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. Such extractions are useful for two main reasons. Firstly the modelling of complex non-linear dynamic systems is orders of magnitude easier in a graphical modelling tools such as Simulink. The manual representation of non-trivial dynamic models as Markov Decision Processes is extremely difficult, if not impossible. An extraction tool would thus allow easier modelling whilst still permitting the use of a Markov Decision Process based system representation. Secondly Markov Decision Processes have inherit advantages compared to 'rule-based' dynamic system respresentation when it comes to optimization. Optimization using Markov Decision Processes is non-myopic, is decoupled between system dynamics and the reward model and can be performed on simple and cheap vector processing units, making the use of 'smart' controllers possible even in cheap products. This thesis presents the theoretical background, approach and implementation of such an extraction tool.

Your task description pdf file here!

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.

Contents

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 motivation of this work was the development of a an extraction tool that can transform a dynamic probabilistic non-linear Simulink model into a Partially Observable Markov Decision Process.

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 described in chapter 5 and finally the results of this work are discussed in chapter 6, by studying the result of an extraction using the developed validation tool.

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; albeit drawbacks that do not outweigh their advantages. 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 incineration process must be managed optimally taking into account two different cost or 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 that takes into account the effect of current decisions on not only the next state, but on the future states of the plant; sometimes producing less may be producing more. Secondly, the decoupling of the reward model from the system dynamics, that Markov Decision Processes permit, allows for an on-line dynamic 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:

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

With the small-angles assumption

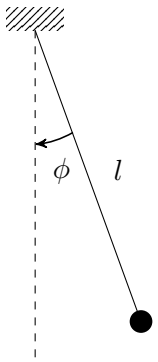

Figure 3.1: *1-dimensional mathematical pendulum*

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

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. 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

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

It is easy to see here that a discrete time system contains less information than a continuous time signal [MAYBE FIGURE HERE]. Nonetheless discrete time signals are prevalent because computers 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 $t \pm \tau \in T$. 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.

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 3.2.

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.

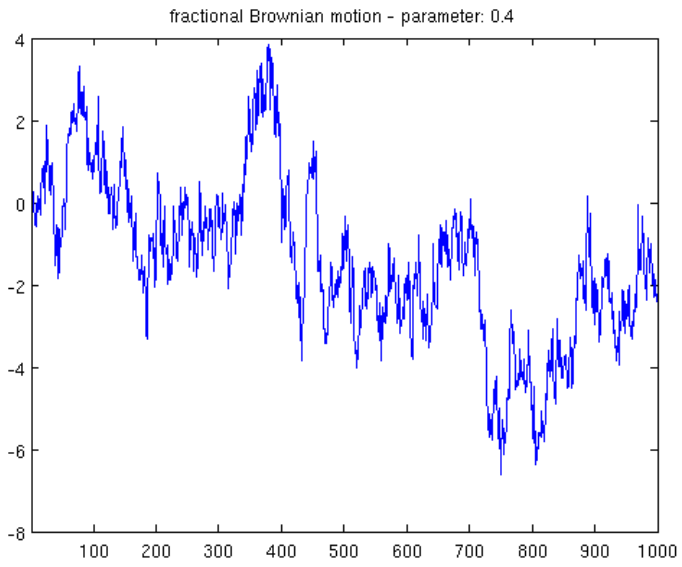

Figure 3.2: *Fractional Brownian motion with Hurst parameter of* 0.4

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:

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

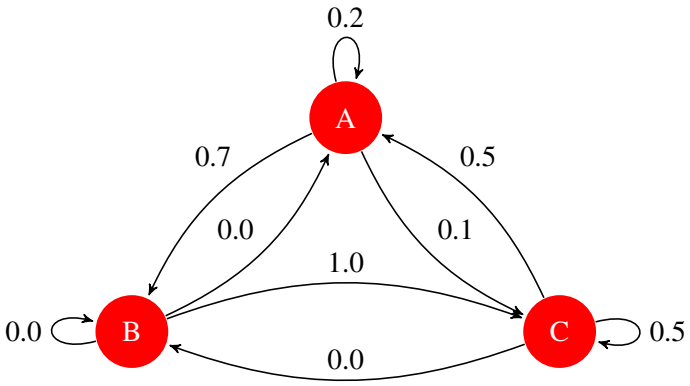

Figure 3.3: *Three-State Markov Chain*

Figure 3.3 shows an example of a three state Markov Chain. The transition probabilities matrix is as follows:

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

Looking at this transition probability matrix or Figure 3.3, 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.

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 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 can be 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.

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),

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

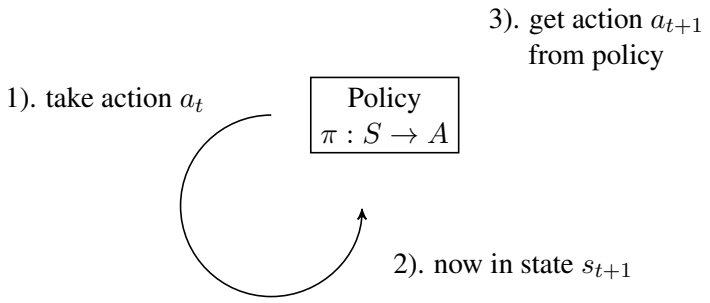

Figure 3.4: *MDP control loop*

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 [Put94] as: TODO. 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.

Given an MDP and a reward function, an optimal policy can be computed. The computation of such a policy 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]. The result of the policy computation is, as described above, a policy $\pi(s) : S \rightarrow A$ that maps elements of the set of states S to elements of the set of actions 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 interesting to look at the use of MDPs as control and optimization tools. Figure 3.4 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 $\pi : S \rightarrow A$ 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 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,

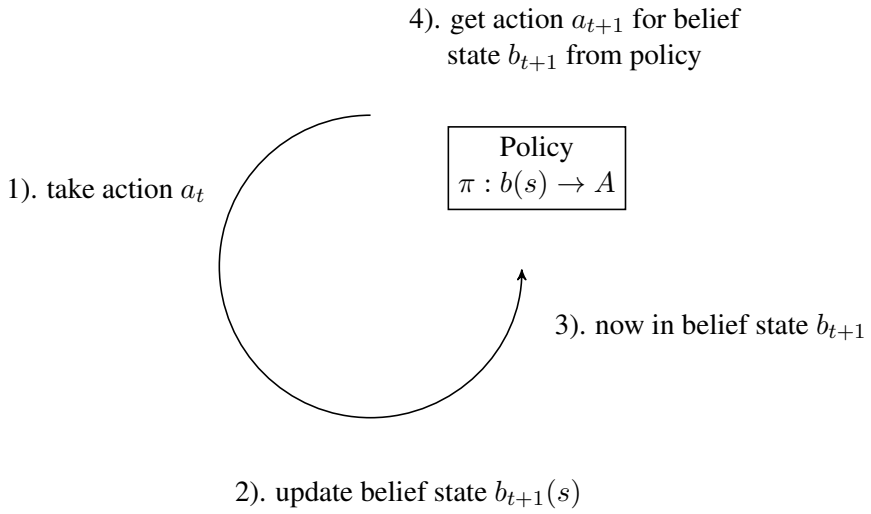

Figure 3.5: *POMDP control loop (offline planning)*

- 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.

Optimization using POMDPs is an order of magniture more complicated than with MDPs. 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 planner 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 3.5 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 $t+1$,

$$b_{t+1}(s) = \frac{1}{\sum_{s_{t+1} \in S} \Omega(o_t|s_{t+1}, a_t) \sum_{s_t \in S} T(s_{t+1}|s_t, a_t) \cdot b(s_t)} \cdot \Omega(o_t|s_{t+1}, a_t) \cdot \sum_{s \in S} T(s_{t+1}|s_t, a_t) \cdot b(s_t),$$

is then used to query the policy for the action a_{t+1} to take at time $t+1$. 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 simple *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][XXX][XXX].

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 covered in detail in [XXX] 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.

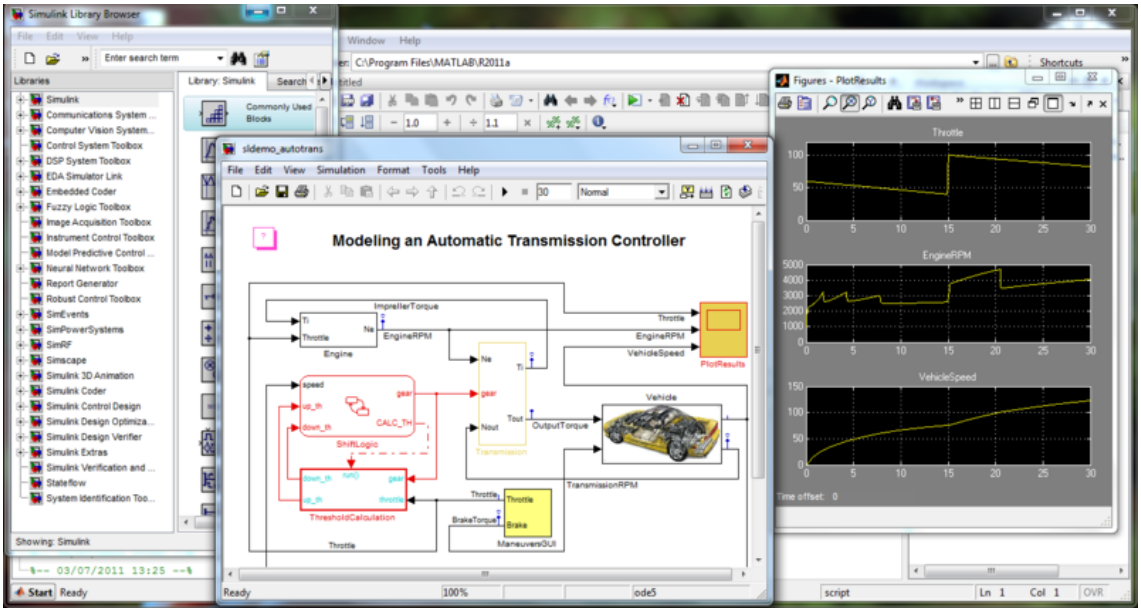

Figure 3.6: *Screenshot of Simulink*

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, et). 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 3.6 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 3.7 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 **??** 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.

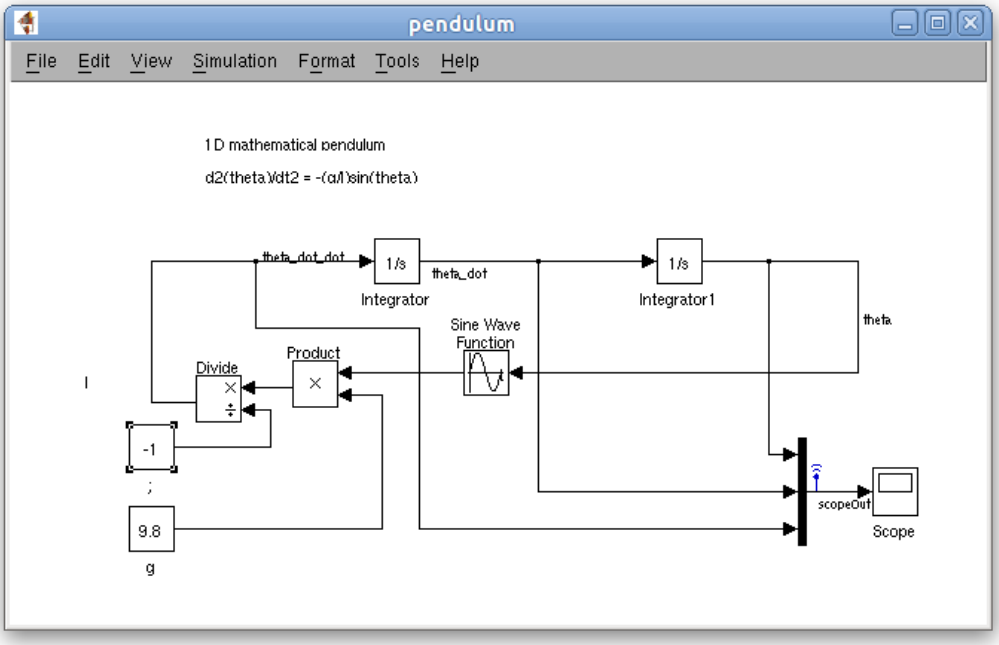

Figure 3.7: *1-dimensional mathematical pendulum model*

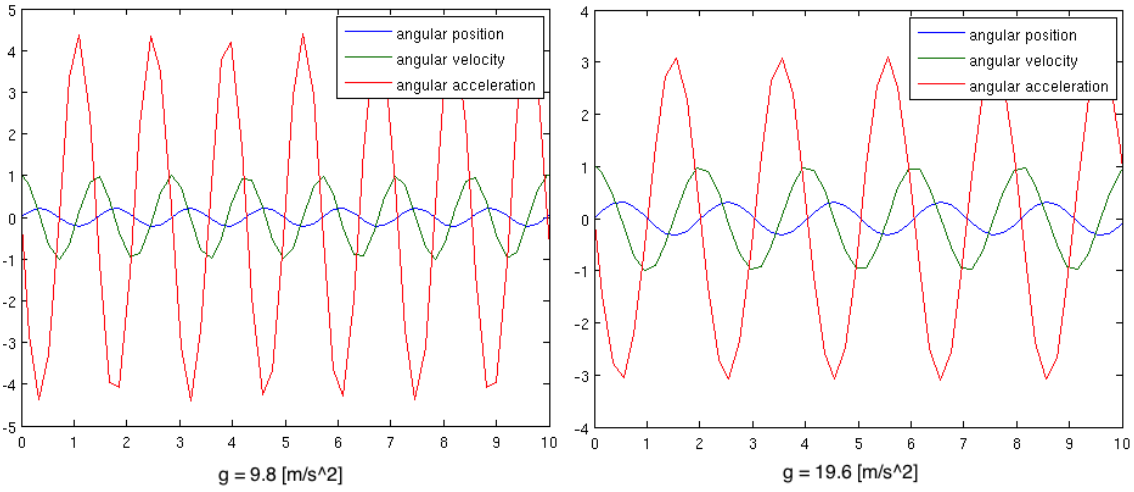

Figure 3.8: *1D mathematical pendulum response*

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 enough times and with enough 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.

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 4.1.8) 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 **??** presents this process in more detail.

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 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. Section [XXX] discusses the difficulty of extremely large state spaces in mode detail.

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 section [XXX].

4.1.3 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 between -1 and 1 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, 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 4.1.7.

4.1.4 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 be able to produce a *finite* POMDP, meaning 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 4.1.5, 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:

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

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

4.1.5 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

Simulation	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*

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 random 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 4.1.4, the error state has a transition probability of 1 not to change for any action. As such, random 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.

4.1.6 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:

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

with N_i being the number of different input values required between the maximum and minimum values of input x_i (see section XXXX discretization). 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. Implementation details are described in section [XXX].

4.1.7 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 model. A certain model may never reach some of the states in the 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.

4.1.8 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 section XXX, 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.

4.1.9 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

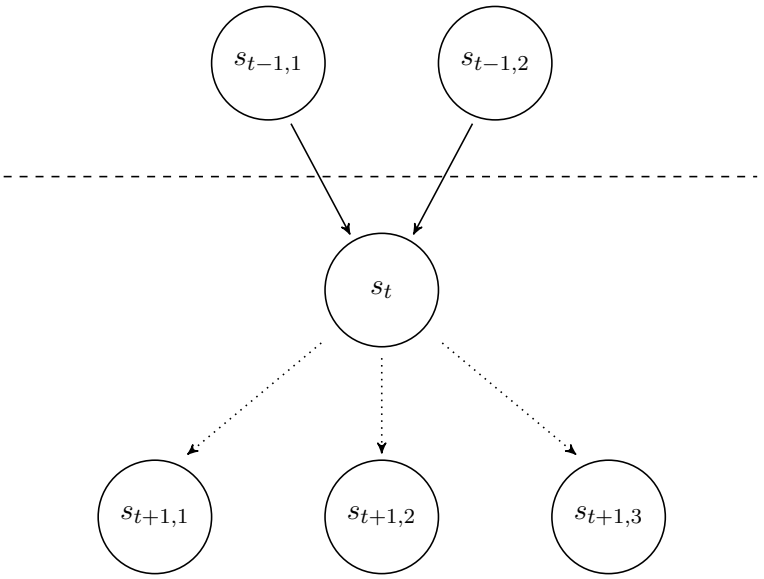

Figure 4.1: *Markov Property*

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 4.1.10 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 4.1.10). 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.

4.1.10 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 4.1 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 4.1.9. 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.

4.1.11 Example

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.

4.1.11.1 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.

4.1.11.2 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: $n_{psim} = 10$;

4.1.11.3 Action Space

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

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

4.1.11.4 Simulations

This example does not contain the results of all simulations. Only a subset thereof is shown. Table 4.2 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 4.1.3 explains this process in more detail. Finally the last column shows the states these output values or results were mapped to.

4.1.11.5 Conditional Transition Probability Extraction

Using the simulation results from table 4.2, 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	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 4.2: *Example extraction: simulation results*

4.1.11.6 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.

4.1.11.7 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 different assumptions and simplifications 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.

4.2.1 Approach

The *validator* measures qualitative difference between two dynamic system descriptions by comparing the responses of the two system to idential stimulation. The approach of analysing the response of a

dynamic system to well-defined stimulation comes from the field on 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 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 4.1.9). 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.

4.2.3 Results

The *validator* produces a number of outputs that can be used to judge the quality of the transformation. Firstly it produces box plots of the distribution of the systems' response in state space over time. These plots can be used to assess the variance between the responses of the two systems. Secondly it produces a plot of the correlation between response distributions in state space for each time step, and finally it produces a plot of the mean of all simulation results of each system in real space over time.

Section XXX uses example validation results to discuss the quality of the extraction of a Markov Process from a third-order Butterworth filter Simulink model.

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, MathWorks' commercial simulation tool, 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 Structure

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 4.1.7, 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 4.1.8) are run for each source-state/action pair, producing a result similar to the one given in the extraction example in section 4.1.11. 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).

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 Miscellaneous

The extraction code base also includes a number of other scripts and classes developed to facilitate both the development of the extractor as well as the extraction itself. A few examples include an advanced logging interface, a predicate based vector element counter and bash scripts that parse running extractions' log files and provide valuable statistics at run-time.

5.1.4 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. The following script provides an example of a complete configuration and extraction-calling script:

```
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();
```

5.1.5 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 5.1.4.

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. It takes a produced MDP or POMDP, a vector containing action indexes, a pre-configured Simulink model and a single configuration parameter, the number of simulations, 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 4.1.9).

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 compelx 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 (see section 4.1.3) output values they represent. The results of all the simulations is now available both in

state-space as well as in real-space.

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 section [XXX].

Results

This is the "Results" chapter!

6.1 Extractor

Talk about extractor output.

6.2 Validator

Talk about the output of the validator.

Conclusion

This is the "Conclusion" chapter!

Outlook

This is the "Outlook" chapter!

Appendix

A.1 Item 1

Previously, this and that has been done and so on.

Bibliography

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

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