Symbolic Controller Synthesis for the Swing-up of a Rotary Inverted Pendulum System

Bachelor Thesis

Scientific work to obtain the degree
B.Sc. Engineering Science

Munich School of Engineering
Technische Universität München

Supervised by Prof. Dr. Majid Zamani & Mr Mahmoud Khaled
Assistant Professorship of Hybrid Control Systems
TUM Department of Electrical and Computer Engineering

Submitted by Renyan Sun
Matriculation Number: 03677518
renyan.sun@tum.de

Filed on München, on 30th June 2018

Abstract

This Bachelor Thesis illustrates the construction of a controller for a rotary inverted pendulum using SCOTS, a software designed by Professorship of Hybrid Control Systems. The software can synthesize a controller automatically through feedback refinement technology. Although there already are mathematically well defined controllers for the rotary inverted pendulum, this thesis is aimed at an automatic construction of such a controller with the above-mentioned software. Two models including the motion model and the energy model are introduced in this thesis in both mathematical and programming forms.

Acknowledgement

Mr. Prof. Dr Majid Zamani, thank you for introducing me to your research group and allowing me to write a thesis at your Professorship. It's been a great honour to work with your group members and learn a lot not only in specific knowledge in the implementation of SCOTS, but also passion, dedication and self-discipline.

Mr. Mahmoud Khaled, my supervisor, I want to thank you for your great support. Despite my numerous questions, you have always been a patient and helpful mentor for me. Your guidance has been an important assistance in completing this thesis.

Siyuan Liu, thank you for your support along the way, especially in the last year. Your willingness to listen and your capacity for compassion have been a great comfort for me to get through all sorts of challenges throughout my senior year.

Guillermo, Emre and HaHyung, thank you for your friendship for the last three years. You also share a piece of success for the completion of this thesis.

Mom, thank you for your support in the past 21 years. You have never given up the effort to make me a better person. Your support is crucial for the completion of this thesis, and eventually the bachelor degree.

Contents

1. Introduction

We usually use formal verification techniques to correctly implement some given formal specifications. In order to construct a system of reliability, high integrity and robust performance, it goes through three phases: modeling, design and verification phase. Usually, the verification phase comes after the design phase as an independent step, which causes error-finding to be very troublesome for engineers, if the system fails to achieve the specification requirements. In the approach provided by[1], both phases are merged to automated correct-by-construction formal synthesis procedures. Formal synthesis is the controller design technique for continous control systems.[2]

SCOTS is a software tool that executes the above-mentioned procedures. It synthesizes controllers automatically for nonlinear control systems based on symbolic models. A symbolic model is also referred to an abstraction, a substituting finite system for an infinite system. A finite system is described by transition systems with finite state, input and output alphabets and has already been well researched. In real life, we require continuous control systems or infinite systems, which are then approximated to finite systems to simulate their behaviours.

The complete workflow of SCOTS encompasses three steps. Firstly, the concrete infinite system with the specification is transformed to an abstract domain and is replaced by a finite system. Then the system with the given specification in the abstract domain is solved by the existing methods for finite systems. Finally, the synthesized controller in the abstract domain is refined to the concrete system via feedback refinement relations.[1] Beyond that, SCOTS offers a Matlab interface to enable the simulation of the synthesized controller, which verifies the existence of the controller, as Rodney Brooks pointed out, “The problem with simulations is that they are doomed to succeed”.

Remove this big space

2. Rotary Inverted Pendulum

The pendulum[Figure 3][4] that is modelled in this thesis is designed by the company Quanser.
The thesis mainly focuses on the motion of the rotary arm and pendulum and aims at designing a symbolic controller that generates and adjusts inputs to execute the swing-up control. To achieve that goal, we need to feed SCOTS with three main mathematical components: the dynamics of the system

described by a state-space constructed by a differential equation system, a growth bound that ensures a parameter quantization step length doesn't exceed the range of the target and a target.

Figure 3: SRV02 ROTPEN system

This thesis presents two models: One is a motion model and the other is an energy model.

Remove this space

2.1 Motion Model

2.1.1 Dynamics

"depicts" is better here

"schematically" not mathematicaly here

[Figure 4][4] represents the pendulum model mathematically. The two most important parameters are θ and α. θ represents a clockwise angle of the rotary arm to its rotation axis, while α describes a clockwise angle of the pendulum to the rotary arm. We use V_m, the voltage, as our control input. The following nonlinear equations of motion for the SRV02 rotary inverted pendulum are given by[4]

$$(m_p L_r^2 + \tfrac{1}{4} m_p L_p^2 - \tfrac{1}{4} m_p L_p^2 cos(\alpha)^2 + J_r)\frac{\partial^2 \theta}{\partial t^2} - (\tfrac{1}{2} m_p L_p L_r cos(\alpha))\frac{\partial^2 \alpha}{\partial t^2} + (\tfrac{1}{2} m_p L_p^2 sin(\alpha) cos(\alpha))\frac{\partial \theta}{\partial t}\frac{\partial \alpha}{\partial t} + (\tfrac{1}{2} m_p L_p L_r sin(\alpha))(\frac{\partial \alpha}{\partial t})^2 = \tau - B_r \frac{\partial \theta}{\partial t}$$

Why use the partial derivitaves ? are you sure this was in the main model ? recheck this

$$-\frac{1}{2}m_pL_pL_r cos(\alpha)\frac{\partial^2\theta}{\partial t^2} + (J_p + \frac{1}{4}m_pL_p^2)\frac{\partial^2\alpha}{\partial t^2} - \frac{1}{4}m_pL_p^2 cos(\alpha)sin(\alpha)(\frac{\partial\theta}{\partial t})^2$$
$$-\frac{1}{2}m_pL_p g sin(\alpha) = -B_p\frac{\partial\alpha}{\partial t},$$

where $\tau = \frac{\eta_g K_g \eta_m k_t (V_m - K_g k_m \frac{\partial\theta}{\partial t})}{R_m}$

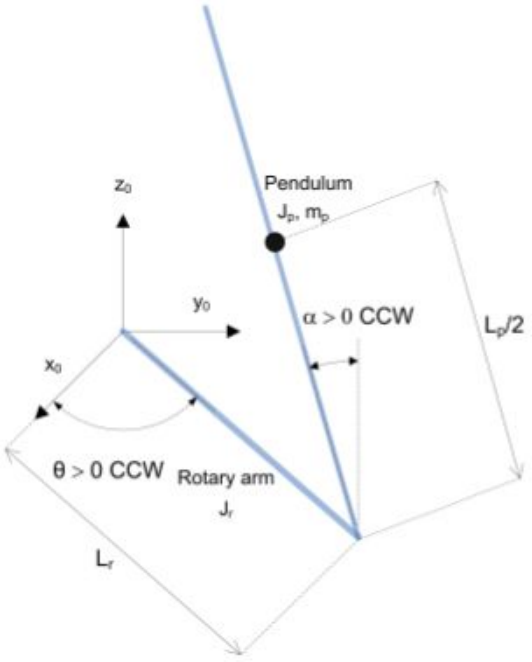


Figure 4

As we can observe from both equations, the $\frac{\partial^2\theta}{\partial t^2}$ and $\frac{\partial^2\alpha}{\partial t^2}$ are not expressed independently. In order to construct a state-space model of $x = \left[\theta\ \alpha\ \frac{\partial\theta}{\partial t}\ \frac{\partial\alpha}{\partial t}\right]^T$, we need to find the separate expressions of $\frac{\partial^2\theta}{\partial t^2}$ and $\frac{\partial^2\alpha}{\partial t^2}$. We use Symbolic Math Toolbox™ to achieve this goal[Appendix A].

again here symbols of partial derivatives !

We obtain the expressions for thetadotdot and alphadotdot.

```
solthetadotdot =

-(2*(8*Br*Jp*Rm*thetadot + 2*Br*Lp^2*Rm*mp*thetadot +
Lp^3*Lr*Rm*alphadot^2*mp^2*sin(alpha) - 8*Jp*Kg*etag*etam*kt*u +
4*Bp*Lp*Lr*Rm*alphadot*mp*cos(alpha) -
Lp^3*Lr*Rm*mp^2*thetadot^2*cos(alpha)^2*sin(alpha) +
8*Jp*Kg^2*etag*etam*km*kt*thetadot - 2*Kg*Lp^2*etag*etam*kt*mp*u -
2*Lp^2*Lr*Rm*g*mp^2*cos(alpha)*sin(alpha) +
4*Jp*Lp*Lr*Rm*alphadot^2*mp*sin(alpha) +
Lp^4*Rm*alphadot*mp^2*thetadot*cos(alpha)*sin(alpha) +
2*Kg^2*Lp^2*etag*etam*km*kt*mp*thetadot +
4*Jp*Lp^2*Rm*alphadot*mp*thetadot*cos(alpha)*sin(alpha)))/(Rm*(Lp^4*mp^2
```

```
+ 16*Jp*Jr + 4*Lp^2*Lr^2*mp^2 + 4*Jp*Lp^2*mp + 16*Jp*Lr^2*mp +
4*Jr*Lp^2*mp - Lp^4*mp^2*cos(alpha)^2 - 4*Lp^2*Lr^2*mp^2*cos(alpha)^2 -
4*Jp*Lp^2*mp*cos(alpha)^2))


solalphadotdot =

-(16*Bp*Jr*Rm*alphadot - 2*Lp^3*Rm*g*mp^2*sin(alpha) +
4*Bp*Lp^2*Rm*alphadot*mp + 16*Bp*Lr^2*Rm*alphadot*mp +
2*Lp^3*Rm*g*mp^2*cos(alpha)^2*sin(alpha) -
4*Bp*Lp^2*Rm*alphadot*mp*cos(alpha)^2 -
Lp^4*Rm*mp^2*thetadot^2*cos(alpha)*sin(alpha) -
8*Lp*Lr^2*Rm*g*mp^2*sin(alpha) - 8*Jr*Lp*Rm*g*mp*sin(alpha) +
Lp^4*Rm*mp^2*thetadot^2*cos(alpha)^3*sin(alpha) +
4*Lp^2*Lr^2*Rm*alphadot^2*mp^2*cos(alpha)*sin(alpha) +
8*Br*Lp*Lr*Rm*mp*thetadot*cos(alpha) -
4*Lp^2*Lr^2*Rm*mp^2*thetadot^2*cos(alpha)*sin(alpha) -
4*Jr*Lp^2*Rm*mp*thetadot^2*cos(alpha)*sin(alpha) +
4*Lp^3*Lr*Rm*alphadot*mp^2*thetadot*cos(alpha)^2*sin(alpha) -
8*Kg*Lp*Lr*etag*etam*kt*mp*u*cos(alpha) +
8*Kg^2*Lp*Lr*etag*etam*km*kt*mp*thetadot*cos(alpha))/(Rm*(Lp^4*mp^2 +
16*Jp*Jr + 4*Lp^2*Lr^2*mp^2 + 4*Jp*Lp^2*mp + 16*Jp*Lr^2*mp +
4*Jr*Lp^2*mp - Lp^4*mp^2*cos(alpha)^2 - 4*Lp^2*Lr^2*mp^2*cos(alpha)^2 -
4*Jp*Lp^2*mp*cos(alpha)^2))
```

Now, we can construct the state-space system by deriving the expressions for $\frac{\partial x}{\partial t}$. Considering the state-space system in C++ and Matlab only differs in syntax, we use Matlab to explain the construction of the state-space system to enhance clarity. $x(i)$ represents the $i-th$ component of x. Define $x(1) = \theta,\ x(2) = \alpha,\ x(3) = \frac{\partial \theta}{\partial t},\ x(4) = \frac{\partial \alpha}{\partial t}$. Then we derive the expression for $\frac{\partial x}{\partial t}$. We obtain:

$\frac{\partial x(1)}{\partial t} = \frac{\partial \theta}{\partial t} = x(3)$

$\frac{\partial x(1)}{\partial t} = \frac{\partial \alpha}{\partial t} = x(4)$

$\frac{\partial x(1)}{\partial t} = \frac{\partial^2 \theta}{\partial t^2} = solthetadotdot$

$\frac{\partial x(1)}{\partial t} = \frac{\partial^2 \alpha}{\partial t^2} = solalphadotdot$

Inserting the above-derived expressions into $dxdt$ together with the parameter specification given by[4], we can construct the ode solver below in both Matlab and C++ syntax:

Move big-listings to appendix and provide a small snippet of it !

```
% rotaryinvertedpendulum_ode.m

function dxdt = rotaryinvertedpendulum_ode(t, x, u)
Bp=0.002400000000000;
Br=0.002400000000000;
g=9.810000000000000;
Jp=0.001200000000000;
```

```
Jr=9.980000000000000e-04;
Lp=0.337000000000000;
Lr=0.216000000000000;
mp=0.127000000000000;
mr=0.257000000000000;
Kg = 70;
kt = 0.00768;
km = 0.00768;
Rm = 2.6;
etag = 0.90;
etam = 0.69;
tau = etag*Kg*etam*kt*(u-Kg*km*x(3))/Rm;
% theta=x(1)
% alpha=x(2)
% thetadot=x(3)
% alphadot=x(4)

dxdt=[x(3);x(4);-(2*(8*Br*Jp*x(3) - 8*Jp*tau - 2*Lp^2*mp*tau +
2*Br*Lp^2*mp*x(3) + Lp^3*Lr*x(4)^2*mp^2*sin(x(2)) -
2*Lp^2*Lr*g*mp^2*cos(x(2))*sin(x(2)) + 4*Jp*Lp*Lr*x(4)^2*mp*sin(x(2)) +
Lp^4*x(4)*mp^2*x(3)*cos(x(2))*sin(x(2)) + 4*Bp*Lp*Lr*x(4)*mp*cos(x(2)) -
Lp^3*Lr*mp^2*x(3)^2*cos(x(2))^2*sin(x(2)) +
4*Jp*Lp^2*x(4)*mp*x(3)*cos(x(2))*sin(x(2))))/(Lp^4*mp^2 + 16*Jp*Jr +
4*Lp^2*Lr^2*mp^2 + 4*Jp*Lp^2*mp + 16*Jp*Lr^2*mp + 4*Jr*Lp^2*mp -
Lp^4*mp^2*cos(x(2))^2 - 4*Lp^2*Lr^2*mp^2*cos(x(2))^2 -
4*Jp*Lp^2*mp*cos(x(2))^2);-(16*Bp*Jr*x(4) - 2*Lp^3*g*mp^2*sin(x(2)) +
4*Bp*Lp^2*x(4)*mp + 16*Bp*Lr^2*x(4)*mp +
2*Lp^3*g*mp^2*cos(x(2))^2*sin(x(2)) - 4*Bp*Lp^2*x(4)*mp*cos(x(2))^2 -
Lp^4*mp^2*x(3)^2*cos(x(2))*sin(x(2)) - 8*Lp*Lr^2*g*mp^2*sin(x(2)) -
8*Lp*Lr*mp*tau*cos(x(2)) - 8*Jr*Lp*g*mp*sin(x(2)) +
Lp^4*mp^2*x(3)^2*cos(x(2))^3*sin(x(2)) -
4*Jr*Lp^2*mp*x(3)^2*cos(x(2))*sin(x(2)) +
4*Lp^2*Lr^2*x(4)^2*mp^2*cos(x(2))*sin(x(2)) +
8*Br*Lp*Lr*mp*x(3)*cos(x(2)) -
4*Lp^2*Lr^2*mp^2*x(3)^2*cos(x(2))*sin(x(2)) +
4*Lp^3*Lr*x(4)*mp^2*x(3)*cos(x(2))^2*sin(x(2)))/(Lp^4*mp^2 + 16*Jp*Jr +
4*Lp^2*Lr^2*mp^2 + 4*Jp*Lp^2*mp + 16*Jp*Lr^2*mp + 4*Jr*Lp^2*mp -
Lp^4*mp^2*cos(x(2))^2 - 4*Lp^2*Lr^2*mp^2*cos(x(2))^2 -
4*Jp*Lp^2*mp*cos(x(2))^2)];
end
```

```
/* rotaryinvertedpendulum.cc (ODE part) */
```

```
const double Bp=0.002400000000000;
const double Br=0.002400000000000;
const double g=9.810000000000000;
const double Jp=0.001200000000000;
const double Jr=9.980000000000000e-04;
const double Lp=0.337000000000000;
const double Lr=0.216000000000000;
const double mp=0.127000000000000;
const double mr=0.257000000000000;
const double Kg = 70;
const double kt = 0.00768;
const double km = 0.00768;
const double Rm = 2.6;
const double etag = 0.90;
const double etam = 0.69;

/* number of intermediate steps in the ode solver */
const int nint=5;
OdeSolver ode_solver(sDIM,nint,tau);

/* we integrate the rotaryinvertedpendulum ode by 0.005 sec (the result
is stored in x)  */
auto  rotaryinvertedpendulum_post = [](state_type &x, input_type &u) ->
void {

  /* the ode describing the rotaryinvertedpendulum */
 // theta=x[0]
 // alpha=x[1]
 // thetadot=x[2]
 // alphadot=x[3]
 // x[0] dot = xx[0]
 // x[1] dot = xx[1]
 // x[2] dot = xx[2]
 // x[3] dot = xx[3]
  auto rhs =[](state_type& xx,  const state_type &x, input_type &u) {
      xx[0] = x[2];
      xx[1] = x[3];
      xx[2] = -(2*(8*Br*Jp*x[2] -
8*Jp*etag*Kg*etam*kt*(u[0]-Kg*km*x[2])/Rm -
2*std::pow(Lp,2)*mp*etag*Kg*etam*kt*(u[0]-Kg*km*x[2])/Rm +
2*Br*std::pow(Lp,2)*mp*x[2] +
std::pow(Lp,3)*Lr*std::pow(x[3],2)*std::pow(mp,2)*std::sin(x[1]) -
2*std::pow(Lp,2)*Lr*g*std::pow(mp,2)*std::cos(x[1])*std::sin(x[1]) +
4*Jp*Lp*Lr*std::pow(x[3],2)*mp*std::sin(x[1]) +
std::pow(Lp,4)*x[3]*std::pow(mp,2)*x[2]*std::cos(x[1])*std::sin(x[1]) +
```

```
4*Bp*Lp*Lr*x[3]*mp*std::cos(x[1]) -
std::pow(Lp,3)*Lr*std::pow(mp,2)*std::pow(x[2],2)*std::pow(std::cos(x[1]
),2)*std::sin(x[1]) +
4*Jp*std::pow(Lp,2)*x[3]*mp*x[2]*std::cos(x[1])*std::sin(x[1])))/(std::p
ow(Lp,4)*std::pow(mp,2) + 16*Jp*Jr +
4*std::pow(Lp,2)*std::pow(Lr,2)*std::pow(mp,2) + 4*Jp*std::pow(Lp,2)*mp
+ 16*Jp*std::pow(Lr,2)*mp + 4*Jr*std::pow(Lp,2)*mp -
std::pow(Lp,4)*std::pow(mp,2)*std::pow(std::cos(x[1]),2) -
4*std::pow(Lp,2)*std::pow(Lr,2)*std::pow(mp,2)*std::pow(std::cos(x[1]),2
) - 4*Jp*std::pow(Lp,2)*mp*std::pow(std::cos(x[1]),2));
      xx[3] = -(16*Bp*Jr*x[3] -
2*std::pow(Lp,3)*g*std::pow(mp,2)*std::sin(x[1]) +
4*Bp*std::pow(Lp,2)*x[3]*mp + 16*Bp*std::pow(Lr,2)*x[3]*mp +
2*std::pow(Lp,3)*g*std::pow(mp,2)*std::pow(std::cos(x[1]),2)*std::sin(x[
1]) - 4*Bp*std::pow(Lp,2)*x[3]*mp*std::pow(std::cos(x[1]),2) -
std::pow(Lp,4)*std::pow(mp,2)*std::pow(x[2],2)*std::cos(x[1])*std::sin(x
[1]) - 8*Lp*std::pow(Lr,2)*g*std::pow(mp,2)*std::sin(x[1]) -
8*Lp*Lr*mp*etag*Kg*etam*kt*(u[0]-Kg*km*x[2])/Rm*std::cos(x[1]) -
8*Jr*Lp*g*mp*std::sin(x[1]) +
std::pow(Lp,4)*std::pow(mp,2)*std::pow(x[2],2)*std::pow(std::cos(x[1]),3
)*std::sin(x[1]) -
4*Jr*std::pow(Lp,2)*mp*std::pow(x[2],2)*std::cos(x[1])*std::sin(x[1]) +
4*std::pow(Lp,2)*std::pow(Lr,2)*std::pow(x[3],2)*std::pow(mp,2)*std::cos
(x[1])*std::sin(x[1]) + 8*Br*Lp*Lr*mp*x[2]*std::cos(x[1]) -
4*std::pow(Lp,2)*std::pow(Lr,2)*std::pow(mp,2)*std::pow(x[2],2)*std::cos
(x[1])*std::sin(x[1]) +
4*std::pow(Lp,3)*Lr*x[3]*std::pow(mp,2)*x[2]*std::pow(std::cos(x[1]),2)*
std::sin(x[1]))/(std::pow(Lp,4)*std::pow(mp,2) + 16*Jp*Jr +
4*std::pow(Lp,2)*std::pow(Lr,2)*std::pow(mp,2) + 4*Jp*std::pow(Lp,2)*mp
+ 16*Jp*std::pow(Lr,2)*mp + 4*Jr*std::pow(Lp,2)*mp -
std::pow(Lp,4)*std::pow(mp,2)*std::pow(std::cos(x[1]),2) -
4*std::pow(Lp,2)*std::pow(Lr,2)*std::pow(mp,2)*std::pow(std::cos(x[1]),2
) - 4*Jp*std::pow(Lp,2)*mp*std::pow(std::cos(x[1]),2));

  };
  ode_solver(rhs,x,u);
};
```

After successful implementation of the state-space system in both languages, we can begin working on defining growth bound.

2.1.2 Growth Bound

A growth bound is an upper bound on the first derivative of each state and can be obtained by a modified Jacobian matrix defined as follows[1]:

$$If\ i = j: \quad L_{i,j}(u) \geq D_j f_i(x,u)$$
$$Otherwise: L_{i,j}(u) \geq \left| D_j f_i(x,u) \right|$$

If you are using Latex, this can be done using CASE

We use Symbolic Math Toolbox™ to derive the modified Jacobian matrix[Appendix B].

Since each entry of the matrix Lsymbolic has awfully many terms, the paper presents each entry separately:

Herea again: big-listing to appendix and provide snippets

```
>> Lsymbolic(3,1)

ans =

0

>> Lsymbolic(3,2)

ans =

abs((2*(2*Lp^2*Lr*Rm*g*mp^2*sin(alpha)^2 -
2*Lp^2*Lr*Rm*g*mp^2*cos(alpha)^2 +
Lp^4*Rm*alphadot*mp^2*thetadot*cos(alpha)^2 -
Lp^4*Rm*alphadot*mp^2*thetadot*sin(alpha)^2 -
Lp^3*Lr*Rm*mp^2*thetadot^2*cos(alpha)^3 +
Lp^3*Lr*Rm*alphadot^2*mp^2*cos(alpha) +
4*Jp*Lp^2*Rm*alphadot*mp*thetadot*cos(alpha)^2 -
4*Jp*Lp^2*Rm*alphadot*mp*thetadot*sin(alpha)^2 -
4*Bp*Lp*Lr*Rm*alphadot*mp*sin(alpha) +
2*Lp^3*Lr*Rm*mp^2*thetadot^2*cos(alpha)*sin(alpha)^2 +
4*Jp*Lp*Lr*Rm*alphadot^2*mp*cos(alpha)))/(Rm*(Lp^4*mp^2 + 16*Jp*Jr +
4*Lp^2*Lr^2*mp^2 + 4*Jp*Lp^2*mp + 16*Jp*Lr^2*mp + 4*Jr*Lp^2*mp -
Lp^4*mp^2*cos(alpha)^2 - 4*Lp^2*Lr^2*mp^2*cos(alpha)^2 -
4*Jp*Lp^2*mp*cos(alpha)^2)) - (2*(2*cos(alpha)*sin(alpha)*Lp^4*mp^2 +
8*cos(alpha)*sin(alpha)*Lp^2*Lr^2*mp^2 +
8*Jp*cos(alpha)*sin(alpha)*Lp^2*mp)*(8*Br*Jp*Rm*thetadot +
2*Br*Lp^2*Rm*mp*thetadot + Lp^3*Lr*Rm*alphadot^2*mp^2*sin(alpha) -
8*Jp*Kg*etag*etam*kt*u + 4*Bp*Lp*Lr*Rm*alphadot*mp*cos(alpha) -
Lp^3*Lr*Rm*mp^2*thetadot^2*cos(alpha)^2*sin(alpha) +
8*Jp*Kg^2*etag*etam*km*kt*thetadot - 2*Kg*Lp^2*etag*etam*kt*mp*u -
2*Lp^2*Lr*Rm*g*mp^2*cos(alpha)*sin(alpha) +
```

```
4*Jp*Lp*Lr*Rm*alphadot^2*mp*sin(alpha) +
Lp^4*Rm*alphadot*mp^2*thetadot*cos(alpha)*sin(alpha) +
2*Kg^2*Lp^2*etag*etam*km*kt*mp*thetadot +
4*Jp*Lp^2*Rm*alphadot*mp*thetadot*cos(alpha)*sin(alpha)))/(Rm*(Lp^4*mp^2
+ 16*Jp*Jr + 4*Lp^2*Lr^2*mp^2 + 4*Jp*Lp^2*mp + 16*Jp*Lr^2*mp +
4*Jr*Lp^2*mp - Lp^4*mp^2*cos(alpha)^2 - 4*Lp^2*Lr^2*mp^2*cos(alpha)^2 -
4*Jp*Lp^2*mp*cos(alpha)^2)^2))

>> Lsymbolic(3,3)

ans =

-(2*(8*Br*Jp*Rm + 2*Br*Lp^2*Rm*mp +
Lp^4*Rm*alphadot*mp^2*cos(alpha)*sin(alpha) + 8*Jp*Kg^2*etag*etam*km*kt
+ 4*Jp*Lp^2*Rm*alphadot*mp*cos(alpha)*sin(alpha) +
2*Kg^2*Lp^2*etag*etam*km*kt*mp -
2*Lp^3*Lr*Rm*mp^2*thetadot*cos(alpha)^2*sin(alpha)))/(Rm*(Lp^4*mp^2 +
16*Jp*Jr + 4*Lp^2*Lr^2*mp^2 + 4*Jp*Lp^2*mp + 16*Jp*Lr^2*mp +
4*Jr*Lp^2*mp - Lp^4*mp^2*cos(alpha)^2 - 4*Lp^2*Lr^2*mp^2*cos(alpha)^2 -
4*Jp*Lp^2*mp*cos(alpha)^2))

>> Lsymbolic(3,4)

ans =

(2*abs(Lp^4*Rm*mp^2*thetadot*cos(alpha)*sin(alpha) +
2*Lp^3*Lr*Rm*alphadot*mp^2*sin(alpha) + 4*Bp*Lp*Lr*Rm*mp*cos(alpha) +
4*Jp*Lp^2*Rm*mp*thetadot*cos(alpha)*sin(alpha) +
8*Jp*Lp*Lr*Rm*alphadot*mp*sin(alpha)))/(abs(Rm)*abs(Lp^4*mp^2 + 16*Jp*Jr
+ 4*Lp^2*Lr^2*mp^2 + 4*Jp*Lp^2*mp + 16*Jp*Lr^2*mp + 4*Jr*Lp^2*mp -
Lp^4*mp^2*cos(alpha)^2 - 4*Lp^2*Lr^2*mp^2*cos(alpha)^2 -
4*Jp*Lp^2*mp*cos(alpha)^2))

>> Lsymbolic(4,1)

ans =

0

>> Lsymbolic(4,2)

ans =

abs((Lp^4*Rm*mp^2*thetadot^2*cos(alpha)^2 -
Lp^4*Rm*mp^2*thetadot^2*cos(alpha)^4 -
```

```
Lp^4*Rm*mp^2*thetadot^2*sin(alpha)^2 + 2*Lp^3*Rm*g*mp^2*cos(alpha) -
2*Lp^3*Rm*g*mp^2*cos(alpha)^3 + 4*Jr*Lp^2*Rm*mp*thetadot^2*cos(alpha)^2
- 4*Jr*Lp^2*Rm*mp*thetadot^2*sin(alpha)^2 +
3*Lp^4*Rm*mp^2*thetadot^2*cos(alpha)^2*sin(alpha)^2 +
4*Lp^3*Rm*g*mp^2*cos(alpha)*sin(alpha)^2 +
8*Lp*Lr^2*Rm*g*mp^2*cos(alpha) -
4*Lp^2*Lr^2*Rm*alphadot^2*mp^2*cos(alpha)^2 +
4*Lp^2*Lr^2*Rm*alphadot^2*mp^2*sin(alpha)^2 +
4*Lp^2*Lr^2*Rm*mp^2*thetadot^2*cos(alpha)^2 -
4*Lp^2*Lr^2*Rm*mp^2*thetadot^2*sin(alpha)^2 + 8*Jr*Lp*Rm*g*mp*cos(alpha)
- 8*Bp*Lp^2*Rm*alphadot*mp*cos(alpha)*sin(alpha) +
8*Br*Lp*Lr*Rm*mp*thetadot*sin(alpha) -
4*Lp^3*Lr*Rm*alphadot*mp^2*thetadot*cos(alpha)^3 +
8*Lp^3*Lr*Rm*alphadot*mp^2*thetadot*cos(alpha)*sin(alpha)^2 -
8*Kg*Lp*Lr*etag*etam*kt*mp*u*sin(alpha) +
8*Kg^2*Lp*Lr*etag*etam*km*kt*mp*thetadot*sin(alpha))/(Rm*(Lp^4*mp^2 +
16*Jp*Jr + 4*Lp^2*Lr^2*mp^2 + 4*Jp*Lp^2*mp + 16*Jp*Lr^2*mp +
4*Jr*Lp^2*mp - Lp^4*mp^2*cos(alpha)^2 - 4*Lp^2*Lr^2*mp^2*cos(alpha)^2 -
4*Jp*Lp^2*mp*cos(alpha)^2)) + ((2*cos(alpha)*sin(alpha)*Lp^4*mp^2 +
8*cos(alpha)*sin(alpha)*Lp^2*Lr^2*mp^2 +
8*Jp*cos(alpha)*sin(alpha)*Lp^2*mp)*(16*Bp*Jr*Rm*alphadot -
2*Lp^3*Rm*g*mp^2*sin(alpha) + 4*Bp*Lp^2*Rm*alphadot*mp +
16*Bp*Lr^2*Rm*alphadot*mp + 2*Lp^3*Rm*g*mp^2*cos(alpha)^2*sin(alpha) -
4*Bp*Lp^2*Rm*alphadot*mp*cos(alpha)^2 -
Lp^4*Rm*mp^2*thetadot^2*cos(alpha)*sin(alpha) -
8*Lp*Lr^2*Rm*g*mp^2*sin(alpha) - 8*Jr*Lp*Rm*g*mp*sin(alpha) +
Lp^4*Rm*mp^2*thetadot^2*cos(alpha)^3*sin(alpha) +
4*Lp^2*Lr^2*Rm*alphadot^2*mp^2*cos(alpha)*sin(alpha) +
8*Br*Lp*Lr*Rm*mp*thetadot*cos(alpha) -
4*Lp^2*Lr^2*Rm*mp^2*thetadot^2*cos(alpha)*sin(alpha) -
4*Jr*Lp^2*Rm*mp*thetadot^2*cos(alpha)*sin(alpha) +
4*Lp^3*Lr*Rm*alphadot*mp^2*thetadot*cos(alpha)^2*sin(alpha) -
8*Kg*Lp*Lr*etag*etam*kt*mp*u*cos(alpha) +
8*Kg^2*Lp*Lr*etag*etam*km*kt*mp*thetadot*cos(alpha)))/(Rm*(Lp^4*mp^2 +
16*Jp*Jr + 4*Lp^2*Lr^2*mp^2 + 4*Jp*Lp^2*mp + 16*Jp*Lr^2*mp +
4*Jr*Lp^2*mp - Lp^4*mp^2*cos(alpha)^2 - 4*Lp^2*Lr^2*mp^2*cos(alpha)^2 -
4*Jp*Lp^2*mp*cos(alpha)^2)^2))

>> Lsymbolic(4,3)

ans =

abs(2*Lp^4*Rm*mp^2*thetadot*cos(alpha)^3*sin(alpha) -
2*Lp^4*Rm*mp^2*thetadot*cos(alpha)*sin(alpha) +
8*Br*Lp*Lr*Rm*mp*cos(alpha) -
```

```
8*Jr*Lp^2*Rm*mp*thetadot*cos(alpha)*sin(alpha) +
4*Lp^3*Lr*Rm*alphadot*mp^2*cos(alpha)^2*sin(alpha) -
8*Lp^2*Lr^2*Rm*mp^2*thetadot*cos(alpha)*sin(alpha) +
8*Kg^2*Lp*Lr*etag*etam*km*kt*mp*cos(alpha))/(abs(Rm)*abs(Lp^4*mp^2 +
16*Jp*Jr + 4*Lp^2*Lr^2*mp^2 + 4*Jp*Lp^2*mp + 16*Jp*Lr^2*mp +
4*Jr*Lp^2*mp - Lp^4*mp^2*cos(alpha)^2 - 4*Lp^2*Lr^2*mp^2*cos(alpha)^2 -
4*Jp*Lp^2*mp*cos(alpha)^2))

>> Lsymbolic(4,4)

ans =

-(16*Bp*Jr*Rm + 4*Bp*Lp^2*Rm*mp + 16*Bp*Lr^2*Rm*mp -
4*Bp*Lp^2*Rm*mp*cos(alpha)^2 +
8*Lp^2*Lr^2*Rm*alphadot*mp^2*cos(alpha)*sin(alpha) +
4*Lp^3*Lr*Rm*mp^2*thetadot*cos(alpha)^2*sin(alpha))/(Rm*(Lp^4*mp^2 +
16*Jp*Jr + 4*Lp^2*Lr^2*mp^2 + 4*Jp*Lp^2*mp + 16*Jp*Lr^2*mp +
4*Jr*Lp^2*mp - Lp^4*mp^2*cos(alpha)^2 - 4*Lp^2*Lr^2*mp^2*cos(alpha)^2 -
4*Jp*Lp^2*mp*cos(alpha)^2))
We take the growth bound r[0] for x[0] (theta) as an example. The
pseudo-code is
r[0]=r[0]+sampling_time*(r[0]*Lsymbolic(1,1)+r[1]*Lsymbolic(1,2)+r[2]*Ls
ymbolic(1,3)+r[3]*Lsymbolic*(1,4))
```

We update the growth bound from all partial derivation directions within the sampling time.
The reason that we don't solve the growth bound directly as beta is, we need to update the growth bound each sampling time. The exp term in beta is also very computationally expensive.

2.1.2 Target

SCOTS provides four functions to define the target set[5] :
SymbolicSet::addPolytope() /*add grid points associated with a polytope to symbolicSet_ */

SymbolicSet::remPolytope() /*remove grid points associated with a polytope from symbolicSet_ */

SymbolicSet::addEllipsoid() /*add grid points associated with an ellipsoid to symbolicSet_ */

SymbolicSet::remEllipsoid() /*remove grid points associated with an ellipsoid f

As the function names suggest, the target set is defined geometrically through polytopes of straight-line boundaries and ellipsoids of curved boundaries. The thesis defines the target set with an inequality restriction of each state by using polytope boundaries.

Since we only need the pendulum to swing up, the rotation angle of the rotary arm θ may have the maximal tolerance, as long as it doesn't exceed its domain. We need to restrict the rotation angle of the pendulum α to a domain around zero. The angular velocities $\frac{\partial\theta}{\partial t}$ and $\frac{\partial\alpha}{\partial t}$ will be given with enough tolerance to allow for SCOTS to find at least a controller, because too many restrictions could lead to the non-existence of a controller.

In SCOTS, we have to give upper and lower bounds for each parameter and its corresponding quantization step length. Due to the complexity of the model, it's relatively difficult to analyze the range of each parameter by Matlab Simulink. However, Quanser has provided the Professorship with one month free trial of a functioning software interface to the model. Therefore, we extracted the range for each parameter and determined the quantization step length with the mathematically well defined input designed by the company. The quantization step length can be decreased in case no controllers can be synthesized to satisfy the specification, because a file containing the synthesized controller would still be generated even though it doesn't exist. Simulation by Matlab can test the existence of the controller. If valid initial conditions verified by Matlab lie within the domain of the target, there are two possible causes: One is the target set domain is too small to find target grid nodes, the other is the quantization step length is still too large to find a controller. The major work here is to find the balance of a well-defined target set and a good quantization. The execution of a four dimensional state-space system with a small quantization step length can take hours, because the number of the discretized states reaches a scale of billion.

again symbols of partial der.

Parameters	Range	Quantization step length
θ	[-1.5,1.5]	0.05
α	[-4,4]	0.1
$\frac{\partial\theta}{\partial t}$	[-15,15]	1.5
$\frac{\partial\alpha}{\partial t}$	[-20,25]	1.25
u	[-23.6,15.8]	0.5

So a proper range for our target set after numerous experiments would be:

here too

Parameters	Range
θ	[-1.5,1.5]
α	[-0.15,0.15]
$\frac{\partial\theta}{\partial t}$	[-5,5]

$\frac{\partial \alpha}{\partial t}$	[-10,10]

```
/* rotaryinvertedpendulum.cc (target set part) */

/* define the target set as a symbolic set */
  double H[8*sDIM]={-1, 0, 0, 0,
                     1, 0, 0, 0,
                     0,-1, 0, 0,
                     0, 1, 0, 0,
                  0, 0, -1, 0,
                  0, 0, 1, 0,
                  0, 0, 0, -1,
                      0, 0, 0, 1,};
  /* compute inner approximation of P={ x | H x<= h1 }  */
  double h[8] = {1.5,1.5,0.15,0.15,5,5,10,10};
  ts.addPolytope(8,H,h, scots::INNER);
  ts.writeToFile("rotaryinvertedpendulum_target.bdd");
```

The reason we restrict α to [-0.15,0.15] instead of 0 is that a controller has to be left with some tolerance viz. a very precise symbolic controller is very hard to synthesize. We have permitted a tolerance that contains two quantization step lengths.

viz ?

Now we have the major mathematical components ready and we can execute the code by first $ make and then ./rotaryinvertedpendulum[Appendix C].

The dynamics might be different, because Symbolic Math Toolbox™ could be error-prone and therefore I calculated the expression for $\frac{\partial^2 \theta}{\partial t^2}$ and $\frac{\partial^2 \alpha}{\partial t^2}$ by hand again for accuracy due to linearity of both expressions in the equations. The growth bound is set to zero, because I couldn't find a valid controller and I need to determine the scale of the parameter by finding at least a controller. Then I utilized Matlab to simulate, since SCOTS also provides a Matlab interface.

How did you know it is error-prone ?
Did MATLAB said this in the manual ?
provide reference

2.1.3 Matlab Simulation

The mex compilation and addpath command are similar to the ones that were introduced in vehicle example in the first section[Appendix D]:

```
>> addpath(genpath('../../../mfiles'))
>> rotaryinvertedpendulum
```

We get the following error after running the simulation:

```
Error using SymbolicSet/getInputs (line 97)
The state [-1.52843629632891 1.90099324956258 -3.23143157639926
-3.34910101809038] is not in the domain of
the controller stored in:rotaryinvertedpendulum_controller.bdd

Error in rotaryinvertedpendulum (line 40)
  u=controller.getInputs(y(end,:));
```

Since [-1.52843629632891 1.90099324956258 -3.23143157639926 -3.34910101809038] is within the given lower and upper bounds, a possible explanation for a functional controller not being found is that the quantization step length for the first state is too large. We can adjust the parameter accordingly to reach such a target. Due to long execution time of this model, we have to set this aside.

As we can see from the previous motion model, the execution time takes extremely long. Is it possible to find a simplified model? Our goal is to construct a controller that enables swing-up motion. θ, the rotation angle of the rotary arm is redundant. Is there an approach, which only considers α, the rotation angle of the pendulum?

Yes, and the answer is energy model.

2.2 Energy Model

2.2.1 Dynamics

We set pivot acceleration, $\frac{\partial^2\theta}{\partial t^2}$, as input. Later on, we can add an open loop PID to use voltage as our input.

$$J_p\frac{\partial^2\alpha}{\partial t^2}+\frac{1}{2}m_pgL_psin(\alpha)=\frac{1}{2}m_pL_pucos(\alpha)$$

again partial derivatives

$$\frac{\partial E}{\partial t}=\frac{\partial\alpha}{\partial t}(J_p\frac{\partial^2\alpha}{\partial t^2}+\frac{1}{2}m_pgL_psin(\alpha))$$

We only need three states for this model, which are α, $\frac{\partial\alpha}{\partial t}$ and E. The energy model is much more simplified than the motion model. After conversion of the equations to a state-space into Matlab and C++ syntax respectively, we get the following ode solvers:

```
% rip_energy_ode.m (ODE part)

function dxdt = rip_energy_ode(t, x, u)
g=9.810000000000000;
Jp=0.001200000000000;
Lp=0.337000000000000;
mp=0.127000000000000;

dxdt(1)=x(2);
dxdt(2)=(mp*Lp/(2*Jp))*(u*cos(x(1))-g*sin(x(1)));
dxdt(3)=0.5*mp*Lp*u*x(2)*cos(x(1));
dxdt = dxdt';
end
```

```
/* rip_energy.cc (ODE part) */

/* other constant parameters */
const double g=9.810000000000000;
const double Jp=0.001200000000000;
const double Lp=0.337000000000000;
const double mp=0.127000000000000;

/* number of intermediate steps in the ode solver */
const int nint=5;
OdeSolver ode_solver(sDIM,nint,tau);

/* we integrate the rip_energy ode by 0.005 sec (the result is stored in
```

```
x)  */
auto  rip_energy_post = [](state_type &x, input_type &u) -> void {

  /* the ode describing the rip_energy */
  auto rhs =[](state_type& xx,  const state_type &x, input_type &u) {
      xx[0] = x[1];
      xx[1] = (mp*Lp/(2*Jp))*(u[0]*std::cos(x[0])-g*std::sin(x[0]));
      xx[2] = 0.5*mp*Lp*u[0]*x[1]*std::cos(x[0]);

  };
  ode_solver(rhs,x,u);
};
```

2.2.2. Growth Bound

Lsymbolic ? The Lsymbolic can be attained by Symbolic Math Toolbox™ as well[Appendix E]

Now we can take another approach. Since we have expressed the dynamics of the model as a series of differential equations, we can do the same to our growth boundary, because $r\_new = r\_old + sampling\_time * L * r$, which is equivalent to the equation $(r\_new - r\_old)/sampling\_time = L * r$, which is $\frac{\partial r}{\partial t} = L * r$.

The C++ ode solver for growth bound is given as follows:

```
/* rip_energy.cc (Growth Bound part) */

/* computation of the growth bound (the result is stored in r)  */
auto radius_post = [](state_type &r, input_type &u) {

    static int flag = 0;

  /* the ode describing the growth bound */
  auto rhs =[](state_type& rr,  const state_type &r, input_type &u) {
      rr[0] = r[0]*0+ r[1]*1+ r[2]*0;
      rr[1] = r[0]*std::abs((-(mp*Lp)/(2*Jp))*(u[0]*std::sin(r[0]) +
g*std::cos(r[0])))+ r[1]*0
```

```
+ r[2]*0;
      rr[2] = r[0]*std::abs((mp*Lp/2)*u[0]*r[1]*sin(r[0]))+
r[1]*std::abs(0.5*mp*Lp*u[0]*cos(r[0]))
+ r[2]*0;

  };
  ode_solver(rhs,r,u);
```

No need for the line that prints the Redius: this was just for testing

```
    if(flag < 1){
        std::cout << "Radius: " << r[0] << ", " << r[1] << ", "  << r[2]
<< ", "  << std::endl;
        flag++;
    }
};
```

The growth bound turns out to be very similar to that of the previous approach, which is given in C++ below:

```
auto radius_post = [](state_type &r, input_type &u) {

    static int flag = 0;
    r[0] = tau*(r[0]*0+ r[1]*1+ r[2]*0);
    r[1] = tau*(r[0]*std::abs((-(mp*Lp)/(2*Jp))*(u[0]*std::sin(r[0]) +
g*std::cos(r[0])))+ r[1]*0
+ r[2]*0);
    r[2] =
tau*(r[0]*std::abs((mp*Lp/2)*u[0]*r[1]*sin(r[0]))+r[1]*std::abs(0.5*mp*L
p*u[0]*cos(r[0]))                                    + r[2]*0);
```

Again no need for this radius printing

```
    if(flag < 1){
        std::cout << "Radius: " << r[0] << ", " << r[1] << ", "  << r[2]
<< ", "  << std::endl;
        flag++;
    }
};
```

For the ode approach, we have a growth bound as

Radius: 0.0168695, 0.29177, 0.0162701,

For the previous approach, we have a growth bound as

Radius: 0.01, 0.0870203, 0.000465571,

The three radii represent growth bound for α, $\frac{\partial \alpha}{\partial t}$ and E respectively.

2.2.3 Parameters' Domain and Quantization Step Length

Mahmoud Khaled

Due to the expiration of the one-month free trial and simplicity of the model, my supervisor Mr. Khaled Mahmoud adviced and assisted me to estimate the domain of parameters and their quantization step length by using Matlab Simulink. The system below is a graphical representation of the state-space system converted from the differential equations listed in the subsection 2.2.1 Dynamics.

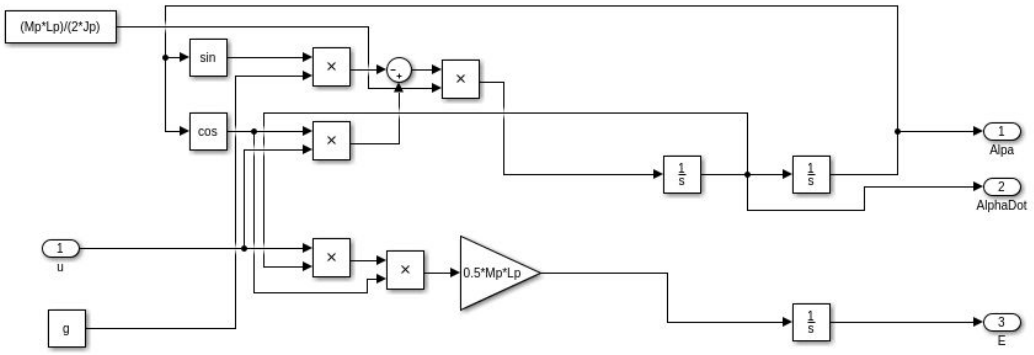


Figure 5: Subsystem

We set the initial condition for Alpha to 3.14[Figure 6], which refers to the pendulum pointing downwards. The whole model is illustrated by[Figure 7].

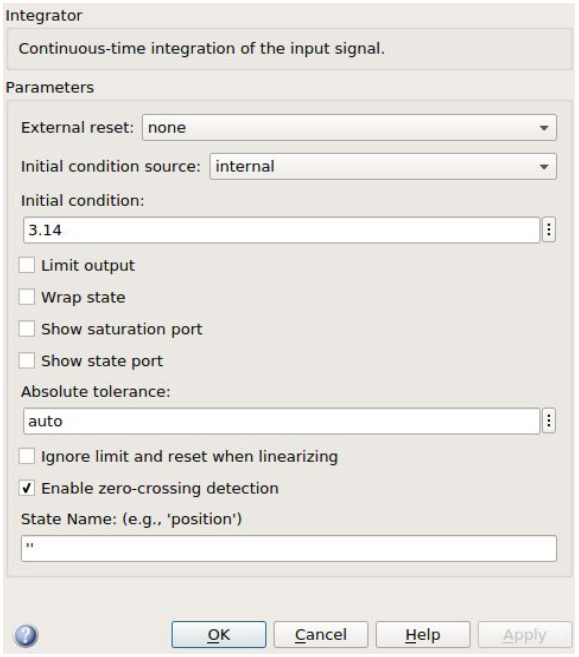


Figure 6: Initial Condition setting

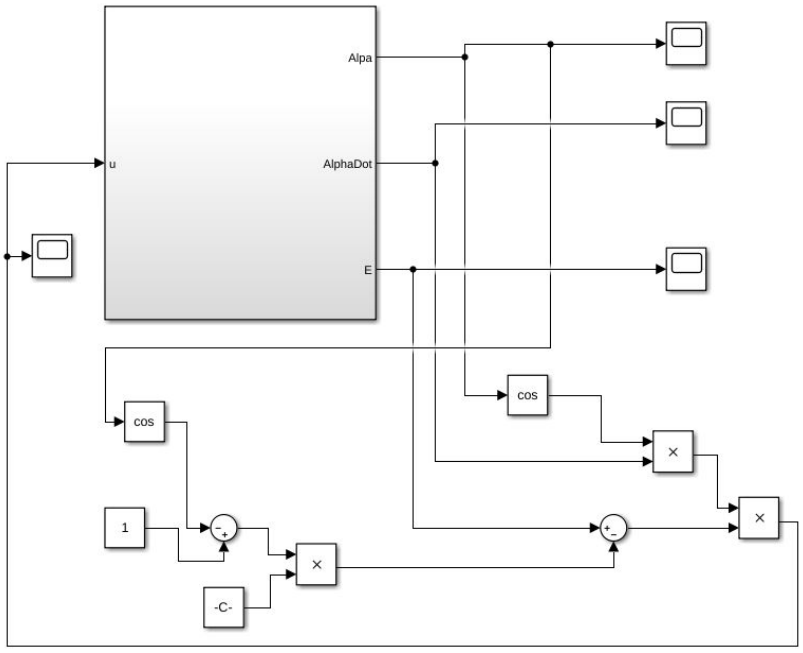


Figure 7: Energy Model

We first populate the parameters in the model:

```
% populate_params.m

Mp = 0.127;
Lp = 0.337;
g  = 9.81;
Jp = 0.0012;
```

Then we run Simulink and get the following measurements[Figure 8]:

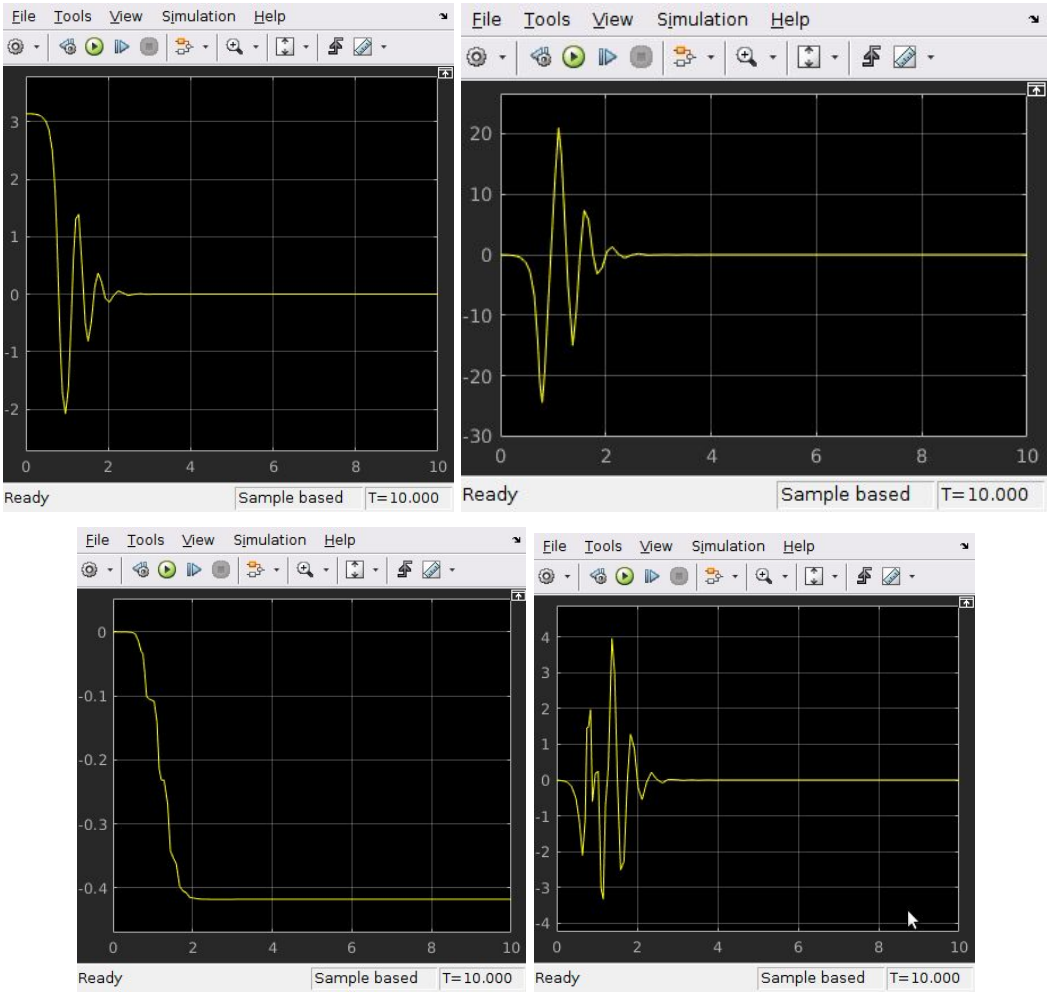


Figure 8: Upper left is a diagram depicting α's development regarding time. Upper right is $\frac{\partial \alpha}{\partial t}(t)$, lower left is $E(t)$ and lower right is $u(t)$ that is given by[4].

Now we need to determine the quantization step length by observing the first derivative of each parameter. We need to select the maximum derivative so that the parameter would "escape" the initial condition to reach our target set.

We already have Alphadot to represent the first derivative of Alpha. Therefore, we only need to insert two "Derivative" operators to observe the first derivatives of Alphadot and E respectively. The adjusted energy model is illustrated as follows[Figure 9]:

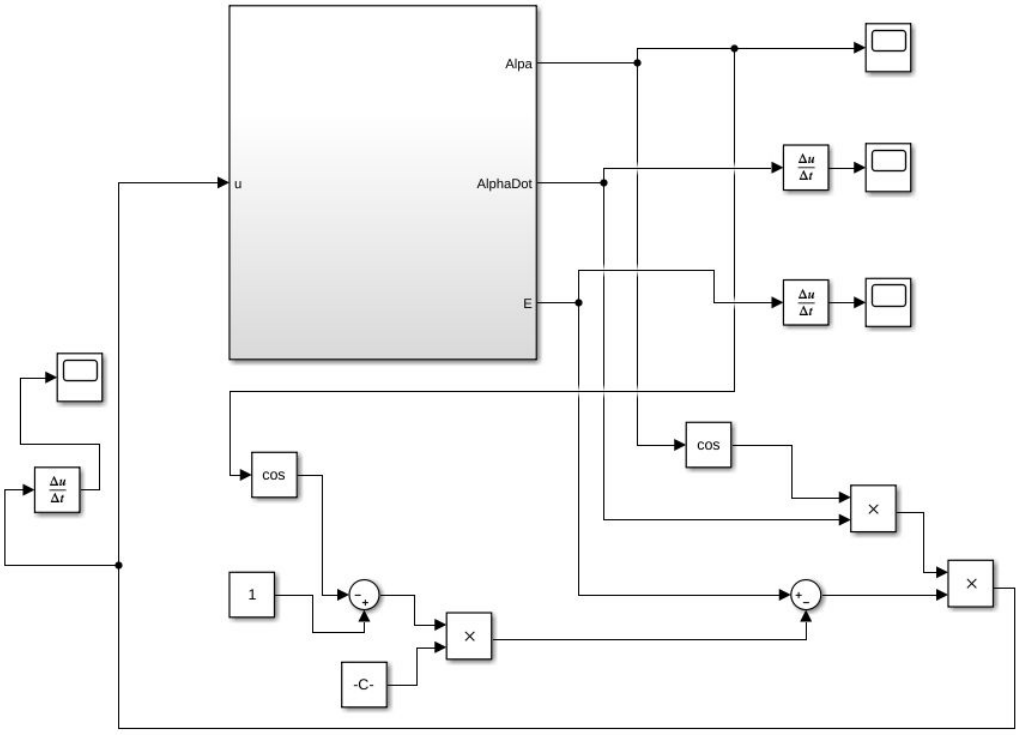
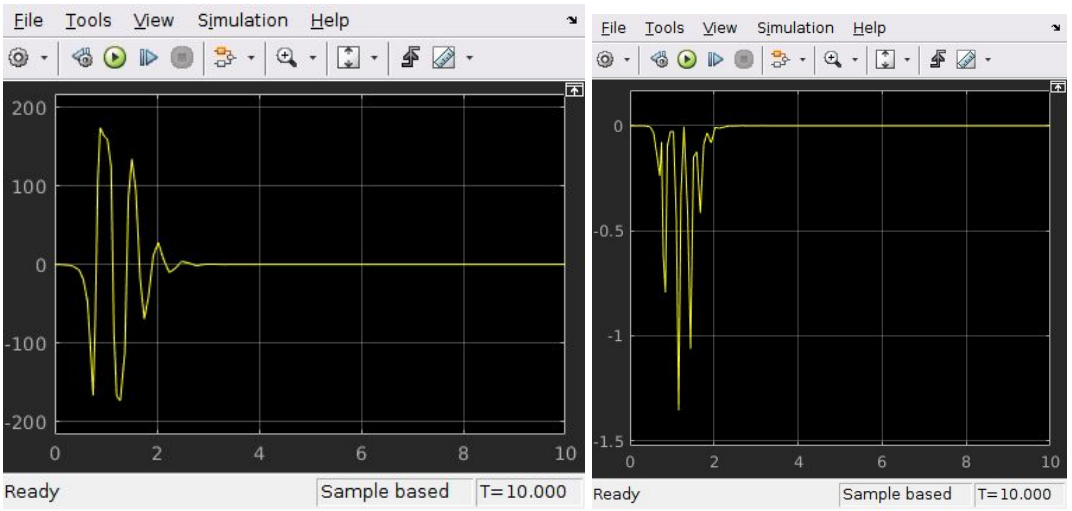


Figure 9

Now, we run Simulink again. We get the following measurements for the first derivatives of Alphado, E and u.[Figure 10]



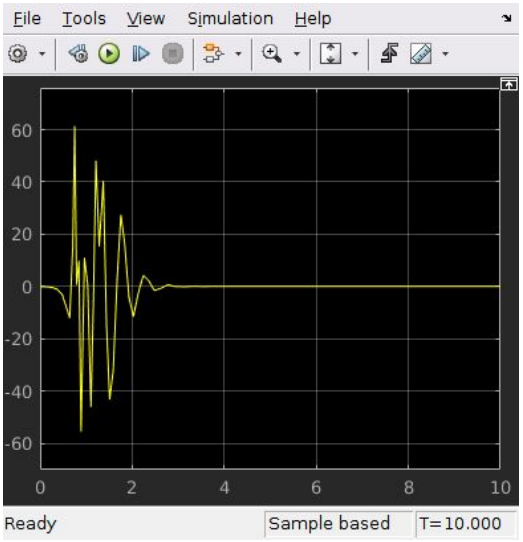


Figure 10: Upper left represents the first derivative for the Simulink state Alphadot, while the upper right represents that for the state E and the lower figure represents that for the input u.

The domain of state parameters can be simply read by the measurements above[Figure 8] and given with some tolerance to find at least one controller.
Quantization step length (qsl in equation) needs to be calculated by the following equation:
$qsl = 0.5 * sampling\_time * |max(first\ derivative)|$

Parameters	Range	Quantization step length
α	$[-\pi -0.4, \pi +0.4]$	0.5
$\dot{\alpha}$	[-5,5]	4
E	[-1,1]	0.3
u	[-5,5]	0.5

This quantization step length can still be adjusted (normally smaller) to allow for the existence of a controller. After testing several sets of parameters, the quantization step length is finalized as:

Parameters	Quantization step length
α	0.01
$\dot{\alpha}$	0.4
E	0.03
u	0.5

2.2.4 Target

As usual, we give α only a tolerance of two quantization step lengths. We give $\frac{\partial \alpha}{\partial t}$ and E rather large tolerances, but restricted compared to the motion model, because the energy model is less time-consuming to execute for testing parameters and synthesize a controller with a higher precision. We have the following target in C++:

```
/* rip_energy.cc (Target part) */

/* define the target set as a symbolic set */
  double H[6*sDIM]={-1, 0, 0,
                     1, 0, 0,
                     0,-1, 0,
                     0, 1, 0,
                 0, 0, -1,
                 0, 0, 1};
  /* compute inner approximation of P={ x | H x<= h1 }  */
  double h[6] = {0.06,0.06,1.5,1.5,1,1};
  ts.addPolytope(6,H,h, scots::INNER);
  ts.writeToFile("rip_energy_target.bdd");
```

Now we have all the key components of the code that is named as rip_energy.cc[Appendix F].

We make the rip_energy.cc file and execute and the following files are generated:
./rip_energy_ss.bdd
./rip_energy_obst.bdd
./rip_energy_target.bdd
./rip_energy_controller.bdd

2.2.5 Matlab Simulation

Now we can simulate the generated controller that is stored in the rip_enery_controller.bdd file with Matlab[Appendix G].

```
>> addpath(genpath('../../../mfiles'))
>> rip_energy
```

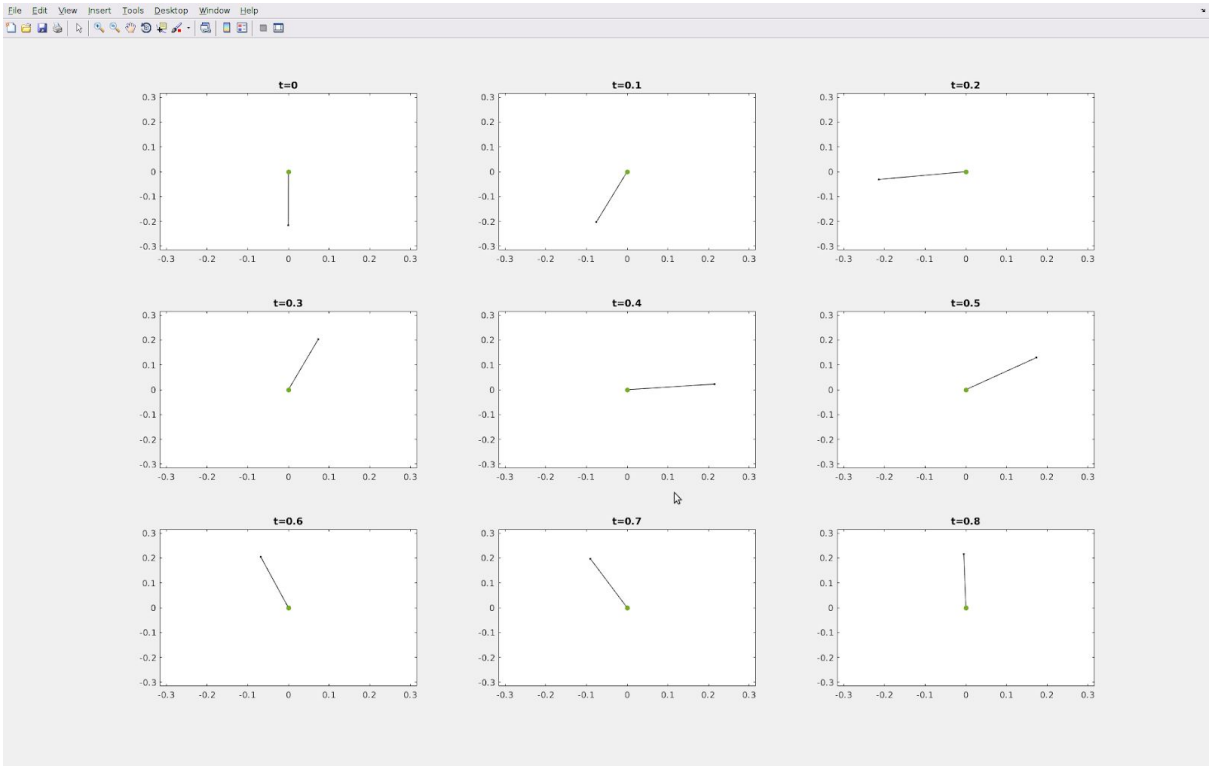


Figure 10: Trajectory

This figure describes the trajectory of the end point of the pendulum perpendicular to the rotary arm, since the position of the rotary arm is irrelevant.

References

[1] G. Reißig, A. Weber, and M. Rungger. *Feedback Refinement Relations for the Synthesis of Symbolic Controllers*. 2015. arXiv: 1503.03715 [math.OC,cs.SY] .
[2] M. Rungger and M. Zamani. “SCOTS: A Tool for the Synthesis of Symbolic Controllers”. In: *HSCC*.ACM, 2016.
[3] https://gitlab.lrz.de/hcs/scots/blob/master
[4] Quanser Inc. User Manual Inverted Pendulum Experiment Set Up and Configuration, 2012
[5] M. Rungger. SCOTS-User Manual

Appendix A

```
% Search for the expression of the second derivative of alpha and theta
% respectively regarding first and zeroth derivatives
syms theta thetadot thetadotdot alpha alphadot alphadotdot
syms mp Lr Lp Jr Jp g
syms tau Br Bp
syms etag Kg etam kt u km Kg Rm
tau = etag*Kg*etam*kt*(u-Kg*km*thetadot)/Rm;
eqns =
[(mp*Lr^2+mp*Lp^2/4-mp*Lp^2*cos(alpha)^2/4+Jr)*thetadotdot-mp*Lp*Lr*cos(
alpha)*alphadotdot/2+mp*Lp^2*sin(alpha)*cos(alpha)*thetadot*alphadot/2+m
p*Lp*Lr*sin(alpha)*alphadot^2/2==tau-Br*thetadot,-mp*Lp*Lr*cos(alpha)*th
etadotdot/2+(Jp+mp*Lp^2/4)*alphadotdot-mp*Lp^2*cos(alpha)*sin(alpha)*the
tadot^2/4-mp*Lp*g*sin(alpha)/2==-Bp*alphadot];
vars = [thetadotdot alphadotdot];
[solthetadotdot solalphadotdot]=solve(eqns, vars)
```

Appendix B

```
% constructionL.m

% Constructing Lsymbolic
% x=[theta alpha thetadot alphadot]
% the first derivative of the first state theta is the third state
thetadot
% x(1) dot= x(3) => The first row of the Jacobian matrix is therefore [0
0 1 0]
% It also applies to x(2) dot=x(4) => The second row of the Jacobian
matrix is therefore
Lsymbolic=[0 0 1 0 ; 0 0 0 1];
Lsymbolic=[Lsymbolic;abs(diff(solthetadotdot,theta))
abs(diff(solthetadotdot,alpha)) diff(solthetadotdot,thetadot)
abs(diff(solthetadotdot,alphadot))];
%Lsymbolic=[Lsymbolic;abs(diff(solalphadotdot,theta))
abs(diff(solalphadotdot,alpha)) abs(diff(solalphadotdot,thetadot))
diff(solalphadotdot,alphadot)];
Lsymbolic
```

Appendix C

```
/* rotaryinvertedpendulum */
#include <cmath>
#include <array>
#include <iostream>

#include "cuddObj.hh"

#include "SymbolicSet.hh"
#include "SymbolicModelGrowthBound.hh"

#include "TicToc.hh"
#include "RungeKutta4.hh"
#include "FixedPoint.hh"


/* state space dim */
#define sDIM 4
#define iDIM 1

/* data types for the ode solver */
typedef std::array<double,4> state_type;
typedef std::array<double,1> input_type;

/* sampling time */
const double tau = 0.005;
/* other constant parameters */
const double Bp=0.002400000000000;
const double Br=0.002400000000000;
const double g=9.810000000000000;
const double Jp=0.001200000000000;
const double Jr=0.002000000000000;
const double Lp=0.337000000000000;
const double Lr=0.216000000000000;
const double mp=0.127000000000000;
const double mr=0.257000000000000;
const double Kg = 70;
const double kt = 0.00768;
const double km = 0.00768;
const double Rm = 2.6;
const double etag = 0.90;
const double etam = 0.69;

/* number of intermediate steps in the ode solver */
```

```
const int nint=5;
OdeSolver ode_solver(sDIM,nint,tau);

/* we integrate the rotaryinvertedpendulum ode by 0.005 sec (the result
is stored in x)  */
auto  rotaryinvertedpendulum_post = [](state_type &x, input_type &u) ->
void {

  /* the ode describing the rotaryinvertedpendulum */
  auto rhs =[](state_type& xx,  const state_type &x, input_type &u) {
      xx[0] = x[2];
      xx[1] = x[3];
      xx[2] = -(((mp*std::pow(Lp,2))/4 +
Jp)*((mp*x[2]*std::cos(x[1])*std::sin(x[1])*std::pow(Lp,2)*x[3])/2 +
(Lr*mp*std::sin(x[1])*Lp*std::pow(x[3],2))/2 + Br*x[2] -
(Kg*etag*etam*kt*(u[0] - Kg*km*x[2]))/Rm) -
(Lp*Lr*mp*std::cos(x[1])*((mp*std::cos(x[1])*std::sin(x[1])*std::pow(Lp,
2)*std::pow(x[2],2))/4 + (g*mp*std::sin(x[1])*Lp)/2 -
Bp*x[3]))/2)/(((mp*std::pow(Lp,2))/4 + Jp)*(Jr + (std::pow(Lp,2)*mp)/4 +
std::pow(Lr,2)*mp - (std::pow(Lp,2)*mp*std::pow(std::cos(x[1]),2))/4) -
(std::pow(Lp,2)*std::pow(Lr,2)*std::pow(mp,2)*std::pow(std::cos(x[1]),2)
)/4);
      xx[3] =
(((mp*std::cos(x[1])*std::sin(x[1])*std::pow(Lp,2)*std::pow(x[2],2))/4 +
(g*mp*std::sin(x[1])*Lp)/2 - Bp*x[3])*(Jr + (std::pow(Lp,2)*mp)/4 +
std::pow(Lr,2)*mp - (std::pow(Lp,2)*mp*std::pow(std::cos(x[1]),2))/4) -
(Lp*Lr*mp*std::cos(x[1])*((mp*x[2]*std::cos(x[1])*std::sin(x[1])*std::po
w(Lp,2)*x[3])/2 + (Lr*mp*std::sin(x[1])*Lp*std::pow(x[3],2))/2 + Br*x[2]
- (Kg*etag*etam*kt*(u[0] - Kg*km*x[2]))/Rm))/2)/(((mp*std::pow(Lp,2))/4
+ Jp)*(Jr + (std::pow(Lp,2)*mp)/4 + std::pow(Lr,2)*mp -
(std::pow(Lp,2)*mp*std::pow(std::cos(x[1]),2))/4) -
(std::pow(Lp,2)*std::pow(Lr,2)*std::pow(mp,2)*std::pow(std::cos(x[1]),2)
)/4);

  };
  ode_solver(rhs,x,u);
};

/* computation of the growth bound (the result is stored in r)  */
auto radius_post = [](state_type &r, input_type &u) {

    static int flag = 0;


    r[0] = 0;//r[0]+1*r[2]*tau;
```

```
    r[1] = 0;//r[1]+1*r[3]*tau;
    r[2] =
0;//r[2]+r[1]*tau*(std::abs((2*(2*std::pow(Lp,2)*Lr*Rm*g*std::pow(mp,2)*
std::pow(std::sin(r[1]),2) -
2*std::pow(Lp,2)*Lr*Rm*g*std::pow(mp,2)*std::pow(std::cos(r[1]),2) +
std::pow(Lp,4)*Rm*r[3]*std::pow(mp,2)*r[2]*std::pow(std::cos(r[1]),2) -
std::pow(Lp,4)*Rm*r[3]*std::pow(mp,2)*r[2]*std::pow(std::sin(r[1]),2) -
std::pow(Lp,3)*Lr*Rm*std::pow(mp,2)*std::pow(r[2],2)*std::pow(std::cos(r
[1]),3) +
std::pow(Lp,3)*Lr*Rm*std::pow(r[3],2)*std::pow(mp,2)*std::cos(r[1]) +
4*Jp*std::pow(Lp,2)*Rm*r[3]*mp*r[2]*std::pow(std::cos(r[1]),2) -
4*Jp*std::pow(Lp,2)*Rm*r[3]*mp*r[2]*std::pow(std::sin(r[1]),2) -
4*Bp*Lp*Lr*Rm*r[3]*mp*std::sin(r[1]) +
2*std::pow(Lp,3)*Lr*Rm*std::pow(mp,2)*std::pow(r[2],2)*std::cos(r[1])*st
d::pow(std::sin(r[1]),2) +
4*Jp*Lp*Lr*Rm*std::pow(r[3],2)*mp*std::cos(r[1])))/(Rm*(std::pow(Lp,4)*s
td::pow(mp,2) + 16*Jp*Jr +
4*std::pow(Lp,2)*std::pow(Lr,2)*std::pow(mp,2) + 4*Jp*std::pow(Lp,2)*mp
+ 16*Jp*std::pow(Lr,2)*mp + 4*Jr*std::pow(Lp,2)*mp -
std::pow(Lp,4)*std::pow(mp,2)*std::pow(std::cos(r[1]),2) -
4*std::pow(Lp,2)*std::pow(Lr,2)*std::pow(mp,2)*std::pow(std::cos(r[1]),2
) - 4*Jp*std::pow(Lp,2)*mp*std::pow(std::cos(r[1]),2))) -
(2*(2*std::cos(r[1])*std::sin(r[1])*std::pow(Lp,4)*std::pow(mp,2) +
8*std::cos(r[1])*std::sin(r[1])*std::pow(Lp,2)*std::pow(Lr,2)*std::pow(m
p,2) +
8*Jp*std::cos(r[1])*std::sin(r[1])*std::pow(Lp,2)*mp)*(8*Br*Jp*Rm*r[2] +
2*Br*std::pow(Lp,2)*Rm*mp*r[2] +
std::pow(Lp,3)*Lr*Rm*std::pow(r[3],2)*std::pow(mp,2)*std::sin(r[1]) -
8*Jp*Kg*etag*etam*kt*u[0] + 4*Bp*Lp*Lr*Rm*r[3]*mp*std::cos(r[1]) -
std::pow(Lp,3)*Lr*Rm*std::pow(mp,2)*std::pow(r[2],2)*std::pow(std::cos(r
[1]),2)*std::sin(r[1]) + 8*Jp*std::pow(Kg,2)*etag*etam*km*kt*r[2] -
2*Kg*std::pow(Lp,2)*etag*etam*kt*mp*u[0] -
2*std::pow(Lp,2)*Lr*Rm*g*std::pow(mp,2)*std::cos(r[1])*std::sin(r[1]) +
4*Jp*Lp*Lr*Rm*std::pow(r[3],2)*mp*std::sin(r[1]) +
std::pow(Lp,4)*Rm*r[3]*std::pow(mp,2)*r[2]*std::cos(r[1])*std::sin(r[1])
+ 2*std::pow(Kg,2)*std::pow(Lp,2)*etag*etam*km*kt*mp*r[2] +
4*Jp*std::pow(Lp,2)*Rm*r[3]*mp*r[2]*std::cos(r[1])*std::sin(r[1])))/(Rm*
std::pow(std::pow(Lp,4)*std::pow(mp,2) + 16*Jp*Jr +
4*std::pow(Lp,2)*std::pow(Lr,2)*std::pow(mp,2) + 4*Jp*std::pow(Lp,2)*mp
+ 16*Jp*std::pow(Lr,2)*mp + 4*Jr*std::pow(Lp,2)*mp -
std::pow(Lp,4)*std::pow(mp,2)*std::pow(std::cos(r[1]),2) -
4*std::pow(Lp,2)*std::pow(Lr,2)*std::pow(mp,2)*std::pow(std::cos(r[1]),2
) - 4*Jp*std::pow(Lp,2)*mp*std::pow(std::cos(r[1]),2),2))))

//+r[2]*tau*(-(2*(8*Br*Jp*Rm + 2*Br*std::pow(Lp,2)*Rm*mp +
```

```
std::pow(Lp,4)*Rm*r[3]*std::pow(mp,2)*std::cos(r[1])*std::sin(r[1]) +
8*Jp*std::pow(Kg,2)*etag*etam*km*kt +
4*Jp*std::pow(Lp,2)*Rm*r[3]*mp*std::cos(r[1])*std::sin(r[1]) +
2*std::pow(Kg,2)*std::pow(Lp,2)*etag*etam*km*kt*mp -
2*std::pow(Lp,3)*Lr*Rm*std::pow(mp,2)*r[2]*std::pow(std::cos(r[1]),2)*st
d::sin(r[1])))/(Rm*(std::pow(Lp,4)*std::pow(mp,2) + 16*Jp*Jr +
4*std::pow(Lp,2)*std::pow(Lr,2)*std::pow(mp,2) + 4*Jp*std::pow(Lp,2)*mp
+ 16*Jp*std::pow(Lr,2)*mp + 4*Jr*std::pow(Lp,2)*mp -
std::pow(Lp,4)*std::pow(mp,2)*std::pow(std::cos(r[1]),2) -
4*std::pow(Lp,2)*std::pow(Lr,2)*std::pow(mp,2)*std::pow(std::cos(r[1]),2
) - 4*Jp*std::pow(Lp,2)*mp*std::pow(std::cos(r[1]),2))))

//+r[3]*tau*((2*std::abs(std::pow(Lp,4)*Rm*std::pow(mp,2)*r[2]*std::cos(
r[1])*std::sin(r[1]) +
2*std::pow(Lp,3)*Lr*Rm*r[3]*std::pow(mp,2)*std::sin(r[1]) +
4*Bp*Lp*Lr*Rm*mp*std::cos(r[1]) +
4*Jp*std::pow(Lp,2)*Rm*mp*r[2]*std::cos(r[1])*std::sin(r[1]) +
8*Jp*Lp*Lr*Rm*r[3]*mp*std::sin(r[1])))/(std::abs(Rm)*std::abs(std::pow(L
p,4)*std::pow(mp,2) + 16*Jp*Jr +
4*std::pow(Lp,2)*std::pow(Lr,2)*std::pow(mp,2) + 4*Jp*std::pow(Lp,2)*mp
+ 16*Jp*std::pow(Lr,2)*mp + 4*Jr*std::pow(Lp,2)*mp -
std::pow(Lp,4)*std::pow(mp,2)*std::pow(std::cos(r[1]),2) -
4*std::pow(Lp,2)*std::pow(Lr,2)*std::pow(mp,2)*std::pow(std::cos(r[1]),2
) - 4*Jp*std::pow(Lp,2)*mp*std::pow(std::cos(r[1]),2))));

    r[3] =
0;//r[3]+r[1]*tau*(std::abs((std::pow(Lp,4)*Rm*std::pow(mp,2)*std::pow(r
[2],2)*std::pow(std::cos(r[1]),2) -
std::pow(Lp,4)*Rm*std::pow(mp,2)*std::pow(r[2],2)*std::pow(std::cos(r[1]
),4) -
std::pow(Lp,4)*Rm*std::pow(mp,2)*std::pow(r[2],2)*std::pow(std::sin(r[1]
),2) + 2*std::pow(Lp,3)*Rm*g*std::pow(mp,2)*std::cos(r[1]) -
2*std::pow(Lp,3)*Rm*g*std::pow(mp,2)*std::pow(std::cos(r[1]),3) +
4*Jr*std::pow(Lp,2)*Rm*mp*std::pow(r[2],2)*std::pow(std::cos(r[1]),2) -
4*Jr*std::pow(Lp,2)*Rm*mp*std::pow(r[2],2)*std::pow(std::sin(r[1]),2) +
3*std::pow(Lp,4)*Rm*std::pow(mp,2)*std::pow(r[2],2)*std::pow(std::cos(r[
1]),2)*std::pow(std::sin(r[1]),2) +
4*std::pow(Lp,3)*Rm*g*std::pow(mp,2)*std::cos(r[1])*std::pow(std::sin(r[
1]),2) + 8*Lp*std::pow(Lr,2)*Rm*g*std::pow(mp,2)*std::cos(r[1]) -
4*std::pow(Lp,2)*std::pow(Lr,2)*Rm*std::pow(r[3],2)*std::pow(mp,2)*std::
pow(std::cos(r[1]),2) +
4*std::pow(Lp,2)*std::pow(Lr,2)*Rm*std::pow(r[3],2)*std::pow(mp,2)*std::
pow(std::sin(r[1]),2) +
4*std::pow(Lp,2)*std::pow(Lr,2)*Rm*std::pow(mp,2)*std::pow(r[2],2)*std::
pow(std::cos(r[1]),2) -
```

```
4*std::pow(Lp,2)*std::pow(Lr,2)*Rm*std::pow(mp,2)*std::pow(r[2],2)*std::
pow(std::sin(r[1]),2) + 8*Jr*Lp*Rm*g*mp*std::cos(r[1]) -
8*Bp*std::pow(Lp,2)*Rm*r[3]*mp*std::cos(r[1])*std::sin(r[1]) +
8*Br*Lp*Lr*Rm*mp*r[2]*std::sin(r[1]) -
4*std::pow(Lp,3)*Lr*Rm*r[3]*std::pow(mp,2)*r[2]*std::pow(std::cos(r[1]),
3) +
8*std::pow(Lp,3)*Lr*Rm*r[3]*std::pow(mp,2)*r[2]*std::cos(r[1])*std::pow(
std::sin(r[1]),2) - 8*Kg*Lp*Lr*etag*etam*kt*mp*u[0]*std::sin(r[1]) +
8*std::pow(Kg,2)*Lp*Lr*etag*etam*km*kt*mp*r[2]*std::sin(r[1]))/(Rm*(std:
:pow(Lp,4)*std::pow(mp,2) + 16*Jp*Jr +
4*std::pow(Lp,2)*std::pow(Lr,2)*std::pow(mp,2) + 4*Jp*std::pow(Lp,2)*mp
+ 16*Jp*std::pow(Lr,2)*mp + 4*Jr*std::pow(Lp,2)*mp -
std::pow(Lp,4)*std::pow(mp,2)*std::pow(std::cos(r[1]),2) -
4*std::pow(Lp,2)*std::pow(Lr,2)*std::pow(mp,2)*std::pow(std::cos(r[1]),2
) - 4*Jp*std::pow(Lp,2)*mp*std::pow(std::cos(r[1]),2))) +
((2*std::cos(r[1])*std::sin(r[1])*std::pow(Lp,4)*std::pow(mp,2) +
8*std::cos(r[1])*std::sin(r[1])*std::pow(Lp,2)*std::pow(Lr,2)*std::pow(m
p,2) +
8*Jp*std::cos(r[1])*std::sin(r[1])*std::pow(Lp,2)*mp)*(16*Bp*Jr*Rm*r[3]
- 2*std::pow(Lp,3)*Rm*g*std::pow(mp,2)*std::sin(r[1]) +
4*Bp*std::pow(Lp,2)*Rm*r[3]*mp + 16*Bp*std::pow(Lr,2)*Rm*r[3]*mp +
2*std::pow(Lp,3)*Rm*g*std::pow(mp,2)*std::pow(std::cos(r[1]),2)*std::sin
(r[1]) - 4*Bp*std::pow(Lp,2)*Rm*r[3]*mp*std::pow(std::cos(r[1]),2) -
std::pow(Lp,4)*Rm*std::pow(mp,2)*std::pow(r[2],2)*std::cos(r[1])*std::si
n(r[1]) - 8*Lp*std::pow(Lr,2)*Rm*g*std::pow(mp,2)*std::sin(r[1]) -
8*Jr*Lp*Rm*g*mp*std::sin(r[1]) +
std::pow(Lp,4)*Rm*std::pow(mp,2)*std::pow(r[2],2)*std::pow(std::cos(r[1]
),3)*std::sin(r[1]) +
4*std::pow(Lp,2)*std::pow(Lr,2)*Rm*std::pow(r[3],2)*std::pow(mp,2)*std::
cos(r[1])*std::sin(r[1]) + 8*Br*Lp*Lr*Rm*mp*r[2]*std::cos(r[1]) -
4*std::pow(Lp,2)*std::pow(Lr,2)*Rm*std::pow(mp,2)*std::pow(r[2],2)*std::
cos(r[1])*std::sin(r[1]) -
4*Jr*std::pow(Lp,2)*Rm*mp*std::pow(r[2],2)*std::cos(r[1])*std::sin(r[1])
+
4*std::pow(Lp,3)*Lr*Rm*r[3]*std::pow(mp,2)*r[2]*std::pow(std::cos(r[1]),
2)*std::sin(r[1]) - 8*Kg*Lp*Lr*etag*etam*kt*mp*u[0]*std::cos(r[1]) +
8*std::pow(Kg,2)*Lp*Lr*etag*etam*km*kt*mp*r[2]*std::cos(r[1]))/(Rm*std:
:pow(std::pow(Lp,4)*std::pow(mp,2) + 16*Jp*Jr +
4*std::pow(Lp,2)*std::pow(Lr,2)*std::pow(mp,2) + 4*Jp*std::pow(Lp,2)*mp
+ 16*Jp*std::pow(Lr,2)*mp + 4*Jr*std::pow(Lp,2)*mp -
std::pow(Lp,4)*std::pow(mp,2)*std::pow(std::cos(r[1]),2) -
4*std::pow(Lp,2)*std::pow(Lr,2)*std::pow(mp,2)*std::pow(std::cos(r[1]),2
) - 4*Jp*std::pow(Lp,2)*mp*std::pow(std::cos(r[1]),2),2))))

//+r[2]*tau*(std::abs(2*std::pow(Lp,4)*Rm*std::pow(mp,2)*r[2]*std::pow(s
```

```
td::cos(r[1]),3)*std::sin(r[1]) -
2*std::pow(Lp,4)*Rm*std::pow(mp,2)*r[2]*std::cos(r[1])*std::sin(r[1]) +
8*Br*Lp*Lr*Rm*mp*std::cos(r[1]) -
8*Jr*std::pow(Lp,2)*Rm*mp*r[2]*std::cos(r[1])*std::sin(r[1]) +
4*std::pow(Lp,3)*Lr*Rm*r[3]*std::pow(mp,2)*std::pow(std::cos(r[1]),2)*st
d::sin(r[1]) -
8*std::pow(Lp,2)*std::pow(Lr,2)*Rm*std::pow(mp,2)*r[2]*std::cos(r[1])*st
d::sin(r[1]) +
8*std::pow(Kg,2)*Lp*Lr*etag*etam*km*kt*mp*std::cos(r[1]))/(std::abs(Rm)*
std::abs(std::pow(Lp,4)*std::pow(mp,2) + 16*Jp*Jr +
4*std::pow(Lp,2)*std::pow(Lr,2)*std::pow(mp,2) + 4*Jp*std::pow(Lp,2)*mp
+ 16*Jp*std::pow(Lr,2)*mp + 4*Jr*std::pow(Lp,2)*mp -
std::pow(Lp,4)*std::pow(mp,2)*std::pow(std::cos(r[1]),2) -
4*std::pow(Lp,2)*std::pow(Lr,2)*std::pow(mp,2)*std::pow(std::cos(r[1]),2
) - 4*Jp*std::pow(Lp,2)*mp*std::pow(std::cos(r[1]),2))))

//+r[3]*tau*(-(16*Bp*Jr*Rm + 4*Bp*std::pow(Lp,2)*Rm*mp +
16*Bp*std::pow(Lr,2)*Rm*mp -
4*Bp*std::pow(Lp,2)*Rm*mp*std::pow(std::cos(r[1]),2) +
8*std::pow(Lp,2)*std::pow(Lr,2)*Rm*r[3]*std::pow(mp,2)*std::cos(r[1])*st
d::sin(r[1]) +
4*std::pow(Lp,3)*Lr*Rm*std::pow(mp,2)*r[2]*std::pow(std::cos(r[1]),2)*st
d::sin(r[1]))/(Rm*(std::pow(Lp,4)*std::pow(mp,2) + 16*Jp*Jr +
4*std::pow(Lp,2)*std::pow(Lr,2)*std::pow(mp,2) + 4*Jp*std::pow(Lp,2)*mp
+ 16*Jp*std::pow(Lr,2)*mp + 4*Jr*std::pow(Lp,2)*mp -
std::pow(Lp,4)*std::pow(mp,2)*std::pow(std::cos(r[1]),2) -
4*std::pow(Lp,2)*std::pow(Lr,2)*std::pow(mp,2)*std::pow(std::cos(r[1]),2
) - 4*Jp*std::pow(Lp,2)*mp*std::pow(std::cos(r[1]),2))));

    if(flag < 1){
        std::cout << "Radius: " << r[0] << ", " << r[1] << ", "  << r[2]
<< ", "  << r[3] << std::endl;
        flag++;
    }
};

/* forward declaration of the functions to setup the state space
 * and input space of the rotaryinvertedpendulum example */
scots::SymbolicSet rotaryinvertedpendulumCreateStateSpace(Cudd &mgr);
scots::SymbolicSet rotaryinvertedpendulumCreateInputSpace(Cudd &mgr);



int main() {
  /* to measure time */
```

```
  TicToc tt;
  /* there is one unique manager to organize the bdd variables */
  Cudd mgr;


/*************************************************************************
*****/
  /* construct SymbolicSet for the state space */

/*************************************************************************
*****/
  scots::SymbolicSet ss=rotaryinvertedpendulumCreateStateSpace(mgr);
  ss.writeToFile("rotaryinvertedpendulum_ss.bdd");


/*************************************************************************
*****/
  /* we define the target set */

/*************************************************************************
*****/
  /* first make a copy of the state space so that we obtain the grid
   * information in the new symbolic set */
  scots::SymbolicSet ts(ss);
  /* define the target set as a symbolic set */
  double H[8*sDIM]={-1, 0, 0, 0,
                     1, 0, 0, 0,
                     0,-1, 0, 0,
                     0, 1, 0, 0,
                 0, 0, -1, 0,
                 0, 0, 1, 0,
                 0, 0, 0, -1,
                     0, 0, 0, 1,};
  /* compute inner approximation of P={ x | H x<= h1 }  */
  double h[8] = {1.5,1.5,0.15,0.15,5,5,10,10};
  ts.addPolytope(8,H,h, scots::INNER);
  ts.writeToFile("rotaryinvertedpendulum_target.bdd");


/*************************************************************************
*****/
  /* construct SymbolicSet for the input space */

/*************************************************************************
*****/
```

```
  scots::SymbolicSet is=rotaryinvertedpendulumCreateInputSpace(mgr);


/*************************************************************************
*****/
  /* setup class for symbolic model computation */

/*************************************************************************
*****/
  /* first create SymbolicSet of post variables
   * by copying the SymbolicSet of the state space and assigning new BDD
IDs */
  scots::SymbolicSet sspost(ss,1);
  /* instantiate the SymbolicModel */
  scots::SymbolicModelGrowthBound<state_type,input_type>
abstraction(&ss, &is, &sspost);
  /* compute the transition relation */
  tt.tic();
  abstraction.computeTransitionRelation(rotaryinvertedpendulum_post,
radius_post);
  std::cout << std::endl;
  tt.toc();
  /* get the number of elements in the transition relation */
  std::cout << std::endl << "Number of elements in the transition
relation: " << abstraction.getSize() << std::endl;


/*************************************************************************
*****/
  /* we continue with the controller synthesis */

/*************************************************************************
*****/
  /* we setup a fixed point object to compute reachabilty controller */
  scots::FixedPoint fp(&abstraction);
  /* the fixed point algorithm operates on the BDD directly */
  BDD T = ts.getSymbolicSet();
  tt.tic();
  /* compute controller */
  BDD C=fp.reach(T,1);
  tt.toc();


/*************************************************************************
*****/
```

```
  /* last we store the controller as a SymbolicSet
   * the underlying uniform grid is given by the Cartesian product of
   * the uniform gird of the space and uniform gird of the input space
*/

/*****************************************************************
*****/
  scots::SymbolicSet controller(ss,is);
  controller.setSymbolicSet(C);
  controller.writeToFile("rotaryinvertedpendulum_controller.bdd");

  return 1;
}

scots::SymbolicSet rotaryinvertedpendulumCreateStateSpace(Cudd &mgr) {

  /* setup the workspace of the synthesis problem and the uniform grid
*/
  /* lower bounds of the states theta alpha thetadot alphadot */
  double lb[sDIM]={-1.5,-4,-15,-20};
  /* upper bounds of the states theta alpha thetadot alphadot */
  double ub[sDIM]={1.5,4,15,25};
  /* discrete angle(angular acceleration) step */
  double eta[sDIM]={0.05,0.1,1.5,1.25};


  scots::SymbolicSet ss(mgr,sDIM,lb,ub,eta);

  /* add the discretized states to the SymbolicSet ss */
  ss.addGridPoints();

 return ss;
}


scots::SymbolicSet rotaryinvertedpendulumCreateInputSpace(Cudd &mgr) {

  /* lower bounds of the input (voltage) */
  double lb[sDIM]={-23.6};
  /* upper bounds of the input (voltage) */
  double ub[sDIM]={15.8};
  /* grid node distance diameter */
  double eta[sDIM]={0.5};

  scots::SymbolicSet is(mgr,iDIM,lb,ub,eta);
```

```
  is.addGridPoints();

  return is;
}
```

Appendix D

```
% rotaryinvertedpendulum.m

function rotaryinvertedpendulum
clear set
close all



%% simulation
Lr=0.216000000000000;
lp=0.156;


% target set
lb=[-1.5 -0.15 -5 -10];
ub=[1.5 0.15 5 10];
x0=[0 pi 0 0];


controller=SymbolicSet('rotaryinvertedpendulum_controller.bdd','projecti
on',[1 2 3 4]);
target=SymbolicSet('rotaryinvertedpendulum_target.bdd');
y=x0;
v=[];

while(1)

  if (target.isElement(y(end,:)))
    break;
  end

  u=controller.getInputs(y(end,:));
  v=[v; u(1,:)];
  [t x]=ode45(@rotaryinvertedpendulum_ode,[0 0.005], y(end,:),
[],u(1,:));

  y=[y; x(end,:)];
end


% colors
colors=get(groot,'DefaultAxesColorOrder');
```

```
% load the symbolic set containig the abstract state space
set=SymbolicSet('rotaryinvertedpendulum_ss.bdd','projection',[1 2]);
hold on

% load the symbolic set containig target set
set=SymbolicSet('rotaryinvertedpendulum_target.bdd','projection',[1 2]);

% plot initial state  and trajectory
plot3(Lr*cos(y(:,1))+lp*sin(y(:,2))*sin(y(:,1)),Lr*sin(y(:,1))-lp*sin(y(
:,2))*cos(y(:,1)),lp*cos(y(:,2)),'k.-')
plot3(Lr*cos(y(1,1))+lp*sin(y(1,2))*sin(y(1,1)),Lr*sin(y(1,1))-lp*sin(y(
1,2))*cos(y(1,1)),lp*cos(y(1,2)),'.','color',colors(5,:),'markersize',20
)

end

% model by hand

function dxdt = rotaryinvertedpendulum_ode(t, x, u)
Bp=0.002400000000000;
Br=0.002400000000000;
g=9.810000000000000;
Jp=0.001200000000000;
Jr=0.002000000000000;
Lp=0.337000000000000;
Lr=0.216000000000000;
mp=0.127000000000000;
mr=0.257000000000000;
Kg = 70;
kt = 0.00768;
km = 0.00768;
Rm = 2.6;
etag = 0.90;
etam = 0.69;
%tau = etag*Kg*etam*kt*(u-Kg*km*x(3))/Rm;
dxdt(1)=x(3);
dxdt(2)=x(4);
dxdt(3)=-(((mp*Lp^2)/4 + Jp)*((mp*x(3)*cos(x(2))*sin(x(2))*Lp^2*x(4))/2
+ (Lr*mp*sin(x(2))*Lp*x(4)^2)/2 + Br*x(3) - (Kg*etag*etam*kt*(u -
Kg*km*x(3)))/Rm) -
(Lp*Lr*mp*cos(x(2))*((mp*cos(x(2))*sin(x(2))*Lp^2*x(3)^2)/4 +
(g*mp*sin(x(2))*Lp)/2 - Bp*x(4)))/2)/(((mp*Lp^2)/4 + Jp)*(Jr +
(Lp^2*mp)/4 + Lr^2*mp - (Lp^2*mp*cos(x(2))^2)/4) -
(Lp^2*Lr^2*mp^2*cos(x(2))^2)/4);
dxdt(4)=(((mp*cos(x(2))*sin(x(2))*Lp^2*x(3)^2)/4 + (g*mp*sin(x(2))*Lp)/2
```

```
- Bp*x(4))*(Jr + (Lp^2*mp)/4 + Lr^2*mp - (Lp^2*mp*cos(x(2))^2)/4) -
(Lp*Lr*mp*cos(x(2))*((mp*x(3)*cos(x(2))*sin(x(2))*Lp^2*x(4))/2 +
(Lr*mp*sin(x(2))*Lp*x(4)^2)/2 + Br*x(3) - (Kg*etag*etam*kt*(u -
Kg*km*x(3)))/Rm))/2)/(((mp*Lp^2)/4 + Jp)*(Jr + (Lp^2*mp)/4 + Lr^2*mp -
(Lp^2*mp*cos(x(2))^2)/4) - (Lp^2*Lr^2*mp^2*cos(x(2))^2)/4);
dxdt=dxdt';
end
```

Appendix E

```
% energy_bound.m

syms alpha alphadot alphadotdot e edot
syms mp Lr Lp Jr Jp g
syms tau Br Bp
syms etag Kg etam kt u km Kg Rm
alphadotdot=(mp*Lp/(2*Jp))*(u*cos(alpha)-g*sin(alpha));
edot=0.5*mp*Lp*u*alphadot*cos(alpha);
Lsymbolic=[0 1 0];
Lsymbolic=[Lsymbolic;abs(diff(alphadotdot,alpha))
diff(alphadotdot,alphadot) abs(diff(alphadotdot,e))];
Lsymbolic=[Lsymbolic;abs(diff(edot,alpha)) abs(diff(edot,alphadot))
diff(edot,e)];
Lsymbolic
```

Appendix F

```
/* rip_energy.cc */

#include <cmath>
#include <array>
#include <iostream>

#include "cuddObj.hh"

#include "SymbolicSet.hh"
#include "SymbolicModelGrowthBound.hh"

#include "TicToc.hh"
#include "RungeKutta4.hh"
#include "FixedPoint.hh"


/* state space dim */
#define sDIM 3
#define iDIM 1

/* data types for the ode solver */
typedef std::array<double,sDIM> state_type;
typedef std::array<double,iDIM> input_type;

/* sampling time */
const double tau = 0.05;
/* other constant parameters */
const double g=9.810000000000000;
const double Jp=0.001200000000000;
const double Lp=0.337000000000000;
const double mp=0.127000000000000;

/* number of intermediate steps in the ode solver */
const int nint=5;
OdeSolver ode_solver(sDIM,nint,tau);

/* we integrate the rip_energy ode by 0.005 sec (the result is stored in
x)  */
auto  rip_energy_post = [](state_type &x, input_type &u) -> void {

  /* the ode describing the rip_energy */
  auto rhs =[](state_type& xx,  const state_type &x, input_type &u) {
      xx[0] = x[1];
```

```
      xx[1] = (mp*Lp/(2*Jp))*(u[0]*std::cos(x[0])-g*std::sin(x[0]));
      xx[2] = 0.5*mp*Lp*u[0]*x[1]*std::cos(x[0]);

  };
  ode_solver(rhs,x,u);
};

/* computation of the growth bound (the result is stored in r)  */
auto radius_post = [](state_type &r, input_type &u) {

    static int flag = 0;

  /* the ode describing the rip_energy */
  auto rhs =[](state_type& rr,  const state_type &r, input_type &u) {
      rr[0] = r[0]*0
+ r[1]*1
+ r[2]*0;
      rr[1] = r[0]*std::abs((-(mp*Lp)/(2*Jp))*(u[0]*std::sin(r[0]) +
g*std::cos(r[0])))     + r[1]*0
+ r[2]*0;
      rr[2] = r[0]*std::abs((mp*Lp/2)*u[0]*r[1]*sin(r[0]))
+ r[1]*std::abs(0.5*mp*Lp*u[0]*cos(r[0]))
+ r[2]*0;

  };
  ode_solver(rhs,r,u);

    if(flag < 1){
        std::cout << "Radius: " << r[0] << ", " << r[1] << ", "  << r[2]
<< ", "  << std::endl;
        flag++;
    }
};

/* forward declaration of the functions to setup the state space
 * and input space of the rip_energy example */
scots::SymbolicSet rip_energyCreateStateSpace(Cudd &mgr);
scots::SymbolicSet rip_energyCreateInputSpace(Cudd &mgr);



int main() {
  /* to measure time */
  TicToc tt;
  /* there is one unique manager to organize the bdd variables */
```

```
  Cudd mgr;


/*************************************************************************
*****/
  /* construct SymbolicSet for the state space */

/*************************************************************************
*****/
  scots::SymbolicSet ss=rip_energyCreateStateSpace(mgr);
  ss.writeToFile("rip_energy_ss.bdd");


/*************************************************************************
*****/
  /* we define the target set */

/*************************************************************************
*****/
  /* first make a copy of the state space so that we obtain the grid
   * information in the new symbolic set */
  scots::SymbolicSet ts(ss);
  /* define the target set as a symbolic set */
  double H[6*sDIM]={-1, 0, 0,
                     1, 0, 0,
                     0,-1, 0,
                     0, 1, 0,
                 0, 0, -1,
                 0, 0, 1};
  /* compute inner approximation of P={ x | H x<= h1 }  */
  double h[6] = {0.06,0.06,1.5,1.5,1,1};
  ts.addPolytope(6,H,h, scots::INNER);
  ts.writeToFile("rip_energy_target.bdd");


/*************************************************************************
*****/
  /* construct SymbolicSet for the input space */

/*************************************************************************
*****/
  scots::SymbolicSet is=rip_energyCreateInputSpace(mgr);


/*************************************************************************
```

```
*****/
  /* setup class for symbolic model computation */

/*************************************************************************
*****/
  /* first create SymbolicSet of post variables
   * by copying the SymbolicSet of the state space and assigning new BDD
IDs */
  scots::SymbolicSet sspost(ss,1);
  /* instantiate the SymbolicModel */
  scots::SymbolicModelGrowthBound<state_type,input_type>
abstraction(&ss, &is, &sspost);
  /* compute the transition relation */
  tt.tic();
  abstraction.computeTransitionRelation(rip_energy_post, radius_post);
  std::cout << std::endl;
  tt.toc();
  /* get the number of elements in the transition relation */
  std::cout << std::endl << "Number of elements in the transition
relation: " << abstraction.getSize() << std::endl;


/*************************************************************************
*****/
  /* we continue with the controller synthesis */

/*************************************************************************
*****/
  /* we setup a fixed point object to compute reachabilty controller */
  scots::FixedPoint fp(&abstraction);
  /* the fixed point algorithm operates on the BDD directly */
  BDD T = ts.getSymbolicSet();
  tt.tic();
  /* compute controller */
  BDD C=fp.reach(T,1);
  tt.toc();


/*************************************************************************
*****/
  /* last we store the controller as a SymbolicSet
   * the underlying uniform grid is given by the Cartesian product of
   * the uniform gird of the space and uniform gird of the input space
*/
```

```
/*************************************************************************
*****/
  scots::SymbolicSet controller(ss,is);
  controller.setSymbolicSet(C);
  controller.writeToFile("rip_energy_controller.bdd");

  return 1;
}

scots::SymbolicSet rip_energyCreateStateSpace(Cudd &mgr) {

  /* setup the workspace of the synthesis problem and the uniform grid
*/
  /* lower bounds of the states theta alpha thetadot alphadot */
  double lb[sDIM]={-M_PI-0.4,-25,-1};
  /* upper bounds of the states theta alpha thetadot alphadot */
  double ub[sDIM]={M_PI+0.4,25,1};
  /* discrete angle(angular acceleration) step */
  double eta[sDIM]={0.01,0.4,0.03};


  scots::SymbolicSet ss(mgr,sDIM,lb,ub,eta);

  /* add the discretized states to the SymbolicSet ss */
  ss.addGridPoints();

 return ss;
}


scots::SymbolicSet rip_energyCreateInputSpace(Cudd &mgr) {

  /* lower bounds of the input (voltage) */
  double lb[sDIM]={-5};
  /* upper bounds of the input (voltage) */
  double ub[sDIM]={5};
  /* grid node distance diameter */
  double eta[sDIM]={0.5};

  scots::SymbolicSet is(mgr,iDIM,lb,ub,eta);
  is.addGridPoints();

  return is;
}
```

Appendix G

```
% rip_energy.m

function rip_energy
clear set
close all



%% simulation
% target set
lb=[-0.06 -1.5 -1];
ub=[0.06 1.5 1];
x0=[3.14 0 0];


controller=SymbolicSet('rip_energy_controller.bdd','projection',[1 2
3]);
target=SymbolicSet('rip_energy_target.bdd');
y=x0;
v=[];

while(1)

  if (target.isElement(y(end,:)))
    break;
  end

  u=controller.getInputs(y(end,:));
  v=[v; u(1,:)];
  [t x]=ode45(@rip_energy_ode,[0 .05], y(end,:), [],u(1,:));

  y=[y; x(end,:)];
end
v
size(v)
y
size(y)
%% plot the rip_energy domain
% colors
colors=get(groot,'DefaultAxesColorOrder');

% load the symbolic set containig the abstract state space
```

```
set=SymbolicSet('rip_energy_ss.bdd','projection',[1]);
hold on

% load the symbolic set containig target set
set=SymbolicSet('rip_energy_target.bdd','projection',[1]);

% plot initial state  and trajectory
Lr=0.216000000000000;

for n=1:2:size(y,1)
    subplot(3,3,(n+1)/2);
    plot([0 -Lr*sin(y(n,1))], [0 Lr*cos(y(n,1))],'k.-')
    hold on
    plot(0,0,'.','color',colors(5,:),'markersize',20)
    axis([-Lr-0.1 Lr+0.1 -Lr-0.1 Lr+0.1])

    title(['t=' num2str((n-1)*0.05)])

    if(n==size(y,1))
        break;
    end

end

end

function dxdt = rip_energy_ode(t, x, u)
g=9.810000000000000;
Jp=0.001200000000000;
Lp=0.337000000000000;
mp=0.127000000000000;

dxdt(1)=x(2);
dxdt(2)=(mp*Lp/(2*Jp))*(u*cos(x(1))-g*sin(x(1)));
dxdt(3)=0.5*mp*Lp*u*x(2)*cos(x(1));
dxdt = dxdt';
end
```