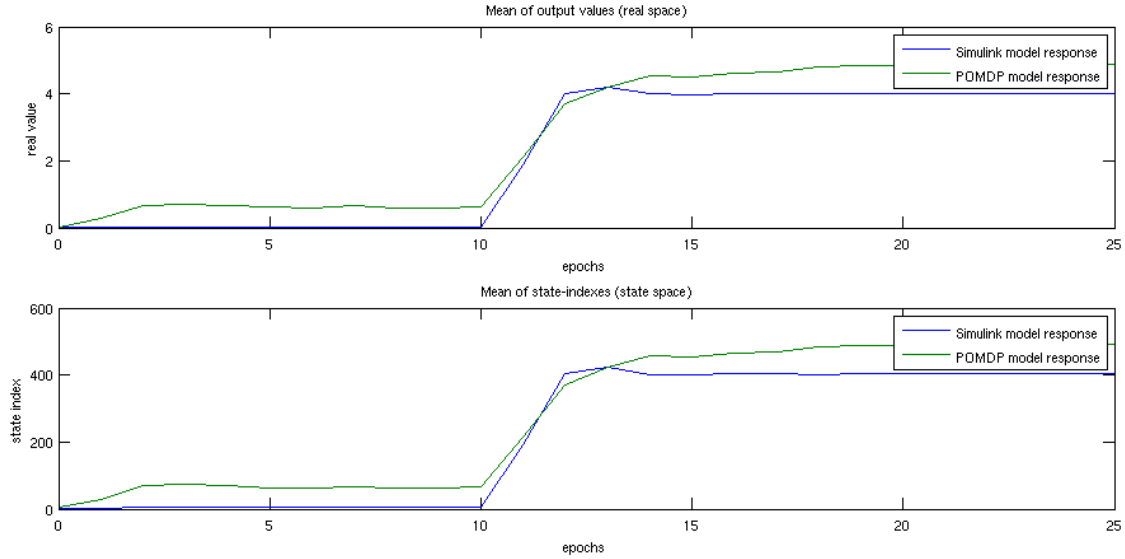

Figure 6.13: *Plot of the real output value mean of Simulink and MDP responses to signal* $u_2(t)$

in chapter 7, this effect may, again, be caused by different inital states and the propagation of uncertainty after the disturbance.

Discussion

Using the results presented in chapter 6 this chapter aims to discuss the positives and negatives of the approach described in the thesis.

As described in chapter 2, the extraction of Markov Processes from Simulink models offers a number of advantages for the optimization of complex dynamic systems. The ability to automatically transfer system dynamics from a Simulink representation to that of a Markov Process would allow scientists and engineers to develop non-myopic controllers that take uncertainty into account, without the difficulty of having to implement these models by manually producing multi-dimensional condition transition probability, conditional observation probability and reward matrices. The test model used in chapter 6, a third-order Butterworth filter, provides a good example. Even though this system is relatively simple, with only a single input and a single output, a manual implementation of its dynamics as a Markov Decision Process would have entailed defining a $636 \times 636 \times 10$ tensor by hand. This example underscores the usefulness of an automatic transformation. A successful transformation means that engineers can concentrate on modelling and leave the error-prone process of manually converting models to transition, observation and reward probabilities to an automated extraction system, whilst still taking advantages of the Markov Process's favourable optimization features.

For a prototype implementation, the extraction worked relatively well for the Butterworth filter. The responses to both disturbance signals more or less compare to the original model's reponses (see figures 6.9 and 6.9). Nonetheless, deficits are visible. The much greater response variance of the MDP's response, suggests, that the number of probabilistic extracton simulations (see sections 4.1.9 and 6.2) should have been increased in order to correctly identify the real random behaviour of the model. Such a requirement does however lead to higher computational extraction cost. The easiest approach to this problem is to take advantage of the extractor's parallelization features to extract on machines with a high number of CPU-cores. Further study is required to identify the number of samplings necessary to truly *extract* the system's random behaviour.

The results do not show an accuracy loss due to discretization. It is, in fact impossible to notice response differences caused by this loss of information (compare figures 6.6 and 6.7 or figures 6.10 and 6.11). It must however be noted, that this accuracy will decrease if a rougher state space granularity is chosen. The tradeoffs between decreasing the size of the state space and loss of accuracy must be thoroughly researched before further conclusions can be drawn.

In the evaluation experiment, the assumption of biconditionality between *system outputs* and *system state* (see section 4.1.3) seems to have led to only marginal differences. The systems react to the disturbances at approximately the same time with the same speed. Nonetheless this is but a single comparison. The effects of this assumption *must* be better understood before this extraction approach is used for actual optimization purposes. A number of factors may influence the system behavior because of this simplification. The most dangerous possibility, is that the dynamics of the extracted system can be influenced by the order with which states are discovered. Currently the action set is produced in an increasing order, meaning, that the input values generally increase over time. A consequence of this may be that most states are discovered when the system is reacting to increasing input disturbance, which would lead to a generally less stable system. Simple experiments, such as extractions of the same model using inverted

action sets, should offer insights into this problem. Additionally the effect of this biconditionality assumption may not be as visible in this experiment as it would be with more complex models. Further research is necessary to better understand the consequences of this assumption.

Further valuable insight can be gained from looking at the response distributions' correlation at each epoch (see figures 6.8 and 6.12). Two interesting observations can be made. Firstly, whilst the average correlation for the first input signal (figure 6.8) averages around 0.4, the correlation of the response distributions to the second disturbance signal is almost zero. The next paragraph offers an opinion as to what may be the cause of this. Secondly, an interesting effect it noticable. The correlation of the responses jumps to much higher values when the system is reacting to changing inputs (around 0.9 in epochs 2 and 3 for the constant disturbance signal and around 0.7 in epoch 13 for the step input signal). This may point to an interesting effect, whereby more dynamic behavior is extracted more accurately than stable behavior. Experiments with extractions using different time step to epoch conversion rates (section 4.1.10) may offer evidence to support or disprove this hypothesis.

A last observation can be made by looking at figures 6.9 and 6.13. Although the similarity of the responses' means underscores the value of this approach, the strange diverging behavior in steady-state must be viewed critically. Especially in the case of the second disturbance signal (scaled step), the system's mean output value diverges from the source model's both before *and* after the disturbance. The difference before the disturbance may be caused by a different initial simulation state and the difference after the disturbance is probably caused by an uncertainty propagation. The latter effect should be studied in more detail by analysing steady-state responses of Markov Processes. The former problem concerning different initial simulation steps is hard to solve. The divergence before the disturbance is, in all likelihood, a consequence of the biconditionality assumption, discussed in section 4.1.3. This divergence probably also affected the responses' distribution correlation, analysed in the previous paragraph. The steady-state behavior of a system is one of its defining properties and often the only really important one (short term dynamics can often be ignored because of most physical systems' inherit inertia), meaning that this effect must be studied in more depth and its causes identified and corrected.

From a different point of view, this example shows, that the extraction of Markov Decision Processes from Simulink models does have a large computational cost. The extraction of this SISO model required more than 40 hours. Taking into account that the extraction of a more accurate model requires an order-of-magintude increase of the number of probabilistic simulations, the cost becomes even higher. A usable transformation of the Butterworth filter system may, indeed, take up to 400 hours. More complex models, with a much larger action space (action set cardinality in the hundreds or thousands), will lead to even higher time and performance costs. Parallelization approaches are the only solution here, and should thus be researched in more detail.

Finally it should be considered, that even though a successful POMDP extraction may be possible, explicitly solving POMDPs is generally inctractable [Lit96]. Consequentially, the usefulness of an extraction may be limited by the computational cost of solving, even approximately, POMDPs with large state spaces.

It can be concluded, that the extraction approach presented in this thesis is promising. Comparisons of the original and the extracted system show that the underlying dynamic behavior has been transfered into the new system description. Nonetheless, the consequences of the assumptions and simplifications made by this approach (see chapter 4) must the researched in more depth, especially in the field of modelling where small mistakes may have disastrous consequences. Additionaly further study is required to assess the usabilify and usefulness of the produced Markov Processes.

Conclusion and Outlook

Building on the evaluation discussed in the previous two chapters, this last chapter aims to provide both a conclusion to this work and an outlook for the future.

The promising evaluation results presented in chapter 6 and discussed in chapter 7 show that this approach has merits, but, as with any complex undertaking, the different effects that different assumptions, model intricacies and exraction configurations have on the quality of the extracted model must be studied in depth before this approach can used in an industrial setting. Interesting approach vectors include comparing Markov Processes extracted with different extraction parameter combinations, inverting the action set before extractions begin, sampling Simulink system state objects instead of repeatedly using a single one, analysing the effect that the number of probabilistic simulations has on the quality of the represented randomness, comparing the effect model complexity has on the transformation quality, etc.

Another possible continuation of this work is the extraction of dynamic systems described not through Simulink models, but other representational forms. Any system description can be used to extract a Markov Decision Process if it fulfills the same requirements as Simulink model, namely the possibility of defining inputs and observing outputs. The outputs must also provide a meaningful description of the system's internal state (section 4.1.3) and the MDP's decision epoch must be longer than the system's internal time lags (section 4.1.11). If these conditions are met, the transformation of the system into a Markov Decision Process should be possible. Experiments using this approach to extract dynamic systems from other system representation would, without doubt, offer interesting results.

Irrespective of the continued development of this approach, it has shown, that a Monte-Carlo approach to extracting dynamic system behaviour has merit.

Appendix

A.1 Example Transition Probabilities

The following matrices represent the entire extracted transition probabilities of the example from section 4.1.12. The 3D tensor is presented as 2D matrices indexed by the action.

$$Pr(s'|s,a_1) = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$Pr(s'|s,a_2) = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

$$Pr(s'|s,a_3) = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$Pr(s'|s,a_4) = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

$$Pr(s'|s, a_5) = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

A.2 Parallelization of Simulink Simulations

MATLAB offers many parallelization tools ranging from the simple parfor loop to the highly-advanced *Parallel Computing Toolbox*. Because of the large number of simulations required for a single POMDP extraction, the idea of taking advantage of these parallelization tools seems obvious. Unfortunately running parallel Simulink simulations is not as easy as it could be. The following two sections discuss the basic parallelization used by the extractor and the main challenge when running parallel simulations: scope. The entire source code of the parallel simulation function used in the main extraction loop (see sections 5.1.2 and 5.1.1.3) is available in appendix A.3.

A.2.1 Basics

The 'parallel_simulations' function parallelizes simulations using MATLAB's 'parfor' loop. As can be seen in appendix A.3, the function actually uses two 'parfor' loops. These two loops are necessary because of scope issues discussed in the next section. Below is the source code of the second parallel for loop (without comments and blank lines):

```
1  parfor n=1:num_runs
2      assignin('base', 'gbl_t', a_t{n});
3      assignin('base', 'gbl_u', a_u{n});
4      lconf = conf;
5      if catch_exception
6          try
7              sim_out = sim(model_name, lconf);
8              outputs = [sim_out.get('save_out').signals.values];
9              result_state = sim_out.get('save_final');
10             par_sim_out{n} = {outputs, result_state};
11         catch err
12             par_sim_out{n} = {getReport(err, 'extended')};
13         end
14     else
15         sim_out = sim(model_name, lconf);
16         outputs = [sim_out.get('save_out').signals.values];
17         result_state = sim_out.get('save_final');
18         par_sim_out{n} = {outputs, result_state};
19     end
20 end
```

A single simulation is run with the 'sim(model_name),lconf)' function call. It takes the name of a Simulink model in the path and a configuration struct and returns different outputs depending on the configuration. Here simulations are either started within or outside of a try-catch block depending on whether or not the caller wants simulation exceptions to bubble up.

In order to take advantage of the 'parfor' loop, MATLAB requires the user to manually start a pool of *MATLAB workers*. This can be done with the 'matlabpool' command. The following command starts a local pool with 8 workers (one on each core):

```
1 matlabpool local 8;
```

Working parallel code can even be used, without modifications, on a distributed MATLAB cluster, but this topic is far outside the scope of this thesis. More informaion is available on the MathWorks homepage.

A.2.2 Scope Problems

Two categories of scope errors were encountered during the implementation of the extractor. The first category has to do with the variables used in 'parfor' loops and the second category contains problems concerning Simulink models' scope.

This first category of problems is the reason why the 'parallel_simulations' function contains two 'parfor' loops. The two loops handle two different cases. In the first case the simulation does not require an initial state and in the second case it does require an initial state. The scope problems occurs if one tries to implement both cases in a single parfor loop. Before the parfor loop is executed it checks which variables *may* be necessary in the loop. These variable are then passed to the worker process. The problem can be seen in the following example code:

```
1 parfor n=1:10
2     if 1
3         sim_without_initstate();
4     else
5         sim_with_initstate(initstate);
6     end
7 end
```

Even though the second case (with an initial state) will never occur, the parfor loop still requires that the 'initstate' variable be in scope before the loop is executed. This simple problem is the cause of the double loops in the 'parallel_simulations' code.

The second category of difficulties has to do with Simulink scoping issues. Because model's can rely on values defined in the general workspace they will often not immediately work inside a parfor loop. The problem is that the worker does not have access to MATLAB's main workspace. This means that any required variables, libraries and/or submodels must be explicitely passed to the worker process. Line 119 in A.3 shows the first solution to this problem. If a model reads inputs via the workspace (using 'from workspace' or root input blocks) these variable must be individually assigned to the worker's own workspace. This is what the call to 'assignin' does on line 119. It effectively passes the value of variable 'a_t{n}' unto the worker and defines it as 'gbl_t' in the worker's workspace. The Simulink model now has access to the variable in it's worker's workspace.

This approach only works if few variable are required by the model. Unfortunately many larger models require dozens if not hundreds of workspace variables. Such models often rely on *driver* scripts that must be exectued before simulations to load all libraries, submodules and variables into the workspace. The solution in this case, is to run the driver script once, save the entire workspace to a 'mat' file and then instruct the model to look for it's variable inside this 'mat' file instead of the workspace. This approach works well within parallel loops.

A.3 Source Code: Parallel Simulation Function

```
function [par_sim_out] = ...
    parallel_simulations( ...
        model_name, ...
        a_t, ...
        a_u, ...
        num_steps, ...
        num_runs, ...
        catch_exception, ...
        a_init_state)

    par_sim_out = cell([0]);

    load_system(model_name);


    % config parameters
    conf = struct();
    conf.SaveOutput = 'on';
    conf.SaveFormat = 'StructureWithTime';
    conf.OutputSaveName = 'save_out';

    conf.SaveFinalState = 'on';
    conf.SaveCompleteFinalSimState = 'on';
    conf.FinalStateName = 'save_final';

    using_init_state = (nargin == 7);

    conf.LoadExternalInput = 'on';
    if using_init_state
        conf.ExternalInput = 'gbl_input_data';
    else
        conf.ExternalInput = '[gbl_t, gbl_u]';
    end


    % Set initial state if it is passed to
    % the function. The stop-time also depends
    % on whether or not an initial state is used.
    % If an initial state is used the stop-time
    % must be adjusted because we no longer
    % start simulations at t=0, but rather at the
    % time the initial state save saved at.
    if using_init_state
        conf.LoadInitialState = 'on';
        conf.InitialState = 'gbl_init_state';
    else
        % Must create dummy var here, because
        % the loop is created and vars assigned
        % before it knows that it won't ever need
        % this variable.
        conf.StopTime = mat2str(num_steps);
        conf.LoadInitialState = 'off';
    end

    % Create rand seeds before run

    % TODO: fix these double loops. The problem is
    % that is you put the assignin for the init state
```

```
59      % in an if checking whether or not we are using
60      % the init state, there will be a matrix size
61      % exceeded error, even when no using init states.
62      % I guess it has to do with pre-loop checks for
63      % the parfor loop (even though the variable wont)
64      % be used.
65      if using_init_state
66
67          parfor n=1:num_runs
68
69              load_system(model_name);
70
71              signal_values = a_u{n};
72              number_of_signals = size(a_u{n},2);
73              number_of_values = size(a_u{n},1);
74
75              input_data = struct();
76              start_time = a_init_state.snapshotTime;
77              end_time   = start_time + num_steps;
78
79              % Input data struct time values
80              input_data.time = (start_time:1:(end_time-1))';
81
82              % Input data signals values
83              for n_signal=1:number_of_signals
84                  input_data.signals(n_signal).dimensions = 1;
85                  input_data.signals(n_signal).values = signal_values(:,n_signal);
86              end
87
88              assignin('base', 'gbl_input_data', input_data);
89
90              lconf = conf;
91
92              assignin('base', 'gbl_init_state', ...
93                      a_init_state);
94
95              lconf.StopTime = mat2str( ...
96                  a_init_state.snapshotTime + num_steps);
97
98              % simulate model
99              if catch_exception
100                 try
101                     sim_out = sim(model_name, lconf);
102                     outputs = [sim_out.get('save_out').signals.values];
103                     result_state = sim_out.get('save_final');
104
105                     par_sim_out{n} = {outputs, result_state};
106                 catch err
107                     par_sim_out{n} = {getReport(err, 'extended')};
108                 end
109             else
110                 sim_out = sim(model_name, lconf);
111                 outputs = [sim_out.get('save_out').signals.values];
112                 result_state = sim_out.get('save_final');
113
114                 par_sim_out{n} = {outputs, result_state};
115             end
116         end
117     else
118         parfor n=1:num_runs
```

```
119             assignin('base', 'gbl_t', a_t{n});
120             assignin('base', 'gbl_u', a_u{n});
121 
122             lconf = conf;
123 
124             % simulate model
125             if catch_exception
126                 try
127                     sim_out = sim(model_name, lconf);
128                     outputs = [sim_out.get('save_out').signals.values];
129                     result_state = sim_out.get('save_final');
130 
131                     par_sim_out{n} = {outputs, result_state};
132                 catch err
133                     par_sim_out{n} = {getReport(err, 'extended')};
134                 end
135             else
136                 sim_out = sim(model_name, lconf);
137                 outputs = [sim_out.get('save_out').signals.values];
138                 result_state = sim_out.get('save_final');
139 
140                 par_sim_out{n} = {outputs, result_state};
141             end % exception catch
142         end
143     end
144 end
```

Bibliography

[BS] A.R. Ballesteros and M.T.J. Spaan. Towards a POMDP-based Intelligent Assistant for Power Plants.

[Han98] Eric A. Hansen. Solving POMDPs by Searching in Policy Space. In *UAI*, pages 211–219, 1998.

[III] Julius Orion Smith III. Simulated sine-wave analysis in matlab. Retrieved February 14, 2012, from JOS Website: `https://ccrma.stanford.edu/~jos/filters/Simulated_Sine_Wave_Analysis_Matlab.html`.

[Lit96] Michael Littman. *Algorithms for Sequential Decision Making*. PhD thesis, Brown University, Computer Science Dept., Providence, 1996.

[Per91] George A. Perdikaris. *Computer Controlled Systems: Theory and Application (Intelligent Systems, Control and Automation: Science and Engineering)*. Springer, Berlin, 1991.

[Put94] Matrin L. Putterman. *Markov Decision Processes, Discrete Stochastic Dynamic Programming*. Wiley-Interscience, Hoboken, New Jersey, 1994.

[RPPCD08] S. Ross, J. Pineau, S. Paquet, and B. Chaib-Draa. Online planning algorithms for POMDPs. *Journal of Artificial Intelligence Research*, 32(1):663–704, 2008.

[SB10] Olivier Sigaud and Olivier Buffet. *Markov Decision Processes in Artificial Intelligence*. John Wiley Sons, Hoboken, New Jersey, 2010.

[TM07] Paul A. Tipler and Gene Mosca. *Physics for Scientists and Engineers Extended Version*. W. H. Freeman, New York, NY, 2007.