Formulation of the piecewise linear control of an inverted pendulum as a linear complementarity problem

Groen, M.J.

Award date:
1999

TU/e technische universiteit eindhoven

Formulation of the Piecewise Linear Control of an Inverted Pendulum as a Linear Complementarity Problem

Author: M.J. Groen

Master of Science Thesis
Gliwice, March 1999 until September 1999
Supervisors: Prof. Dr. Ir. P.P.J. van den Bosch
A. Polański, M.Sc., Ph.D.

Institutions:
Control Theory Group
Institute of Automatic Control
Silesian Technical University
Gliwice, Poland

Measurement and Control Section
Measurement and Control Systems
Faculty of Electrical Engineering
Eindhoven University of Technology
Eindhoven, The Netherlands

Summary

A popular subject for research in Control Engineering is the inverted pendulum. Strategies to swing-up and balance a pendulum in its inverted position have been described in literature many times. In this report it is shown that the pendulum can be swung up by means of the *energy pumping method*, a method widely used in research. Subsequently, the pendulum is balanced with a linear controller.

The topic of this report is the analysis of the stability of the (controlled and uncontrolled) inverted pendulum with the aid of piecewise linear Lyapunov functions. For this purpose the state space is decomposed into disjunct triangular shaped cells, the *triangulation*. On this triangulation a piecewise affine approximation of the inverted pendulum is calculated. It is shown how a piecewise linear Lyapunov function can be defined upon this triangulation. A method is described to formulate the stability of a piecewise affine approximation of the pendulum as a linear program (LP). This enables the system analyst to present the problem to an LP solver. This solver will produce a piecewise linear Lyapunov function that can be used to examine the stability of a region in the state space.

MatLab functions have been written to transform the differential equations of the pendulum to an LP problem. Additionally, a function has been written to display a 3-dimensional representation of the piecewise linear Lyapunov function found by the LP solver. Numerical examples of a few simple control situations are presented to demonstrate that the LP solver does find results that conform to knowledge about the actual stability of these simple situations. The generation times are acceptable on a desktop computer for triangulations that were relatively fine (0.0125 rad $\times$ 0.0125 rad s^{-1}) and still covered a not too small area (0.375 rad $\times$ 0.375 rad s^{-1} decomposed into 1800 cells). The results are presented in the form of 3-dimensional graphs of the Lyapunov function.

Next the report extends the idea of having a solver look for piecewise linear Lyapunov functions. It is shown that the problem of finding a piecewise linear controller to turn a point into a stable equilibrium point can be formulated as a linear complementarity problem (LCP). These problems can then be presented to a solver for (linear) complementarity problems.

To automate the formulation of these LCP's MatLab functions have been developed to generate the LCP in a form suitable to be submitted to a complementarity solver that is available through the Internet. Simple LCP's generated by this function have been submitted to this solver, but convergence has not yet been obtained within the maximum number of iterations allowed by the solver. The reason for this should be studied further.

Contents

1. Introduction

The inverted pendulum is a classical subject for research on control techniques. A wide range of solutions has been found to swing-up the inverted pendulum and stabilise it in its inverted position. A popular method to do the swing-up is by means of the *energy pumping method*. The controller compares the energy of the pendulum with the energy of a pendulum balanced in the inverted position. If the amount of energy is lower, the controller will feed energy to the system; if it is higher, it will drain energy from the system.

Traditionally the problem of swinging up the inverted pendulum and keeping it in its upright position has been subdivided into two subproblems. Firstly the swing-up of the pendulum, and secondly the problem of stabilising the inverted pendulum in its upright position. Both problems are usually solved independently, what results in a composite controller: one controller to do the swing-up and another to do the stabilising in the inverted position.

An interesting question to ask is, whether it is feasible to use piecewise linear functions to analyse the inverted pendulum and if so, if it is possible to design a swing-up strategy using piecewise affine functions. For the analysis the focus will be on the process of finding piecewise linear Lyapunov functions by numerical calculation. In case of the design process an attempt will be made to find a piecewise linear control law that is able to do a swing-up *and* stabilise the pendulum in its inverted position by making use of the techniques of finding a piecewise linear Lyapunov function.

The numerical calculation of piecewise linear Lyapunov functions has several advantages:

- This method might enable us to find a Lyapunov function that defines a stability region for the linearly controlled pendulum in its inverted position which is larger than a region obtained by more traditional methods like quadratic Lyapunov functions.
- For swing-up strategies we can look for Lyapunov functions that prove the instability of the pending position.

The interest of piecewise linear control is the following:

- All solutions to the swing-up problem of the inverted pendulum mentioned in literature consist of an algorithm that first swings up the pendulum and secondly switches to another method to stabilise the pendulum in the inverted position. The question which rises is, whether it is possible to find a single control law
$$u = u(\theta, \dot{\theta})$$
that does a swing-up and has a stable upward position. It might be possible to find such a control law with piecewise linear control.
- All the energy pumping methods need assumptions about the possibility to ignore friction. Doing the computer design of the control law and the control Lyapunov function enables us to study this problem from another point of view.
- If it turns out, that it is possible to do a swing-up for a relatively coarse triangulation – in other words a *simple* piecewise linear controller exists – then the design may be considered a new practical solution to the swing-up problem of the inverted pendulum.

From the above we arrive at the following goal of the project:

Firstly, develop a method to find a piecewise linear Lyapunov function for the (controlled or uncontrolled) inverted pendulum by means of numerical calculation. Secondly, develop a way to design a piecewise linear controller by making use of this method.

The first part will be done by reformulating the conditions for a proper piecewise linear Lyapunov function as a linear program. This LP can then be solved by an LP-solver. The second part is to be done by restating the conditions for the swing-up and the stability of the inverted position as a *linear complementarity problem* (LCP) in order to submit the problem to a solver for LCP's.

The contents of this report is subdivided in the following way. As an introduction chapter 2 *Inverted pendulum on a cart* describes the object of our case study: the inverted pendulum. The mathematical model of the inverted pendulum is presented. In chapter 3 *Swing-up by the energy method* a simple method is presented to swing up the pendulum. It is shown that this can be done by means of the energy pumping method. With linearisation it is shown that both the pending position and the upright position are unstable. A simple linear controller can stabilise the pendulum in its inverted position. With simulations in Simulink it is shown that this approach should work. In chapter 4 *Piecewise affine approximation of the non-linear system*, a triangulation is defined that decomposes the state space into triangular shaped cells. The mathematical model of the inverted pendulum as presented in chapter 2 is approximated by a piecewise affine function. Next it is shown in chapter 5 *Piecewise linear Lyapunov functions*, that linear programming tools can be used to find piecewise linear Lyapunov functions to show the stability or instability of regions of the state space. Chapter 6 *Design of piecewise linear control* shows how linear complementarity problems can be used to try to find piecewise linear controllers to swing-up the pendulum and stabilise it in the inverted position. In the last chapter of the report conclusions and recommendations are presented.

2. Inverted pendulum on a cart

The inverted pendulum is introduced in this chapter. It is outlined what assumptions have been made. The first section tries to form a picture of the inverted pendulum. The next section presents the mathematical model of the inverted pendulum. Finally, in section three it is mentioned on which part this report will focus.

2.1 The pendulum

The inverted pendulum on a cart consists of a cart that is placed on a rail; this way it can only move in one dimension. On the cart is an axle, that is oriented perpendicular to the direction in which the cart can move, and the horizontal plane. To this axle a rod is attached with a mass at its end (the pendulum), so that it can rotate freely in the vertical plane parallel to the direction of movement of the cart. The natural position of the pendulum will be its pending position. Figure 1 shows the pendulum on a cart and defines some of the variables and parameters.

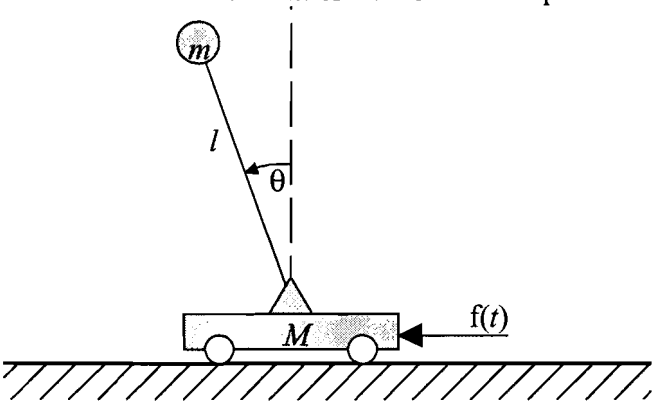


Figure 1: The inverted pendulum on a cart

The objective of the inverted pendulum is to apply a varying force to the cart, so that the horizontal movements of the cart will swing up the pendulum from its pending position to its upright position. From the moment the pendulum has reached its upright position, the pendulum should be balanced in this position by applying appropriate forces to the cart.

The following assumptions are made within the rest of this report, unless specified explicitly different:

- All motion is considered to be frictionless.
- The mass of the pendulum is assumed to be concentrated at the end of the rod.
- The cart position can range from $-\infty$ to $+\infty$.

2.2 The mathematical model

The pendulum on a cart can be modelled by the following system of differential equations:

$$(M+m)\ddot{x} + ml\cos\theta\cdot\ddot{\theta} - ml\sin\theta\cdot\dot{\theta}^2 = f \qquad (1)$$

$$\ddot{x}ml\cos\theta + ml^2\ddot{\theta} + d\dot{\theta} - mgl\sin\theta = 0 \qquad (2)$$

with variables:

f	-	external force applied to the cart	[N]
x	-	position of the cart	[m]
θ	-	angle between the rod of the pendulum and the upright position	[rad]

and parameters:

d	-	damping of the rod rotation	[kg m^2 s^{-2}]
g	-	gravitational acceleration	[m s^{-2}]
l	-	length of the pendulum	[m]
m	-	mass of the pendulum	[kg]
M	-	mass of the cart	[kg]

The friction of the pendulum will be neglected in this report, so the equations can be rewritten, omitting the friction term $d\dot{\theta}$:

$$(M+m)\ddot{x} + ml\cos\theta\cdot\ddot{\theta} - ml\sin\theta\cdot\dot{\theta}^2 = f \qquad (3)$$

$$\ddot{x}ml\cos\theta + ml^2\ddot{\theta} - mgl\sin\theta = 0 \qquad (4)$$

2.3 Cascaded structure

Looking at the second differential equation (4), we see that we can consider this equation to describe the relationship between the cart acceleration and the movement of the pendulum. The first differential equation (3) then expresses the relation between the external force f and the acceleration $\ddot{x}$ of the cart. This approach enables us to regard the pendulum without the cart as an isolated system, with the acceleration $\ddot{x}$ being the control input, and θ and $\dot{\theta}$ being the outputs. This idea with additional controllers is depicted in figure 2.

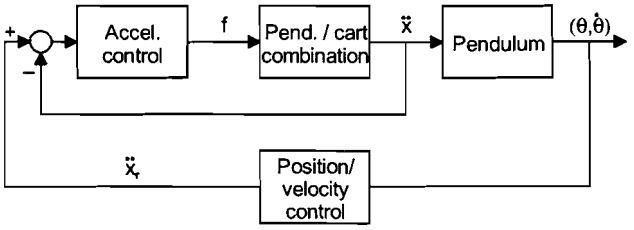


Figure 2: Cascaded structure of the pendulum on a cart

In this report it is assumed that the dynamics of the inner loop from figure 2 is negligible, which means $\ddot{x} \cong \ddot{x}_r$. This means that as far as the mathematical model is concerned we will focus on equation (4).

3. Swing-up by the energy method

A popular method to swing up the inverted pendulum is the energy approach. This approach can be found in the papers [1], [2] and [7]. A controller 'measures' the kinetic and potential energy of the pendulum and either decreases the energy of the pendulum if the total of kinetic and potential energy is more than the energy of an in the inverted position balanced pendulum or increases the pendulum's energy if this total is less. This same approach is followed in this chapter to swing-up the inverted pendulum.

The basic idea of this method is explained in the first section. Furthermore the equation for the control law is derived. In the second section of this chapter, the stability of both the inverted and pending position of the pendulum are analysed by means of linearisation around these two points. These analyses show that both the inverted and pending position of the controlled pendulum are not stable. The controller might be able to swing-up the pendulum, but is not able to stabilise it in the inverted position.

Therefor in the third section a linear controller is introduced, that is able to stabilise the pendulum in its inverted position. The pendulum will be swung up by the first (non-linear) controller and if it comes close enough to the inverted position, control is switched to the linear controller to stabilise the pendulum.

In the fourth section of this chapter this is illustrated with some simulations. These simulations show that the pendulum can be swung up by the energy method, but can not be stabilised in this position. A second simulation shows that the addition of the linear controller enables us to do the complete job of swinging up the pendulum and stabilising it in the inverted position.

3.1 The swing-up

All methods mentioned in literature that do a swing-up and stabilise the inverted pendulum in its upright position, have one thing in common. All approaches subdivide the problem in at least two subproblems: Firstly, the swing-up of the pendulum, and secondly the stabilisation of the pendulum.

At this moment we start with the same idea. We design a controller that does a swing-up and then analyse if this controller is capable of keeping the pendulum in its inverted position. Probably it will not be able to stabilise the pendulum in this position. In that case we will add a second controller to keep the pendulum up.

As mentioned in the previous chapter, we restrict ourselves by only considering the second model equation (4). The second derivative of the position $\ddot{x}$ is considered to be the control variable of our system:

$$u = \ddot{x},$$

which yields the following equation after substitution in equation (4):

$$uml\cos\theta + ml^2\ddot{\theta} - mgl\sin\theta = 0 \qquad (5)$$

The total energy of the pendulum equals the sum of the kinetic energy and the potential energy. If we define the potential energy to be 0 if the pendulum is in one of the horizontal positions ($\theta = \pm\frac{\pi}{2}$), the total energy of the pendulum equals:

$$E = \tfrac{1}{2} m(l\dot{\theta})^2 + mgl\cos\theta \qquad (6)$$

The derivative of the total energy is:

$$\dot{E} = ml^2\dot{\theta}\ddot{\theta} - mgl\dot{\theta}\sin\theta \qquad (7)$$

Equation (5) can be rewritten as:

$$ml^2\ddot{\theta} - mgl\sin\theta = -uml\cos\theta \qquad (8)$$

Substituting (8) into (7) leads to:

$$\dot{E} = -uml\dot{\theta}\cos\theta \qquad (9)$$

Since the change of energy is proportional to the control signal, it is easy to control the energy of the pendulum. The system is however not controllable if either $\dot{\theta} = 0$ or $\theta = \pm\frac{\pi}{2}$, this means when the pendulum is in one of its horizontal positions or when it reverses its velocity.

With equations (6) and (9) we can derive the control law for the swing-up. The energy of the in the upward position balanced pendulum determines whether energy should be supplied to or drawn from the pendulum. This energy equals:

$$E_{up} = mgl \qquad (10)$$

It is quite natural to assume that the control input is bounded:

$$u_{min} \le u(t) \le u_{max}, \qquad (11)$$

where u_{min} and u_{max} are finite.

If $u_{min} = -u_{max}$ and $u_{max} > 0$, the following control should do a swing-up:

$$u(t) = \begin{cases} u_{max}\,\mathrm{sgn}(\dot{\theta}\cos\theta) & \text{if} \quad E(t) > mgl \\ u_{min}\,\mathrm{sgn}(\dot{\theta}\cos\theta) & \text{if} \quad E(t) < mgl, \\ 0 & \text{if} \quad E(t) = mgl \end{cases} \qquad (12)$$

where *sgn*(.) denotes the signum function:

$$\mathrm{sgn}(x) = \begin{cases} -1 & \text{if} \quad x < 0 \\ 0 & \text{if} \quad x = 0 \\ 1 & \text{if} \quad x > 0 \end{cases}$$

In the papers [1] and [2] energy pumping strategies different from (12) were proposed. In [1] energy pumping was realised by the following function:

$$u = \mathrm{sat}_{ng}\big(k(E - E_0)\big)\mathrm{sgn}(\dot{\theta}\cos\theta)$$

where k is a design parameter, the function sat_{ng} denotes a function which saturates at $n\,g$, E is the actual swing energy, and E_0 is the desired swing energy.

The paper [2] employed the following strategy,

$$u = \alpha\dot{\theta}\cos\theta \cdot E$$

with E being the difference between the actual swing energy and the desired swing energy.

3.2 Stability analysis

In this section we will analyse the stability of the controlled system by means of linearisation in the working point. The controlled pendulum will be examined in respectively the inverted position and the pending position.

3.2.1 Linearisation around the inverted position

To analyse the system we linearise equation (5) around the upward position $(\theta,\dot{\theta}) = (0,0)$:

$$\ddot{\theta} - \frac{g}{l}\Delta\theta = -\frac{1}{l}u \qquad (13)$$

In the neighbourhood of the inverted position $(\theta,\dot{\theta}) = (0,0)$ the total energy of the pendulum (6) can be approximated by

$$E(t) \approx \tfrac{1}{2}m(l\Delta\dot{\theta})^2 + mgl\left(1 - \tfrac{1}{2}(\Delta\theta)^2\right) \qquad (14)$$

Substituting this approximation (14) in equation (12) and writing $\Delta\theta$ for θ and $\Delta\dot{\theta}$ for $\dot{\theta}$ leads to

$$u(t) = \begin{cases} u_{\max}\ \mathrm{sgn}(\Delta\dot{\theta}\cos\Delta\theta) & \text{if } \tfrac{1}{2}ml^2(\Delta\dot{\theta})^2 - \tfrac{1}{2}mgl(\Delta\theta)^2 > 0 \\ u_{\min}\ \mathrm{sgn}(\Delta\dot{\theta}\cos\Delta\theta) & \text{if } \tfrac{1}{2}ml^2(\Delta\dot{\theta})^2 - \tfrac{1}{2}mgl(\Delta\theta)^2 < 0 \\ 0 & \text{if } \tfrac{1}{2}ml^2(\Delta\dot{\theta})^2 - \tfrac{1}{2}mgl(\Delta\theta)^2 = 0 \end{cases}$$

or

$$u(t) = \begin{cases} u_{\max}\ \mathrm{sgn}(\Delta\dot{\theta}) & \text{if } (\Delta\dot{\theta})^2 > \frac{g}{l}(\Delta\theta)^2 \\ u_{\min}\ \mathrm{sgn}(\Delta\dot{\theta}) & \text{if } (\Delta\dot{\theta})^2 < \frac{g}{l}(\Delta\theta)^2 \\ 0 & \text{if } (\Delta\dot{\theta})^2 = \frac{g}{l}(\Delta\theta)^2 \end{cases} \qquad (15)$$

If we take a careful look at this equation we can see, that the state space around the origin can be divided into six regions like depicted in figure 3.

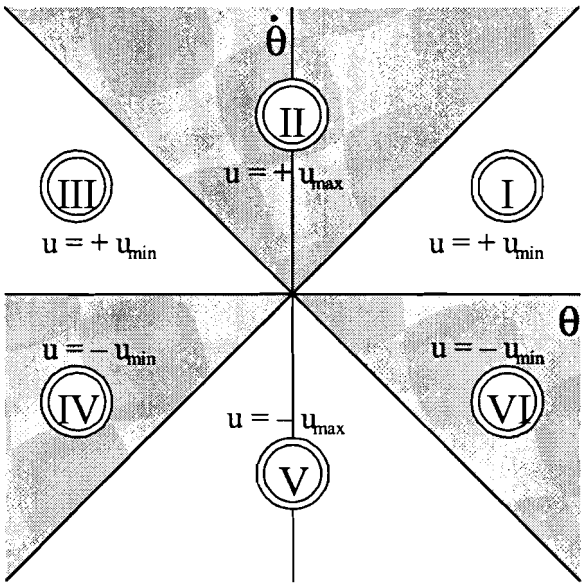


Figure 3: The state space around $(\theta,\dot{\theta}) = (0,0)$ *can be divided into six regions*

The control signal in equation (15) can be substituted into the linearised second model equation (13):

$$\ddot{\theta} - \frac{g}{l}\Delta\theta = -\frac{1}{l}\begin{cases} u_{\max}\,\mathrm{sgn}(\Delta\dot{\theta}) & \text{if} \quad (\Delta\dot{\theta})^2 > \frac{g}{l}(\Delta\theta)^2 \\ u_{\min}\,\mathrm{sgn}(\Delta\dot{\theta}) & \text{if} \quad (\Delta\dot{\theta})^2 < \frac{g}{l}(\Delta\theta)^2 \\ 0 & \text{if} \quad (\Delta\dot{\theta})^2 = \frac{g}{l}(\Delta\theta)^2 \end{cases}$$

or

$$\ddot{\theta} = \begin{cases} \frac{g}{l}\left(\Delta\theta - \frac{u_{\max}}{g}\mathrm{sgn}(\Delta\dot{\theta})\right) & \text{if} \quad (\Delta\dot{\theta})^2 > \frac{g}{l}(\Delta\theta)^2 \\ \frac{g}{l}\left(\Delta\theta - \frac{u_{\min}}{g}\mathrm{sgn}(\Delta\dot{\theta})\right) & \text{if} \quad (\Delta\dot{\theta})^2 < \frac{g}{l}(\Delta\theta)^2 \\ \frac{g}{l}\Delta\theta & \text{if} \quad (\Delta\dot{\theta})^2 = \frac{g}{l}(\Delta\theta)^2 \end{cases} \qquad (16)$$

To derive the expression for the trajectories in region I we take from (16) the differential equation for that region:

$$\ddot{\theta} = \frac{g}{l}\left(\Delta\theta - \frac{u_{\max}}{g}\right)$$

We write the second derivative with differentials:

$$\frac{\mathrm{d}\,\Delta\dot{\theta}}{\mathrm{d}\,\Delta\theta}\cdot\frac{\mathrm{d}\,\Delta\theta}{\mathrm{d}\,t} = \frac{g}{l}\left(\Delta\theta - \frac{u_{\max}}{g}\right),$$

rearranging the differentials:

$$\Delta\dot{\theta}\cdot \mathrm{d}\,\Delta\dot{\theta} = \frac{g}{l}\left(\Delta\theta - \frac{u_{\max}}{g}\right)\mathrm{d}\,\Delta\theta,$$

taking the integral of both sides:

$$\int \Delta\dot{\theta}\ \mathrm{d}\,\Delta\dot{\theta} = \int \frac{g}{l}\left(\Delta\theta - \frac{u_{\max}}{g}\right)\mathrm{d}\,\Delta\theta$$

which equals:

$$\tfrac{1}{2}\Delta\dot{\theta}^2 = \frac{g}{l}\left(\tfrac{1}{2}\Delta\theta^2 - \frac{u_{\max}}{g}\Delta\theta\right) + c_1,$$

which can be rearranged:

$$\Delta\dot{\theta}^2 = \frac{g}{l}\left(\Delta\theta - \frac{u_{\max}}{g}\right)^2 - \frac{{u_{\max}}^2}{lg} + 2c_1,$$

and finally absorbing the constant $-\dfrac{{u_{\max}}^2}{lg}$ in the integration constant:

$$\Delta\dot{\theta}^2 = \frac{g}{l}\left(\Delta\theta - \frac{u_{\max}}{g}\right)^2 + c_I \qquad (17)$$

The same derivation can be done for the other five regions, leading to the following expressions for the trajectories:

Region I and III: $(\Delta\dot{\theta})^2 = \dfrac{g}{l}\left(\Delta\theta - \dfrac{u_{\min}}{g}\right)^2 + c_I$

Region II: $(\Delta\dot{\theta})^2 = \dfrac{g}{l}\left(\Delta\theta - \dfrac{u_{\max}}{g}\right)^2 + c_{II}$

Region IV and VI: $(\Delta\dot{\theta})^2 = \dfrac{g}{l}\left(\Delta\theta + \dfrac{u_{\min}}{g}\right)^2 + c_{IV}$

Region V: $(\Delta\dot{\theta})^2 = \dfrac{g}{l}\left(\Delta\theta + \dfrac{u_{\max}}{g}\right)^2 + c_V$

These equations represent hyperbolae, so the trajectories have a hyperbolic shape. The trajectories can be plotted with MatLab as is done in figure 4. For the parameters the following values were used: $g = 9.81\ \mathrm{m\,s^{-2}}$, $l = 1\ \mathrm{m}$, $u_{\min} = -2\ \mathrm{m\,s^{-2}}$, and $u_{\max} = +2\ \mathrm{m\,s^{-2}}$.

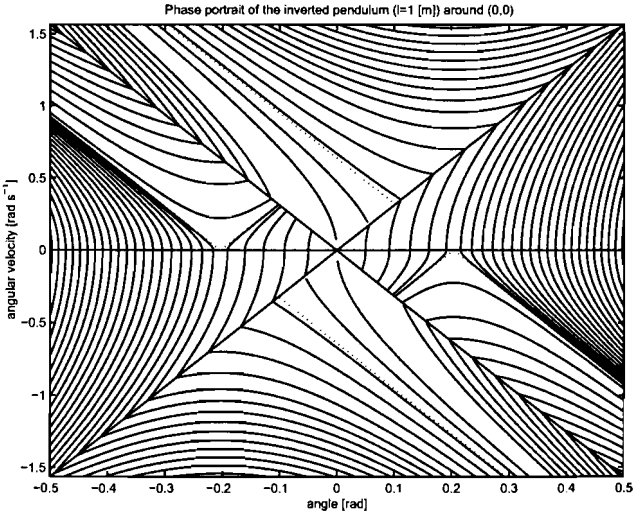


Figure 4: Trajectories around the inverted position

Looking at the trajectories in figure 4, it can be seen that switching between $u_{\min}$ and $u_{\max}$ leads to sliding motions of the system. Sliding motions are solutions with infinitely fast switching of $u(t)$ between $u_{\min}$ and $u_{\max}$. In practice the frequency of the switching is high, but finite. There are two sliding lines in figure 4, intersecting in the equilibrium point in the origin. One extends in the second and fourth quadrant. Sliding along this line is stable and leads to the origin. The other sliding line is within the first and third quadrant. Sliding along this second line leads away form the origin. The conclusion is that the origin is unstable, since arbitrarily close to the origin are starting points of divergent trajectories. In other words: This non-linear controller can not balance the pendulum in its inverted position.

3.2.2 Linearisation around the pending position

Linearising equation (5) around the pending position $(\theta,\dot{\theta}) = (\pi,0)$ we arrive at:

$$\ddot{\theta} + \frac{g}{l}(\Delta\theta - \pi) = \frac{1}{l}u \qquad (18)$$

To get a suitable expression for the control signal u, we approximate the energy of the pendulum (5) with the following expression:

$$E(t) \approx \tfrac{1}{2}m(l\Delta\dot{\theta})^2 - mgl\left(1 - \tfrac{1}{2}(\Delta\theta - \pi)^2\right)$$

which is equivalent to:

$$E(t) \approx \tfrac{1}{2}ml^2(\Delta\dot{\theta})^2 + \tfrac{1}{2}mgl(\Delta\theta - \pi)^2 - mgl$$

Substituting the energy in equation (12) with this approximation and writing $\Delta\theta$ for θ and $\Delta\dot{\theta}$ for $\dot{\theta}$, we get the following expression

$$u(t) = \begin{cases} u_{\max}\,\mathrm{sgn}(\Delta\dot{\theta}\cos\Delta\theta) & \text{if } \tfrac{1}{2}ml^2(\Delta\dot{\theta})^2 + \tfrac{1}{2}mgl(\Delta\theta - \pi)^2 - mgl > mgl \\ u_{\min}\,\mathrm{sgn}(\Delta\dot{\theta}\cos\Delta\theta) & \text{if } \tfrac{1}{2}ml^2(\Delta\dot{\theta})^2 + \tfrac{1}{2}mgl(\Delta\theta - \pi)^2 - mgl < mgl \\ 0 & \text{if } \tfrac{1}{2}ml^2(\Delta\dot{\theta})^2 + \tfrac{1}{2}mgl(\Delta\theta - \pi)^2 - mgl = mgl \end{cases},$$

which can be simplified:

$$u(t) = \begin{cases} u_{max}\,\mathrm{sgn}(\Delta\dot{\theta}\cos\Delta\theta) & \text{if} \quad (\Delta\dot{\theta})^2 > 4\frac{g}{l} - \frac{g}{l}(\Delta\theta - \pi)^2 \\ u_{min}\,\mathrm{sgn}(\Delta\dot{\theta}\cos\Delta\theta) & \text{if} \quad (\Delta\dot{\theta})^2 < 4\frac{g}{l} - \frac{g}{l}(\Delta\theta - \pi)^2 \\ 0 & \text{if} \quad (\Delta\dot{\theta})^2 = 4\frac{g}{l} - \frac{g}{l}(\Delta\theta - \pi)^2 \end{cases}$$

This control system can be substituted into the linearised second model equation (18)

$$\ddot{\theta} + \frac{g}{l}(\Delta\theta - \pi) = \frac{1}{l}\begin{cases} u_{max}\,\mathrm{sgn}(\Delta\dot{\theta}\cos\Delta\theta) & \text{if} \quad (\Delta\dot{\theta})^2 > 4\frac{g}{l} - \frac{g}{l}(\Delta\theta - \pi)^2 \\ u_{min}\,\mathrm{sgn}(\Delta\dot{\theta}\cos\Delta\theta) & \text{if} \quad (\Delta\dot{\theta})^2 < 4\frac{g}{l} - \frac{g}{l}(\Delta\theta - \pi)^2 \\ 0 & \text{if} \quad (\Delta\dot{\theta})^2 = 4\frac{g}{l} - \frac{g}{l}(\Delta\theta - \pi)^2 \end{cases}$$

which is equal to

$$\ddot{\theta} = \begin{cases} -\dfrac{u_{max}\,\mathrm{sgn}(\Delta\dot{\theta}) + g(\Delta\theta - \pi)}{l} & \text{if} \quad \frac{1}{4}\frac{l}{g}(\Delta\dot{\theta})^2 + \frac{1}{4}(\Delta\theta - \pi)^2 > 1 \\ -\dfrac{u_{min}\,\mathrm{sgn}(\Delta\dot{\theta}) + g(\Delta\theta - \pi)}{l} & \text{if} \quad \frac{1}{4}\frac{l}{g}(\Delta\dot{\theta})^2 + \frac{1}{4}(\Delta\theta - \pi)^2 < 1 \\ -\dfrac{g}{l}(\Delta\theta - \pi) & \text{if} \quad \frac{1}{4}\frac{l}{g}(\Delta\dot{\theta})^2 + \frac{1}{4}(\Delta\theta - \pi)^2 = 1 \end{cases} \qquad (19)$$

The third condition in this equation defines an ellipse in the state space and the first and second condition describe respectively the space around the ellipse and the space inside the ellipse. The state space can be divided into four regions according to these conditions as is shown in figure 5.

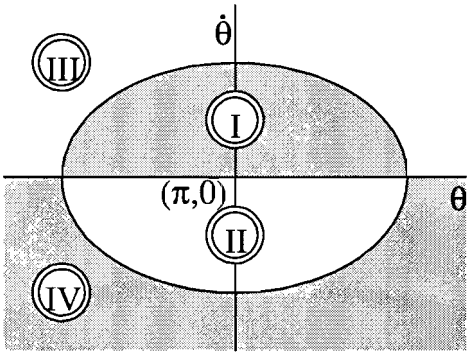


Figure 5: The state space around $(\theta,\dot{\theta}) = (\pi,0)$ *can be divided into four regions*

The trajectories in the regions I till IV can be derived, in a similar manner as the trajectories for the linearisation around the inverted position of the pendulum – see the derivation of (17) for an example. These trajectories can be described by the following equations:

$$\text{Region I:} \quad (\Delta\dot{\theta})^2 = -\frac{g}{l}\left(\Delta\theta + \frac{u_{\min}}{g} - \pi\right)^2 + c_I$$

$$\text{Region II:} \quad (\Delta\dot{\theta})^2 = -\frac{g}{l}\left(\Delta\theta - \frac{u_{\min}}{g} - \pi\right)^2 + c_{II}$$

$$\text{Region III:} \quad (\Delta\dot{\theta})^2 = -\frac{g}{l}\left(\Delta\theta + \frac{u_{\max}}{g} - \pi\right)^2 + c_{III}$$

$$\text{Region IV:} \quad (\Delta\dot{\theta})^2 = -\frac{g}{l}\left(\Delta\theta - \frac{u_{\max}}{g} - \pi\right)^2 + c_{IV}$$

These equations represent ellipses, so the trajectories have an elliptic shape. Figure 6 shows the plot of the trajectories. For the parameters the following values were used: $g = 9.81$ m s^{-2}, $l = 1$ m, $u_{\min} = -2$ m s^{-2}, and $u_{\max} = +2$ m s^{-2}.

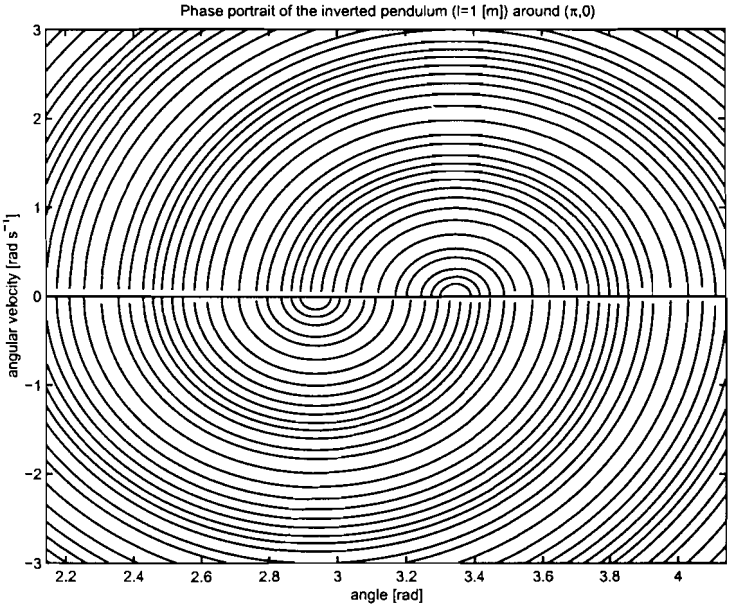


Figure 6: Trajectories around the pending position

Looking at figure 6, we can see that the phase portrait is formed by spirally shaped trajectories leading away from the origin. The part of the horizontal axis between (and including) the centres of the semi-ellipses consists of singular points. From the equations of the trajectories we can derive the position of the two centres of the semi-ellipses. The centre of the semi-ellipses in the upper half of the state space is in the point:

$$(\theta, \dot{\theta}) = (\pi - \frac{u_{\min}}{g}, 0)$$

and the centre of the semi-ellipses in the lower half of the state space in the point:

$$(\theta, \dot{\theta}) = (\pi + \frac{u_{\min}}{g}, 0)$$

3.3 Stabilising the pendulum

Thus far, we have shown that we can do a swing-up. However, the inverted position is not stable. To keep the pendulum in its inverted position after it has been swung up, we switch to a linear controller if $(\theta, \dot{\theta})$ is close enough to (0,0). The linear controller is modelled by the following equation:

$$u = a_1 \Delta\theta + a_2 \Delta\dot{\theta} \tag{20}$$

If we substitute this control input u into the linearised second model equation (13), we get the following differential equation:

$$\Delta\ddot{\theta} - \frac{g}{l}\Delta\theta = -\frac{1}{l}(a_1 \Delta\theta + a_2 \Delta\dot{\theta})$$

which can be simplified to:

$$\Delta\ddot{\theta} + \frac{a_2}{l}\Delta\dot{\theta} + \frac{a_1 - g}{l}\Delta\theta = 0 \tag{21}$$

According to the Hurwitz stability criterion, this system is stable if the following two conditions are met:

$$\frac{a_1 - g}{l} > 0 \quad \text{and} \quad \frac{a_2}{l} > 0$$

which leads to the following two conditions if we assume that the pendulum length l is positive:

$$a_1 > g \quad \text{and} \quad a_2 > 0 \tag{22}$$

3.4 Numerical examples

The model of the pendulum has been defined in Simulink and some simulations have been done. Appendix A shows the Simulink files that have been used. Table 1 contains the settings of the parameters used in the Simulink simulations.

In the first place the non-linearly controlled pendulum has been simulated. In figure 7 the angle θ of the pendulum is shown as it changes in time.

Table 1: Parameters for examples

Parameter	Value	Dimension
g	9.81	$\mathrm{m\,s^{-2}}$
l	1	m
$u_{\min}$	−2	$\mathrm{m\,s^{-2}}$
$u_{\max}$	2	$\mathrm{m\,s^{-2}}$
a_1	15	$\mathrm{m\,s^{-2}}$
a_2	3	$\mathrm{m\,s^{-1}}$

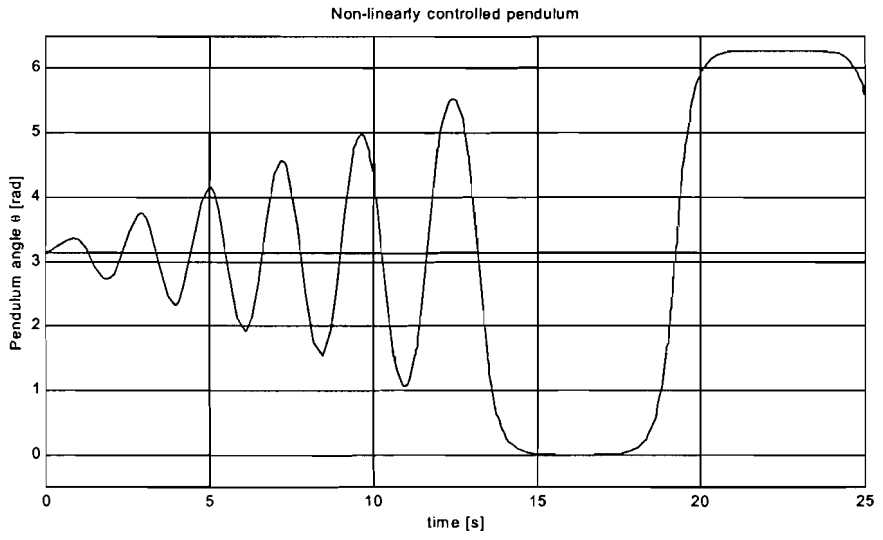


Figure 7: Non-linearly controlled pendulum

As can be seen the non-linear controller does swing up the pendulum. After approximately 15 seconds the pendulum reaches the inverted position. The non-linear controller is not able to maintain the pendulum in the inverted position though. After a few seconds the pendulum falls over and the non-linear controller directs it again to the inverted position.

To stabilise the pendulum in its inverted position, a linear controller was added. This results in a composite controller consisting of the non-linear controller to do the swing-up and a linear controller to stabilise the pendulum in its inverted position. The non-linear controller is in effect when the following condition is valid:

$$\theta^2 + \dot{\theta}^2 \geq r, \tag{23}$$

with the radius r set to 1.

If the pendulum comes close enough to the inverted position in the state-space the above condition (23) is no longer valid:

$$\theta^2 + \dot{\theta}^2 < r,$$

and the linear controller takes over from the non-linear controller.

Figure 8 displays the result of a Simulink simulation with the additional linear controller. It shows that the linear controller is able to keep the pendulum in its inverted position.

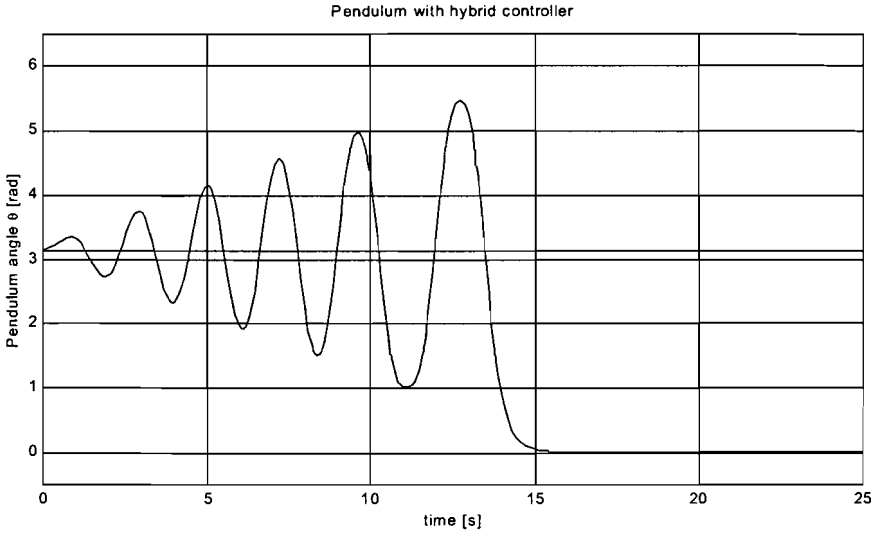


Figure 8: Pendulum with hybrid controller

After implementing the energy pumping method (12) in Simulink, it turned out that the simulation program can not deal with the sliding solutions as shown in figure 4. In order to get around this problem, the signum function sgn(.) was replaced by a saturation with high gain (gain $k = 50$). The block diagrams of this Simulink simulation are given in Appendix A.

4. Piecewise affine approximation of the non-linear system

The majority of non-linear systems is hard to analyse, because of the non-linear behaviour. A common way to be able to analyse such systems is by linearising the system in a working point. The disadvantage of this approach is that the results from this analysis are only valid in a small neighbourhood around this working point. An alternative to linearisation around a working point is to calculate a piecewise affine approximation of the system, which lacks the disadvantage of being valid in a small region.

This chapter explains how a piecewise affine approximation of the system is found. The first section of this chapter shows how the state space is decomposed into disjunct cells. The second section shows how a linear function is defined in each cell to form together the piecewise affine approximation. Next it is shown how the MatLab code finds the piecewise affine approximation. The last section gives some statistics about the order of the errors made when using the piecewise affine approximation instead of the non-linear model.

4.1 Introduction to space state decomposition

The state space is divided into separate regions, which will be called *cells*. The cells will be of triangular shape, so this decomposition will be named the *triangulation* of the state space.

Since the state space is periodic in the direction of pendulum angle θ, the triangulation will be periodic too, with the same period 2π [rad]. Formally put:

$$\left(\theta,\dot{\theta}\right) \in \mathrm{C}(m) \Leftrightarrow \left(\theta + 2\pi,\dot{\theta}\right) \in \mathrm{C}(m),$$

where $\left(\theta,\dot{\theta}\right) \in \mathrm{C}(m)$ denotes that point $\left(\theta,\dot{\theta}\right)$ lies within cell m.

For practical reasons it is assumed, that the state space is limited in the direction of the angular velocity $\dot{\theta}$:

$$-\omega_{\mathrm{lim}} \le \dot{\theta} \le +\omega_{\mathrm{lim}} \qquad (24)$$

with $\omega_{\mathrm{lim}} \in \mathbf{R}^{+}$ and finite. Then the state space is finite and the total number of cells M will be finite too.

Each cell in the triangulation will have a unique identifying number m. The cells in the triangulation are determined by their vertices. Since the cells have a triangular shape, the number of vertices that determine a cell is three. The points in the state space that correspond with a vertex of a cell from the triangulation, will be called *nodes* [1] and they will all have a unique identifying number k. The total number of nodes in the triangulation will be K, which is finite under assumption (24).

The triangulation is completely defined, if the co-ordinates of the nodes are known and when it is known which nodes are vertices of which cells.

[1] In this report the term *node* will be used when referring to the triangulation and the term *vertex* will be used when referring to the intersection of the edges of the cells. However both terms can be used interchangeably.

The co-ordinates of node k are denoted with the column vector x_k:

$$x_k = \begin{bmatrix} \theta_k & \dot{\theta}_k \end{bmatrix}^T \tag{25}$$

The co-ordinates of the K nodes of the triangulation are arranged into a matrix X:

$$X = [x_0 \quad x_1 \quad x_2 \quad \ldots \quad x_K] \tag{26}$$

Two functions are defined to give the relationship between nodes and cells:

1. $\mu(k)$ returns a list containing the identifying numbers m of the cells of which k is a vertex.
2. $\kappa(m)$ returns a list containing the three identifying numbers k of the nodes that are vertices of cell m.

In this report the following assumptions are made considering the triangulation: One node is supposed to reside in the origin of the state space. To be able to conveniently examine the neighbourhood around the pending position, another node is supposed to coincide with the pending position $(\theta,\dot{\theta}) = (\pi,0)$. Due to the periodicity of the state space, a copy of the first node (the one coinciding with the origin) is supposed to be in the position $(\theta,\dot{\theta}) = (2\pi,0)$. These two nodes are considered to be equivalent. All the nodes in the triangulation are supposed to be evenly spaced in both the horizontal and vertical direction. In that case the number of nodes in the horizontal direction will be even. The number of nodes in the vertical direction will be odd. (One node on the horizontal axis and the same number of nodes in both the upper and the lower plane.)

4.1.1 Triangulation coarseness

The coarseness of the triangulation is defined by two parameters:

1. the horizontal diameter D_h, which is determined by the number of nodes in the horizontal direction N_h:

$$D_h = \frac{2\pi}{N_h}$$

2. the vertical diameter D_v, which is determined by the number of nodes in the vertical direction N_v:

$$D_v = \frac{2\omega_{\text{lim}}}{N_v - 1}$$

4.1.2 Global triangulation

There are several ways of dividing the state space into triangular shaped cells and there are different ways of assigning identifying numbers to the cells and nodes. An arbitrary triangulation is shown in figure 9 with a numbering of the cells and nodes. This particular numbering system will be referred to as the global numbering system. The origin of the co-ordinate system coincides with node number 0 in the middle of the leftmost column. Note that the state space is periodic in the horizontal direction, so that the triangulation will be periodic too. This is depicted in figure 9 by a shaded copy of the first column of nodes at the right hand side of the triangulation. Node number 0 for example appears twice in the figure; namely at the left side of the triangulation where the origin resides, and at the right side at co-ordinates $(2\pi,0)$. In the vertical direction the triangulation can extend towards (plus/ minus) infinity, but as stated before the triangulation is supposed to be bounded in this direction.

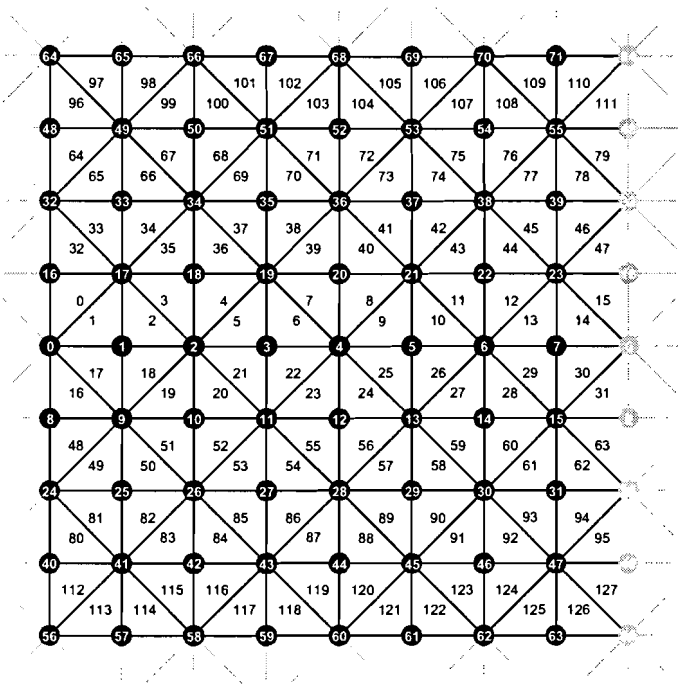


Figure 9: Layout of the triangulation with global numbering system

4.1.3 Partial triangulation

During the research some disadvantages of the numbering system of the triangulation in the previous paragraph became apparent. Two of these disadvantages are:

1. The previously introduced numbering system applies to the complete state space (which in itself is of course an advantage). However, often just a part of the state space is examined. If we want to let our MatLab code run through all nodes within the region to be examined, the sequence of node numbers makes a jump periodically.
2. During examination of the neighbourhood of the origin, it is necessary to take into account the behaviour of points with negative values for θ, which means that a number of columns on the left side of the triangulated space are being examined and a number of columns on the right side, which introduces extra jumps in the numbering.

To get around these problems a second numbering system was introduced to be used concurrently. During calculations cell and node numbers are being mapped from one numbering system to the other and vice versa.

The alternative numbering system is shown in figure 10. The equilibrium point – which can be the origin of the state space, but might as well be the pending position – is in the centre and will have the number 0. From node number 0 we start numbering to the right. Then we continue on the left of node 0 numbering from left to right. The same is done, alternating between the lower and the upper plane, for the next row under this first row, and then the next row above the first row, and so on. Cells are numbered according to the same pattern as the nodes.

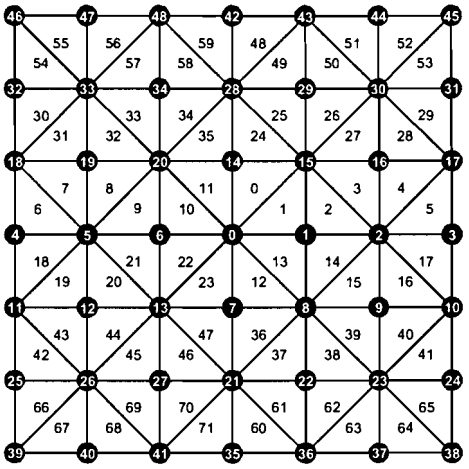


Figure 10: The partial numbering system

When node number 0 of the partial numbering system coincides with the origin, the nodes in the columns to the left of node 0 correspond with nodes at the far right end of the global numbering system. These cells and nodes are being mapped from the right side in the global numbering system to the left side in the partial numbering system. Co-ordinates of nodes that are at the right side in the global numbering system have a θ co-ordinate being close to (but less than) 2π. These co-ordinates are being mapped to the area to the left of the origin, by subtracting 2π from their θ co-ordinate.

The only real advantage of this numbering system is that the jumps in the number sequence have disappeared. The mapping between the two numbering systems back and forth is quite awkward though and especially the transformation of co-ordinates from the area close to 2π to negative values is susceptible to coding errors.

4.2 Piecewise affine approximation of the non-linear system

The idea of a piecewise affine approximation of a non-linear system is illustrated in figure 11, figure 12, and figure 13. Figure 11 shows a non-linear surface, which might represent one model equation of a second order system with states x and y. In figure 12 this same surface is shown, but now with the piecewise affine approximation shown transparently together with it. In this example this approximation consists of eight triangular shaped flat planes. The vertices of these triangular planes coincide with the smooth surface. The other points of the planes usually do not coincide with the smooth surface. Finally, figure 13 shows the piecewise affine approximation again, now without the smooth surface and without being transparent. The eight planes can now each be described by a of linear (differential) equation.

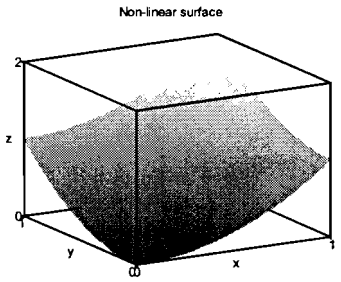


Figure 11: Non-linear surface

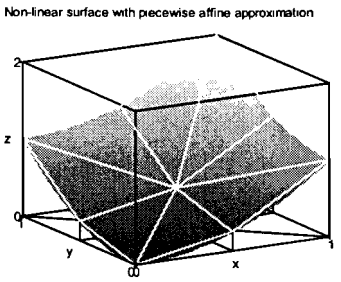


Figure 12: Non-linear surface with piecewise affine approximation

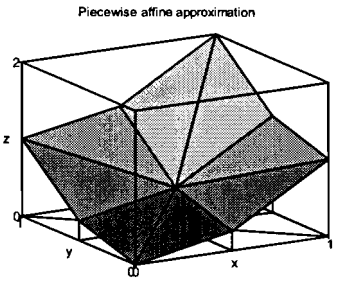


Figure 13: Piecewise affine approximation

The non-linear system is going to be described by a set of affine equations, each of them valid in one cell of the triangulation. The equation of our inverted pendulum can be written in the following form:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = f(x_1, x_2) + g(x_1, x_2)u,$$

or

$$x = f(x) + g(x)u, \tag{27}$$

where $x_1 = \theta$, $x_2 = \dot{\theta}$, and $x = \begin{bmatrix} \theta & \dot{\theta} \end{bmatrix}^T$. The functions $f(.)$ and $g(.)$ follow from the model equations. Writing the second model equation (4) in state space notation:

$$\dot{x} = \begin{bmatrix} x_2 \\ \frac{g}{l}\sin x_1 - \frac{1}{l}u\cos x_1 \end{bmatrix}, \tag{28}$$

we can see that

$$\begin{aligned} f_1(x) &= x_2 & g_1(x) &= 0 \\ f_2(x) &= \tfrac{g}{l}\sin x_1 & g_2(x) &= -\tfrac{1}{l}\cos x_1 \end{aligned} \tag{29}$$

In each cell m we want to approximate the second model equation with a piecewise affine approximation:

$$\dot{x} = a_m + A_m x + b_m u \tag{30}$$

The next section describes how the parameters of the approximation can be found.

4.3 Finding the piecewise affine approximation

For every cell m we have to find a piecewise affine approximation like equation (30). The parameter b_m will be estimated by taking the mean value of the values of $g_2(x)$ from equation (29) in the three vertices of cell m.

The parameters a_m and A_m have respectively two and four elements, totalling six unknown parameters. By setting the value of (30) in the three vertices of a cell m equal to the vector $[f_1(x)\ \ f_2(x)]^T$, where $f_1(x)$ and $f_2(x)$ as in (29), a set of six linear equations with six unknowns is formed:

$$\begin{cases} a_1 + A_{1,*}\, x_{*,1} = f_1(x_{*,1}) \\ a_2 + A_{2,*}\, x_{*,1} = f_2(x_{*,1}) \\ a_1 + A_{1,*}\, x_{*,2} = f_1(x_{*,2}) \\ a_2 + A_{2,*}\, x_{*,2} = f_2(x_{*,2}) \\ a_1 + A_{1,*}\, x_{*,3} = f_1(x_{*,3}) \\ a_2 + A_{2,*}\, x_{*,3} = f_2(x_{*,3}) \end{cases} \tag{31}$$

where $x = \begin{bmatrix} x_{k_1} & x_{k_2} & x_{k_3} \end{bmatrix}$ with x_{k_n} being the column vector with the co-ordinates of the n-th vertex of the cell under consideration.

Equation (31) can be rewritten in matrix notation:

$$\begin{bmatrix} 1 & 0 & x_{1,1} & x_{2,1} & 0 & 0 \\ 0 & 1 & 0 & 0 & x_{1,1} & x_{2,1} \\ 1 & 0 & x_{1,2} & x_{2,2} & 0 & 0 \\ 0 & 1 & 0 & 0 & x_{1,2} & x_{2,2} \\ 1 & 0 & x_{1,3} & x_{2,3} & 0 & 0 \\ 0 & 1 & 0 & 0 & x_{1,3} & x_{2,3} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ A_{1,1} \\ A_{1,2} \\ A_{2,1} \\ A_{2,2} \end{bmatrix} = \begin{bmatrix} f_1(x_{*,1}) \\ f_2(x_{*,1}) \\ f_1(x_{*,2}) \\ f_2(x_{*,2}) \\ f_1(x_{*,3}) \\ f_2(x_{*,3}) \end{bmatrix} \tag{32}$$

This is implemented in the MatLab function `fitcell`. The code and a brief description of this function can be found in appendix B. The function accepts the number of the cell as input and calculates a_m, A_m and b_m.

4.4 Quality of the piecewise affine approximation

The function `testfit` was written to calculate the deflection of the piecewise affine approximation of the system compared to the differential equations that describes our process. This function was applied to the triangulation used for the numerical examples in chapter 5 *Piecewise linear Lyapunov functions*, this means a 15 × 15 and a 31 × 31 triangulation with a horizontal and vertical node spacing of 0.0125 × 0.0125 [rad × rad s^{-1}]. The function was only applied to the inverted position of the pendulum, because it does not work properly yet in the neighbourhood of other points. All three control situations (uncontrolled, non-linear, and linear) were examined.

The errors of the first differential equation are not mentioned here, because they are all zero due to the linearity of this differential equation. The deflections in the approximation of the second differential equation are shown in table 2. It shows the mean value of the absolute values of the relative errors [2], the variance of these values, the maximum value, and the number of situations where the relative error is undefined [3]. For these figures a distinction is made between points that are nodes and points that are randomly chosen within the cells. In every cell all three nodes are examined and an additional 50 points, randomly chosen within the cell. Since the 15 × 15 triangulation has 392 cells, the number of samples for the nodes is 1 176 and for the random points it is 19 600. For the 31 × 31 triangulation these numbers are 5 400 and 90 000

[2] The reference of the relative errors are the values of the non-linear differential equations.

[3] The relative error is undefined if the denominator of the division equals zero. This means the non-linear differential equation is zero valued.

respectively. Finally, the table mentions the number of points where the piecewise affine approximation has a wrong sign.

Table 2: Relative errors of the piecewise affine approximation of the inverted pendulum compared to the value of the second differential equation

	Nodes				Random points (50)				All
	μ (%)	s^2 (%)	max (%)	und (#)	μ (%)	s^2 (%)	max (%)	und (#)	sign (#)
uncontrolled									
15 × 15	0.00	0.00	0.00	0	0.00	0.00	0.00	0	0
31 × 31	0.00	0.00	0.00	0	0.00	0.00	0.00	0	0
non-linearly									
15 × 15	0.03	0.00	0.12	0	0.01	0.00	2.99	0	0
31 × 31	0.12	0.00	2.25	0	0.04	0.00	16.95	0	0
linear									
15 × 15	0.11	0.00	3.74	0	0.19	0.01	7.68	0	0
31 × 31	0.32	0.00	10.09	0	0.43	0.29	40.29	0	4

μ = mean of the absolute value of the relative errors
s^2 = variance of the absolute value of the relative errors
max = maximum of the absolute value of the relative errors
und = number of cases where the relative error is undefined
sign = the number of values with a wrong sign

For the uncontrolled pendulum, the errors are all zero. In the nodes they should be zero, because the fit is always exact in the nodes. In the random points errors might occur, but the table shows that they are very small. For the non-linearly controlled pendulum the errors in the nodes are no longer zero. This is caused by the fact that the parameter b_m in the piecewise affine approximation is the average of the value for the three nodes. Therefor the approximation can not be exact anymore when the control function depends on the state of the system, (which is always the case in practical situations). The mean value is very small, so is the variance. The maximum error is not that small though. This means that it should be taken into account that in rare occasions the error can be quite large. The results for the linearly controlled pendulum confirm this. It is however strange that the errors for the linearly controlled pendulum are larger than for the non-linearly controlled pendulum. Intuition tells the opposite. This fact needs more study.

4.5 Conclusions

In this chapter it was demonstrated how a piecewise affine approximation of the pendulum can be defined and how the parameters of the approximation can be calculated. A MatLab function has been written to calculate these piecewise affine approximations. Additionally, a MatLab function has been written to test the quality of the approximation.

For a triangulation with a node spacing of 0.0125 rad in the horizontal direction and 0.0125 rad s^{-1} in the vertical direction, the deflections of the piecewise affine approximation compared to the actual system are not large (tenths of a percent) when considering just the average values and the spread of the errors. However, the maximum values of these errors are sometimes not particularly small (several tens percent). Therefor one should take into account the fact that occasionally the deflections can be large, but generally they are small. A second remarkable fact is that the errors for a non-linearly controlled pendulum are smaller than the errors for a linearly controlled pendulum. This contradicts with our intuition. The quality of the fit has been discussed only briefly in this report and certainly needs more attention.

The two different numbering systems for identifying cells and nodes in the triangulation are quite awkward and make the writing of new MatLab functions and the adaptation of existing MatLab functions susceptible to errors. If the majority of the MatLab code is to be rewritten in future a more practical alternative for these numbering systems will be a large improvement.

5. Piecewise linear Lyapunov functions

In the previous chapter it was explained how the state space can be decomposed and how a piecewise affine approximation can be defined on this *triangulated* state space. The next thing to do is to define a Lyapunov function on the triangulation. In this chapter it will be shown that it is possible to restate the conditions for a Lyapunov function as a linear program (LP). This method enables us to easily find Lyapunov functions with the aid of an LP solver and proof either stability or instability of an equilibrium point. Recently Julián et. al. [4] have published a similar approach.

In the first section of this chapter it is shown how a Lyapunov function is defined on our triangulated state space. To determine if a Lyapunov function is *proper*, it is necessary to be able to calculate the derivative of the Lyapunov function along the solutions of the differential equations. Section two is concerned with the calculation of this derivative. The conditions that need to be satisfied to ensure that a Lyapunov function is proper and proves the stability (or instability) of an equilibrium point, are formulated as a set of linear inequalities in section three. Next this set of linear inequalities is transformed to suit the needs of an LP solver, in this case PCx. This is done in section four. Section five describes, what MatLab functions were written to read the results from PCx into MatLab and present them. The sixth section shows four examples of Lyapunov functions found this way to show that an equilibrium point is either stable or unstable. In section seven a table with times is presented to show how long it takes to generate the LP problem, to solve the LP problem and to plot the Lyapunov function found. Section eight gives some concluding remarks.

5.1 Calculation of the value of the Lyapunov function

The Lyapunov function in our triangulated system is defined by specifying its values at the nodes and (linearly) interpolating within the cells. In this section we will derive a formula for the value of the Lyapunov function at any point in the state space that is within the triangulated area.

For a cell m the co-ordinates of its vertices can be found in the following way:

$$X(m) = [X]_{\kappa(m)}, \tag{33}$$

where X is the vector holding the co-ordinates of the nodes of the triangulation as defined in (26), and $[X]_{\kappa(m)}$ denotes that $X(m)$ is constructed by picking from vector X the elements specified by the index numbers returned by function κ; this means picking the co-ordinates of the vertices of cell m.

We can specify the position of a point x in a cell m by means of its triangular co-ordinates[4]:

$$x = X(m)\,\beta \tag{34}$$

where $\beta \in \mathbf{R}^3$, and $\beta_1, \beta_2, \beta_3 \in [0,1]$, and

$$\mathbf{1}^T \beta = 1, \tag{35}$$

with $\mathbf{1}$ being the vector $[1\ \ 1\ \ 1]^T$.

Conditions (34) and (35) can be jointly written as follows, introducing the new matrix $\overline{X}(m)$:

[4] Sometimes triangular co-ordinates are referred to as Barrycentric co-ordinates

$$\begin{bmatrix} x \\ 1 \end{bmatrix} = \begin{bmatrix} X(m) \\ \mathbf{1}^{\mathrm{T}} \end{bmatrix} \beta = \bar{X}(m)\beta \tag{36}$$

The Lyapunov function $V(\theta, \dot{\theta}) = V(x)$ is defined by specifying its values at the nodes. We define the value at node number k by v_k, so the Lyapunov function is completely defined by the column vector V:

$$V^{\mathrm{T}} = [v_0 \quad v_1 \quad v_2 \quad \ldots \quad v_K] \tag{37}$$

The first node is supposed to be the equilibrium point in the area under examination.

Now, we can find the value of the Lyapunov function in point x lying in cell m by using the triangular co-ordinates β to linearly interpolate the values of the Lyapunov function at the vertices of cell m:

$$V(x) = [V^{\mathrm{T}}]_{\kappa(m)}\, \beta, \tag{38}$$

where the meaning of the notation $[V^{\mathrm{T}}]_{\kappa(m)}$ is the same as in equation (33).

Now β in (38) can be substituted with the result from (36):

$$V(x) = \left[V^{\mathrm{T}}\right]_{\kappa(m)} \bar{X}^{-1}(m) \begin{bmatrix} x \\ 1 \end{bmatrix} \tag{39}$$

This formula expresses the relationship between the value of the Lyapunov function of a point x laying in cell m and the value of the Lyapunov function in the vertices of m. Now, it is possible to define a (piecewise linear) Lyapunov function by assigning values in all the nodes of the triangulation. In all other points (within the triangulated area) the value of the Lyapunov function is obtained by formula (39).

5.2 Derivative of the Lyapunov function along trajectories

To check whether a certain function is a proper Lyapunov function, it is necessary to know the derivative of the Lyapunov function. Since our piecewise linear Lyapunov function is *not* continuously differentiable, we need an appropriate extension of the Lyapunov theorem. Rouche et. al. [8] describe such an extension, which is obtained by replacing the usual directional derivative by the right upper Dini derivative along the solutions of the set of differential equations. The right upper Dini derivative is defined as follows:

$$\mathrm{D}^{+} f(t) = \limsup_{\Delta t \downarrow 0} \frac{f(t + \Delta t) - f(t)}{\Delta t} \tag{40}$$

According to Yoshizawa [9] the right upper Dini derivative of a function $V\big(t, x(t)\big)$ along the solutions of a differential equation $f\big(t, x(t)\big)$ is:

$$\mathrm{D}^{+} V(t) = \limsup_{\Delta t \downarrow 0} \frac{V\big(t + \Delta t, x(t) + \Delta t\, f\big(t, x(t)\big)\big) - V\big(t, x(t)\big)}{\Delta t} \tag{41}$$

In the case of the piecewise linear Lyapunov functions lim sup = lim (which means: the right upper derivative equals the right derivative) and the Lyapunov function $V\big(t, x(t)\big)$ does not directly depend on the time, so $V\big(t, x(t)\big) = V\big(x(t)\big)$.

For the uncontrolled situation, the right derivative of $V(x)$ along (30) inside cell m will be denoted by $D_m^+(x)$ and equals:

$$\begin{aligned} D_m^+(x) &= \lim_{\Delta t \downarrow 0} \frac{V\big(x + \Delta t(a_m + A_m x)\big) - V\big(x(t)\big)}{\Delta t} \\ &= \lim_{\Delta t \downarrow 0} \frac{\big[V^{\mathrm{T}}\big]_{\kappa(m)} \overline{X}^{-1}(m) \begin{bmatrix} x + \Delta t(a_m + A_m x) \\ 1 \end{bmatrix} - \big[V^{\mathrm{T}}\big]_{\kappa(m)} \overline{X}^{-1}(m) \begin{bmatrix} x \\ 1 \end{bmatrix}}{\Delta t} \\ &= \lim_{\Delta t \downarrow 0} \frac{\big[V^{\mathrm{T}}\big]_{\kappa(m)} \overline{X}^{-1}(m) \begin{bmatrix} \Delta t(a_m + A_m x) \\ 0 \end{bmatrix}}{\Delta t} \\ &= \big[V^{\mathrm{T}}\big]_{\kappa(m)} \overline{X}^{-1}(m) \begin{bmatrix} a_m + A_m x \\ 0 \end{bmatrix} \end{aligned} \tag{42}$$

It is also possible to derive the extension of formula (42) for the case where $u = u(x)$ is a piecewise linear control. This will be used in section 6.1 *Piecewise linear control* on page 34.

$$\begin{aligned} D_m^+\big(x,u(x)\big) &= \lim_{\Delta t \downarrow 0} \frac{V\big(x + \Delta t\big(a_m + A_m x + b_m u(x)\big)\big) - V\big(x(t)\big)}{\Delta t} \\ &= \lim_{\Delta t \downarrow 0} \frac{\big[V^{\mathrm{T}}\big]_{\kappa(m)} \overline{X}^{-1}(m) \begin{bmatrix} x + \Delta t\big(a_m + A_m x + b_m u(x)\big) \\ 1 \end{bmatrix} - \big[V^{\mathrm{T}}\big]_{\kappa(m)} \overline{X}^{-1}(m) \begin{bmatrix} x \\ 1 \end{bmatrix}}{\Delta t} \\ &= \lim_{\Delta t \downarrow 0} \frac{\big[V^{\mathrm{T}}\big]_{\kappa(m)} \overline{X}^{-1}(m) \begin{bmatrix} \Delta t\big(a_m + A_m x + b_m u(x)\big) \\ 0 \end{bmatrix}}{\Delta t} \\ &= \big[V^{\mathrm{T}}\big]_{\kappa(m)} \overline{X}^{-1}(m) \begin{bmatrix} a_m + A_m x + b_m u(x) \\ 0 \end{bmatrix} \end{aligned} \tag{43}$$

5.3 Set of linear inequalities

If the controller is a linear controller or a piecewise linear controller, then the derivative $D_m^+(x,u)$ is a linear form over the co-ordinates x, so the following statement is true:

$$\forall_{k \in \kappa(m)}\ D_m^+(x_k,u_k) < 0 \quad \Rightarrow \quad \forall_{x \in C(m)}\ D_m^+\big(x,u(x)\big) < 0$$

So to ensure that the derivative along solutions is negative for all x in $C(m)$, the Dini-derivative should be less than zero in all three vertices. So for all vertices of each cell m the following should be true:

$$\begin{bmatrix} a_m^{\mathrm{T}} + x^{\mathrm{T}} A_m^{\mathrm{T}} + b_m^{\mathrm{T}} u_k^{\mathrm{T}} & 0 \end{bmatrix} \big(\overline{X}^{-1}(m)\big)^T [V]_{\kappa(m)} \le 0, \tag{44}$$

where the equality only holds if the node is the equilibrium point.

All the inequalities (44) for all cells can be collected into one big matrix yielding the following system of linear inequalities:

$$CV \le 0, \tag{45}$$

with the additional constraint $V \ge 0$ and again the equalities only holding if the actual node is the equilibrium point. Each row in (45) corresponds to an inequality of the form (44). Since every

cell has three vertices and there is an inequality for every vertex in every call, matrix C will have $3M$ rows, where M is the total number of cells in the triangulation.

The linear inequality in (45) will be presented to a solver for linear programs (LP). The value of the Lyapunov function in the equilibrium point v_0 will be set to zero for convenience. This means that the first column of matrix C can be cancelled as can all the rows that correspond to the node in the equilibrium point. This cancellation will produce the matrix $\tilde{C}$. The same applies to vector V; if its first element is set to zero it can be cancelled as far as the linear inequality is concerned, resulting in vector $\tilde{V}$ and the following set of inequalities:

$$\tilde{C}\tilde{V} < 0 \qquad (46)$$

with $\tilde{V} > 0$.

5.4 Presenting the LP to the solver

The set of linear inequalities (46) is not ready yet to be presented to the LP-solver. The solver that is used, is PCx from the Optimization Technology Center. (See the section on Software on page 45 for more information on PCx.) As all LP solvers, PCx needs an objective to minimise (maximise). Since there is no objective to minimise, one will be introduced. The set of linear inequalities (46) will be rewritten in the following way:

$$\tilde{C}\tilde{V} + z \cdot \mathbf{1} \leq 0 \qquad (47)$$

In the above $\mathbf{1}$ is a column vector of length equal to the number of rows of matrix $\tilde{C}$. All elements of vector $\mathbf{1}$ are equal to one. We also add and use scalar variable z. Now, we have an objective to be minimised in the form of the variable $-z$, i.e.

$$\min -z \qquad (48)$$

When we can solve (47) with $z > 0$ then (46) follows. This problem will be presented to PCx together with a constraint on the variable z:

$$0 \leq z \leq 1 \qquad (49)$$

It is necessary to limit the variable z, as in (49), because otherwise the solution may be unbounded from below. Let us observe that constraints written in the form (47) are never infeasible, since they are solved by substituting $\tilde{V} = 0$ and $z = 0$. If the solution to (47) (48) is $z_{opt} = 1$ then the system (46) is feasible. Another possibility is $z_{opt} = 0$ which indicates that the system of linear inequalities (46) is infeasible.

The upper bound on V is set as a constant in the MatLab code and can be set to a higher value if necessary. Normally the MatLab functions try to find a Lyapunov function that proves stability. For that situation $\tilde{V}$ has a lower bound of zero. However, if we try to find a Lyapunov function to prove that an equilibrium point is unstable, the lower bound has to be negative to allow for negative values of the Lyapunov function. This can be done by supplying an extra argument to the MatLab function to tell it to set the lower bound to the negative value of the upper bound.

The functions `lyapunov` and `lyadown` are written to generate the input file for the LP solver. These two functions provide the same functionality, but the former one considers the pendulum in its inverted position, while the latter considers the pending position. The functions write the LP problem to an MPS file, that can be read by PCx. The code and a brief description for `lyapunov` and `lyadown` can be found in appendix B.

5.5 Reading the results

Three functions are available to process the results from the solver. The functions `readmps` should be used to read the results if PCx is used to solve the LP problem. Optionally, the function `readling` can read the results, if the LP solver Lingo is used. The solution to the LP problem will be stored in a vector. The function `plotlya` will plot the Lyapunov function in this vector in a 3-dimensional graph.

The function `plotlya` plots a 3-dimensional graph of the piecewise linear Lyapunov function. Because it is not always easy to see if a particular point of a 3-dimensional graph is positive or negative, this representation is not ideal for the purpose of displaying Lyapunov functions. Besides that, often problems arise with the 2-dimensional representation of 3-dimensional graphs, which might lead to misinterpretations. This problem is even bigger when the graph is printed on paper, since the reader is no longer able to rotate the graph in the 3-dimensional space.

Especially in the case of Lyapunov functions, contour plots provide a better picture of the shape of the function. Since it is unknown how MatLab deals with piecewise linear functions, the available functions for producing contour plots can not be used without being studied first.

5.6 Numerical examples

5.6.1 The uncontrolled upright position

In chapter 3 *Swing-up by the energy method* it was shown that the uncontrolled upright position is unstable. Of course this is not a surprise, but it is a good opportunity to show that the LP solver is able to find a Lyapunov function that proves the instability of the uncontrolled upright position of the pendulum. The instability is proven in a neighbourhood of the upright position that is decomposed into a 31×31 triangulation with a node spacing of 0.0125×0.0125 [rad $\times$ rad s^{-1}]. Based on this triangulation, entries of the matrix $\tilde{C}$ were calculated and the problem (47) (48) was submitted to the LP solver PCx. The size of the LP problem was 5392 rows (inequalities) and 961 columns (variables). The value of the objective z was 1, which indicates that (46) is feasible.

The condition (46) is satisfied which proves that the derivative of the Lyapunov function is strictly negative. On the other hand as seen in figure 14 (and this can also be verified by inspecting the output file of the LP solver) the Lyapunov function takes negative values arbitrarily close to the origin. This proves the instability of the uncontrolled upright position.

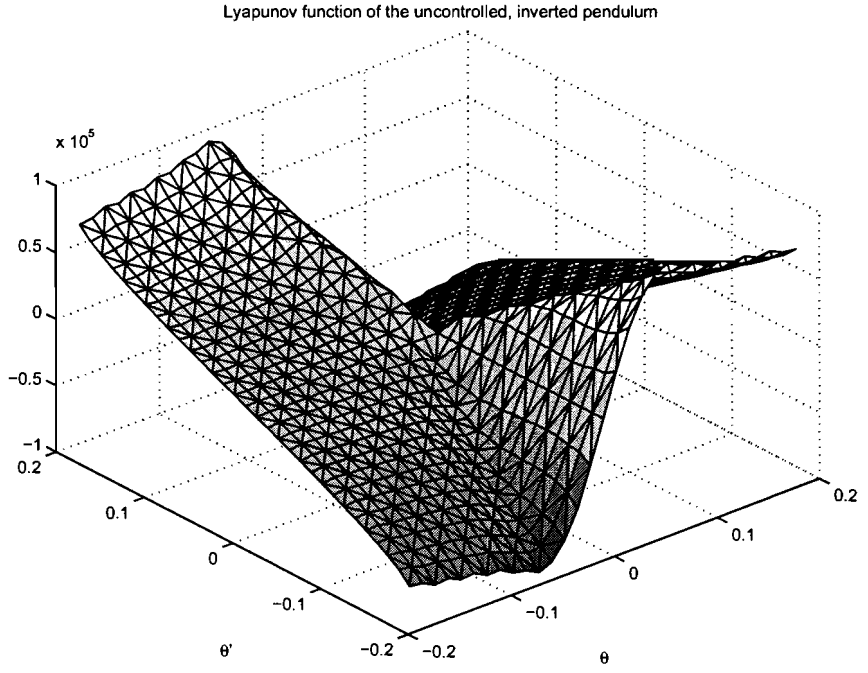


Figure 14: Lyapunov function of the uncontrolled upright pendulum

5.6.2 The non-linearly controlled pending position

In this paragraph it is shown that the pendulum controlled by the non-linear controller of equation (12) from chapter 3 *Swing-up by the energy method* is unstable in its pending position. This means that the pendulum will always leave this state what makes the pendulum swing up. The pendulum is actually not controlled by the non-linear controller, but by the piecewise linear approximation of this controller. The state space is decomposed into a 15 × 15 triangulation with a node spacing of 0.0249 × 0.0250 [rad × rad s^{-1}]. The size of the LP problem is 1172 rows and 225 columns.

The condition (46) is satisfied which proves that the derivative of the Lyapunov function is strictly negative. On the other hand as seen in figure 15 the Lyapunov function takes negative values arbitrarily close to the origin. This proves the instability of the uncontrolled upright position. Even stronger, we can see that the value of the Lyapunov function gets more and more negative if we move away from the pending position, which means that if the pendulum is in a state close to the pending position, it will be forced further away from this pending position. This proves that a piecewise linear controller approximating the non-linear controller does indeed do a swing-up.

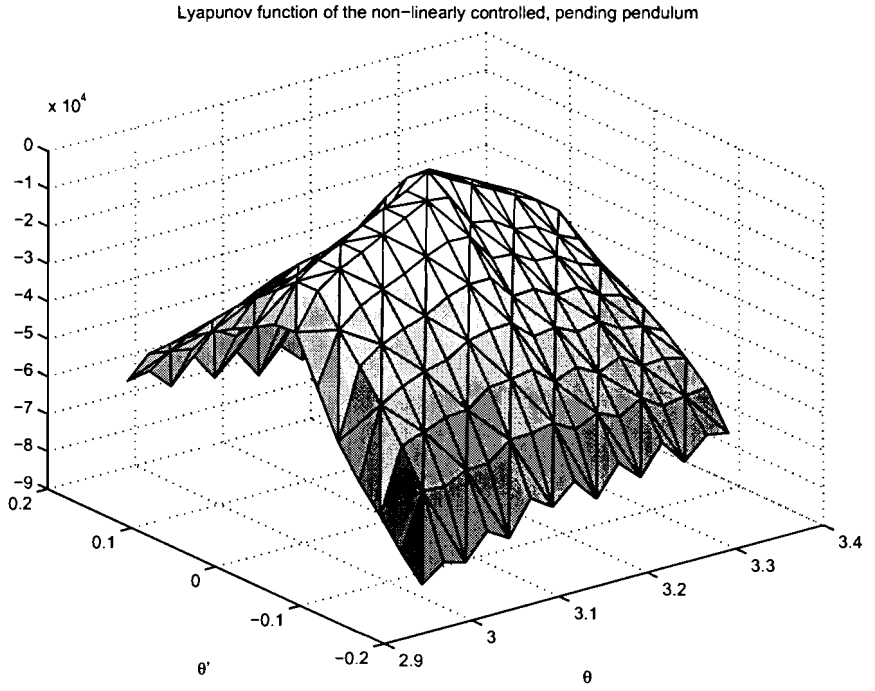


Figure 15: Lyapunov function of non-linearly controlled pendulum (pending position)

This calculation has also been done on a number of finer triangulations among which the triangulation of the previous paragraph on the uncontrolled, inverted pendulum, namely a 31×31 triangulation with a node spacing of 0.0125×0.0125 [rad $\times$ rad s^{-1}]. Remarkable is that the LP solver comes with a solution where the objective z is a very small number close to zero ($5 \cdot 10^{-7}$), showing that the solver could not find another solution than the zero solution.

A possible explanation for this behaviour is that the singular points around the equilibrium point in the downward position are causing problems. This problem might be evaded by splitting the nodes that coincide with a singular point into two separate nodes with the same co-ordinates, one being the vertex of the cells in the upper half of the state space and the other being the vertex of the cells in the lower half of the state space. This aspect needs further studying.

5.6.3 The non-linearly controlled upright position

In the previous paragraph it was shown that by applying the non-linear control law to the pendulum, it will do a swing-up. In this paragraph we examine the upright position. It will be shown that the non-linearly controlled pendulum is in fact not stable in its upward position. This means that the controller is not able to keep the pendulum in its upright position. The LP problem has the following size: 5392 rows and 961 columns.

Condition (46) is satisfied again which proves that the derivative of the Lyapunov function is strictly negative. It can be seen that the Lyapunov function takes negative values arbitrarily close to the origin. This proves that the pendulum is unstable in its inverted position if controlled by a piecewise linear approximation of the non-linear controller as defined in (12) in chapter 3 *Swing-up by the energy method.*

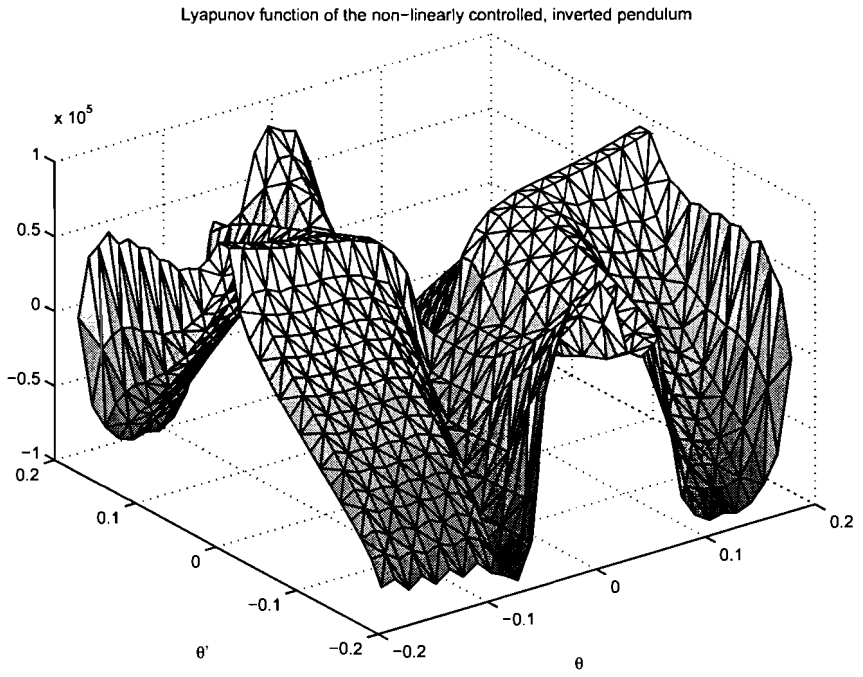


Figure 16: Lyapunov function of the non-linearly controlled pendulum

5.6.4 The linearly controlled upright position

Since the non-linear controller is not able to keep the pendulum in its upright position, we switch to another (linear) controller to balance the pendulum. This paragraph shows that the linear controlled pendulum is stable in its upright position, so the controller is able to balance the pendulum. The size of the LP problem is 5392 rows and 961 columns.

Since condition (46) is satisfied, the derivative of the Lyapunov function is strictly negative. Figure 17 shows that the Lyapunov function is positive in all the points of the examined region. It is difficult to see the shape of the Lyapunov function accurately though. By rotating the 3-dimensional graph on a computer it can be seen that the Lyapunov function does have a minimum in the origin. This proves the stability of the linearly controlled inverted position.

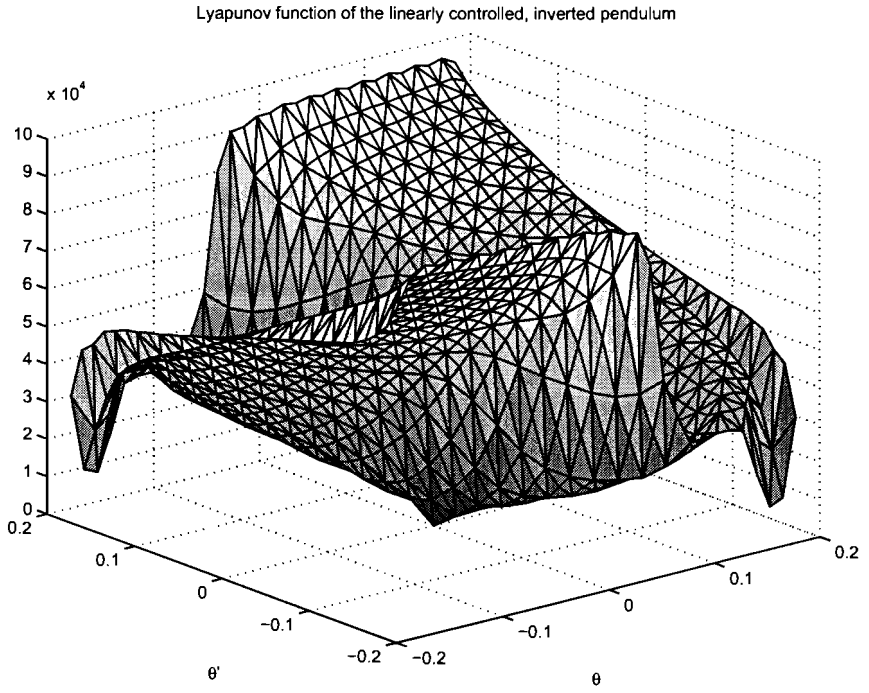


Figure 17: Lyapunov function of the linearly controlled upright position

5.7 Generation times of the Lyapunov functions

The times needed to generate the LP problem, to solve the LP problem and to plot the Lyapunov functions have been measured and are mentioned in the next table. The calculations were performed on a Pentium 75 MHz with Windows 95 and 48 MB RAM. The state space decomposition was set to respectively a 15 × 15 and a 31 × 31 triangulation with a node spacing of 0.049 × 0.050 in both cases. Table 3 shows the times for the different situations and subprocesses.

Table 3: Process time of various subprocesses for finding a Lyapunov function

Process time in seconds [s]	LP generation	LP solving	Plotting	Total
15 x 15 triangulation				
Uncontrolled upright position	896.2	27.7	423.2	1347.1
Non-linearly controlled pending position	853.0	33.6	129.8	1016.4
Non-linearly controlled upright position	893.9	37.9	539.6	1471.4
Linearly controlled upright position	894.7	33.5	155.9	1084.1
Average	884.5	33.2	312.1	1229.8
31 x 31 triangulation				
Uncontrolled upright position	4988.5	403.5	823.7	6215.7
Non-linearly controlled pending position	4851.2	535.2	566.0	5952.4
Non-linearly controlled upright position	5099.6	558.5	1777.7	7435.8
Linearly controlled upright position	5028.8	572.77	696.9	6298.5
Average	4992.0	517.5	966.1	6475.6

Clearly the generation of the LP problem takes the majority of the time – over 70%. The majority of the remaining time is needed for the plotting of the Lyapunov function. Although the solving of the LP problem is most probably the toughest job, it needs the smallest amount of time. This can be easily explained. The LP solver has been written in the programming language C and a lot of time has been put in the optimisation of the code by its programmers. The code that generates the LP problem and the code that plots the Lyapunov function are written in the form of a MatLab m-file. A limited amount of time was devoted to optimising the MatLab code, so probably the code can be made more efficient.

5.8 Conclusions

A method to define a piecewise linear Lyapunov function on our triangulation has been presented in this chapter. It was shown how the derivative of the piecewise linear Lyapunov function along solutions of the differential equations can be calculated. Then it was shown how the problem of finding a piecewise linear Lyapunov function can be formulated as an LP problem, to solve the problem with an LP solver.

Four numerical examples have been presented, to show how the stability of a region can be demonstrated with the piecewise linear Lyapunov functions found by the LP solver. The 3-dimensional representation of the piecewise linear Lyapunov functions is sensitive to misinterpretation by the viewer/ reader, especially if these representations are printed on paper. A function that plots the piecewise linear Lyapunov functions in a contour plot might aid the viewer/ reader a lot in interpreting the represented data. An additional advantage of contour plots is that it is much easier to find a proper Lyapunov contour.

Measurement have been performed on the time to generate the LP problems, to solve them, and to present the results. The majority of the time is needed for the generation of the problems. Solving the problems takes the least amount of time. On a Pentium 75 MHz with 48 MB RAM the process of generating the problem till displaying the results takes roughly two hours for a 31×31 triangulation. This time can probably be decreased by optimising the MatLab code. The complexity of the algorithm is linear with the number of nodes in the triangulation.

6. Design of piecewise linear control

In the previous chapter it was shown that we are able to transform the problem of finding a Lyapunov function to accommodate the evaluation of the stability of a piecewise affine approximation of a system into a Linear Program, which can be solved by solvers for LP problems. In this chapter this idea is extended. A mathematical solver will not only find a Lyapunov function, but will also find a piecewise linear controller that will turn an arbitrary point in state space into a stable equilibrium point. This is done by transforming the problem into a *linear complementarity problem* (LCP). It can not be guaranteed that the resulting LCP is feasible though.

In the first section it is demonstrated how a piecewise linear controller can be defined on our triangulated state space in a similar way as was done with the definition of the piecewise linear Lyapunov function. The second section shows how the problem is first formulated as a bilinear system of inequalities. This formulation is then transformed into a set of linear inequalities with additional complementarity conditions. Finally a transformation is made into an LCP. A MatLab function was written to derive matrices that describe this LCP for the case of the inverted pendulum. This MatLab function then transforms the matrices into a form that can be send to an LCP server on the Internet. The results obtained with this MatLab function are described in the third section of this chapter. The final section gives some concluding remarks on this chapter.

6.1 Piecewise linear control

In the previous chapter an expression was derived for the calculation of the value of the Lyapunov function for an arbitrary point x in the state space. We can derive a similar expression for the value of the control action $u(x)$ for an arbitrary point x in the piecewise linearly approximated state space.

The control law is defined as a function of state co-ordinates $u(\theta,\dot{\theta}) = u(x)$. Its values are specified in each node. Within the cells, the value of the control law can be found by interpolating the values at the three vertices of the cell. The value at a node k will be denoted by u_k. The values u_k are collected in a vector U:

$$U^{\mathrm{T}} = [u_0 \quad u_1 \quad u_2 \quad \ldots \quad u_K]$$

As in the case of the Lyapunov function – for comparison, see equation (38) – the value of the control law for an arbitrary point x within cell m can be found with the following formula:

$$u(x) = [U^{\mathrm{T}}]_{\kappa(m)}\,\beta \qquad (50)$$

Combining equations (36) and (50) lead to:

$$u(x) = \left[U^{\mathrm{T}}\right]_{\kappa(m)} \overline{X}^{-1}(m)\begin{bmatrix} x \\ 1 \end{bmatrix}, \qquad (51)$$

where m is the cell in which the point x is situated.

6.2 Conversion of the problem to an LCP

The problem of finding the piecewise linear controller is first formulated as a bilinear system of inequalities. This formulation will be transformed into a set of linear inequalities with additional complementarity conditions. Finally, this form is transformed into an LCP.

6.2.1 Formulation as a bilinear system of inequalities

If we substitute the equation for the linear controller (51) into our expression for the derivative (43) we obtain the following equation:

$$D_m^+(x) = \left[V^{\mathrm{T}}\right]_{\kappa(m)} \bar{X}^{-1}(m) \begin{bmatrix} a_m + A_m x + b_m u(x) \\ 0 \end{bmatrix} \tag{52}$$

$$= \left[V^{\mathrm{T}}\right]_{\kappa(m)} \bar{X}^{-1}(m) \begin{bmatrix} a_m + A_m x + b_m \left[U^{\mathrm{T}}\right]_{\kappa(m)} \bar{X}^{-1}(m) \begin{bmatrix} x \\ 1 \end{bmatrix} \\ 0 \end{bmatrix}$$

For stability the derivative should be negative, so in each cell C(m) the following inequality should hold for all three vertices:

$$\left[V^{\mathrm{T}}\right]_{\kappa(m)} \bar{X}^{-1}(m) \begin{bmatrix} a_m + A_m x + b_m \left[U^{\mathrm{T}}\right]_{\kappa(m)} \bar{X}^{-1}(m) \begin{bmatrix} x \\ 1 \end{bmatrix} \\ 0 \end{bmatrix} < 0 \tag{53}$$

The problem of finding a piecewise linear controller involves the calculation of the two vectors V and U. In cell C(m) we can write for the point $x = x_k$:

$$u_k = \left[U^T\right]_{\kappa(m)} \bar{X}^{-1}(m) \begin{bmatrix} x_k \\ 1 \end{bmatrix} \tag{54}$$

Substituting equation (54) into equation (53) results in:

$$D_m^+(x_k, u_k) = \left[V^{\mathrm{T}}\right]_{\kappa(m)} \bar{X}^{-1}(m) \begin{bmatrix} a_m + A_m x_k + b_m u_k \\ 0 \end{bmatrix} \tag{55}$$

Since every cell has three nodes, there are three equations like equation (55) connected with each cell. Now we collect all equations (55) for all $m \in \mu(k)$. The following notation is introduced: $m_1, \ldots, m_{I_k}$ denote the cell numbers returned by the function $\mu(k)$, with $I_k \in \{1, 2, 4, 8\}$.[5] Next we define row vectors $g_i^{\mathrm{T}}(k)$ with i in the range $1, 2, \ldots, I_k$:

$$g_i^{\mathrm{T}}(k) = \left\langle \bar{X}^{-1}(m_i) \begin{bmatrix} a_{m_i} + A_{m_i} x_k \\ 0 \end{bmatrix} \right\rangle_{\kappa(m_i)}, \tag{56}$$

where $\langle . \rangle_{\kappa(m_i)}$ denotes that the vector $g_i^{\mathrm{T}}(k)$ is obtained by placing the three elements of the vector

[5] A node is connected to either one, two, four or eight cells. Nodes on the corner of the triangulated area are connected to either one or two cells, the nodes on the borders of the triangulation (except for the corner nodes) are connected to either two or four cells and all remaining nodes to either four or eight cells.

$$\bar{X}^{-1}(m_i)\begin{bmatrix} a_{m_i} + A_{m_i} x_k \\ 0 \end{bmatrix}$$

into the positions given by the elements of the list $\kappa(m_i)$. All the other elements of vector $g_i^{\mathrm{T}}(k)$ are zero.

The row vectors $g_i^{\mathrm{T}}(k)$ are then collected into the matrix G_k:

$$G_k = \begin{bmatrix} g_1^{\mathrm{T}}(k) \\ g_2^{\mathrm{T}}(k) \\ \vdots \\ g_{l_k}^{\mathrm{T}}(k) \end{bmatrix}$$

Similar to the above approach, we can define row vectors $l_i^{\mathrm{T}}(k)$:

$$l_i^{\mathrm{T}}(k) = \left\langle \bar{X}^{-1}(m_i)\begin{bmatrix} b_{m_i} \\ 0 \end{bmatrix} \right\rangle_{\kappa(m_i)} \tag{57}$$

The term $\langle\rangle_{\kappa(m_i)}$ has a similar meaning as in equation (56).

The rows $l_i^{\mathrm{T}}(k)$ are collected into the matrix L_k:

$$L_k = \begin{bmatrix} l_1^{\mathrm{T}}(k) \\ l_2^{\mathrm{T}}(k) \\ \vdots \\ l_{l_k}^{\mathrm{T}}(k) \end{bmatrix}$$

Now we can rewrite the derivatives in the following way:

$$D^+(k) = \begin{bmatrix} D^+_{m_1}(x_k, u_k) \\ D^+_{m_2}(x_k, u_k) \\ \vdots \\ D^+_{m_{l_k}}(x_k, u_k) \end{bmatrix} = G_k V + u_k L_k V \tag{58}$$

The problem of finding a piecewise linear controller can be restated as follows:

Find vectors V and U such that:

$$\forall_{k=1,2,\ldots,K} D^+(k) < 0 \tag{59}$$

Equations (58) and (59) form a bilinear system of inequalities. To solve for the vectors V and U this system is transformed into a linear complementarity problem. However, first this bilinear system of inequalities is transformed into a conjunction of inequalities with additional complementarity conditions.

6.2.2 Transformation into a conjunction of inequalities with additional complementarity conditions

A solution to inequality (59) exists if one of the following conditions is true:

1. The first term in equation (58) is less than zero. By setting u_k to zero inequality (59) will hold:

$$G_k V + u_k L_k V < 0, \tag{60}$$

which evaluates to the following inequality if $u_k = 0$:

$$G_k V < 0. \tag{61}$$

2. If $G_k \geq 0$, then inequality (59) can be satisfied by either setting u_k to u_{max}:

$$G_k V + u_{\max} L_k V < 0, \tag{62}$$

which will be denoted as:

$$\hat{G}_k V < 0, \tag{63}$$

where $\hat{G}_k = G_k + u_{\max} L_k$,

3. or by setting u_k to $u_{\min}$:

$$G_k V + u_{\min} L_k V < 0, \tag{64}$$

which will be denoted as:

$$\check{G}_k V < 0, \tag{65}$$

where $\check{G}_k = G_k + u_{\min} L_k$.

By introducing the nonnegative scalar variables α_k^+, α_k^-, $\hat{\alpha}_k^+$, $\hat{\alpha}_k^-$, $\check{\alpha}_k^+$, $\check{\alpha}_k^- \geq 0$ we can write this as a conjunction of inequalities:

$$\begin{aligned} &G_k V + (\alpha_k^+ - \alpha_k^-)\mathbf{1} \leq 0 \\ &\hat{G}_k V + (\hat{\alpha}_k^+ - \hat{\alpha}_k^-)\mathbf{1} \leq 0 \\ &\check{G}_k V + (\check{\alpha}_k^+ - \check{\alpha}_k^-)\mathbf{1} \leq 0 \\ &\alpha_k^+ + \hat{\alpha}_k^+ + \check{\alpha}_k^+ > 0 \end{aligned} \tag{66}$$

where $\mathbf{1}$ is a $I_k \times 1$ vector; and additional complementarity conditions:

$$\begin{aligned} &\alpha_k^+ \alpha_k^- = 0 \\ &\hat{\alpha}_k^+ \hat{\alpha}_k^- = 0. \\ &\check{\alpha}_k^+ \check{\alpha}_k^- = 0 \end{aligned} \tag{67}$$

6.2.3 Transformation into a Linear Complementarity Problem

Formulae (66) and (67) can be transformed into a linear complementarity problem. Once stated as a LCP the problem of swinging up the pendulum and stabilising it in its inverted (upright) position can be solved by offering the LCP to a solver for LCP's.

We rewrite (66) for $k = 1, 2, \ldots, K$ respectively as:

$$-GV - L\alpha^+ - M\alpha^- = z \tag{68}$$

and

$$N\alpha^+ - q = u \tag{69}$$

where

$$G = \begin{bmatrix} G_1 \\ \hat{G}_1 \\ \breve{G}_1 \\ \vdots \\ G_K \\ \hat{G}_K \\ \breve{G}_K \end{bmatrix},$$

$$L = \text{diag}(\mathbf{1}, \mathbf{1}, \ldots, \mathbf{1}), \quad \text{and} \quad M = -L,$$

$$\alpha^+ = \begin{bmatrix} \alpha_1^+ \\ \hat{\alpha}_1^+ \\ \breve{\alpha}_1^+ \\ \vdots \\ \alpha_K^+ \\ \hat{\alpha}_K^+ \\ \breve{\alpha}_K^+ \end{bmatrix}, \quad \text{and} \quad \alpha^- = \begin{bmatrix} \alpha_1^- \\ \hat{\alpha}_1^- \\ \breve{\alpha}_1^- \\ \vdots \\ \alpha_K^- \\ \hat{\alpha}_K^- \\ \breve{\alpha}_K^- \end{bmatrix}$$

$$N = \text{diag}([1\ 1\ 1], [1\ 1\ 1], \ldots, [1\ 1\ 1])$$

z and u are additional vector variables satisfying $z \geq 0$ and $u \geq 0$. q is a given vector $q = \varepsilon \cdot \mathbf{1}$, where ε is small positive real-valued scalar.

6.2.4 Rearranging the matrices

The matrix L is of the following form:

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 & \cdots \\ 1 & 0 & 0 & 0 & \cdots \\ 1 & 0 & 0 & 0 & \cdots \\ 0 & 1 & 0 & 0 & \cdots \\ 0 & 1 & 0 & 0 & \cdots \\ 0 & 1 & 0 & 0 & \cdots \\ 0 & 0 & 1 & 0 & \cdots \\ 0 & 0 & 1 & 0 & \cdots \\ 0 & 0 & 1 & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

We can rearrange the matrix by taking the first row from each group of equal rows and placing those in the upper part of the matrix and leaving the remaining rows in the lower part of the matrix. The first K rows will then form a submatrix with the form of an identity matrix:

$$L = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & \cdots \\ 0 & 1 & 0 & \cdots & 0 & \cdots \\ 0 & 0 & 1 & \cdots & 0 & \cdots \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & 0 & 0 & \cdots & 0 & \cdots \\ 1 & 0 & 0 & \cdots & 0 & \cdots \\ 0 & 1 & 0 & \cdots & 0 & \cdots \\ 0 & 1 & 0 & \cdots & 0 & \cdots \\ 0 & 0 & 1 & \cdots & 0 & \cdots \\ 0 & 0 & 1 & \cdots & 0 & \cdots \\ \vdots & \vdots & \vdots & \cdots & \vdots & \ddots \end{bmatrix}$$

After rearranging L and rearranging G and M accordingly, equation (68) has the following form:

$$-\begin{bmatrix} G_1 \\ G_2 \end{bmatrix} V - \begin{bmatrix} \mathrm{I} \\ L_2 \end{bmatrix} \alpha^+ - \begin{bmatrix} -\mathrm{I} \\ M_2 \end{bmatrix} \alpha^- = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix}$$

This equation is equivalent to the following system of equations:

$$-G_1 V - \alpha^+ + \alpha^- = z_1 \qquad (70)$$

$$-G_2 V - L_2 \alpha^+ - M_2 \alpha^- = z_2 \qquad (71)$$

Solving (70) for α^+ leads to:

$$a^+ = a^- - G_1 V - z_1 \qquad (72)$$

This equation can now be put into a form that suits better for linear complementarity solvers:

$$\begin{bmatrix} \alpha^+ \\ .. \\ .. \\ .. \end{bmatrix} = \begin{bmatrix} \mathrm{I} & -G_1 & -\mathrm{I} & 0 \\ .. & .. & .. & .. \\ .. & .. & .. & .. \\ .. & .. & .. & .. \end{bmatrix} \begin{bmatrix} \alpha^- \\ V \\ z_1 \\ .. \end{bmatrix} \qquad (73)$$

Matrix elements denoted by a double period (..) are to be filled in yet. The variables V and z_1 both need corresponding complementary variables. These complementary variables will simply be introduced and called c_V and c_{z_1} respectively. Since we are not interested in the value of these complementary variables, we will make them independent of the other variables by putting zeros in the corresponding rows of the matrix.

$$\begin{bmatrix} \alpha^+ \\ c_V \\ c_{z_1} \\ .. \end{bmatrix} = \begin{bmatrix} \mathrm{I} & -G_1 & -\mathrm{I} & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ .. & .. & .. & .. \end{bmatrix} \begin{bmatrix} \alpha^- \\ V \\ z_1 \\ .. \end{bmatrix} \qquad (74)$$

Equation (71) has to be rearranged in a similar way. We start by substituting (72) into the (71):

$$-G_2 V - L_2 (\alpha^- - G_1 V - z_1) - M_2 \alpha^- = z_2$$

which can be rewritten as:

$$(-L_2 - M_2) \alpha^- + (G_1 L_2 - G_2) V + L_2 z_1 - z_2 = 0$$

Next a complementary variable for z_2 is introduced and is named c_{z_2} :

$$(-L_2 - M_2)\alpha^- + (G_1 L_2 - G_2)V + L_2 z_1 - z_2 = c_{z_2}$$

This equation can be added to the matrix (74) too:

$$\begin{bmatrix} \alpha^+ \\ c_V \\ c_{z_1} \\ c_{z_2} \\ .. \end{bmatrix} = \begin{bmatrix} \mathrm{I} & -G_1 & -\mathrm{I} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ -L_2 - M_2 & G_1 L_2 - G_2 & L_2 & -\mathrm{I} & 0 \\ .. & .. & .. & .. & .. \end{bmatrix} \begin{bmatrix} \alpha^- \\ V \\ z_1 \\ z_2 \\ .. \end{bmatrix} \qquad (75)$$

The first equation (70) of our system has been transformed into a linear complementarity problem. The second equation (69) is to be rewritten too, so it can be fitted into the linear complementarity problem. We do this by substituting the solution for α^+ (equation (72)) into the second equation (69) of our system:

$$N(\alpha^- - G_1 V - z_1) - q = u$$

Next, the complementary variable of u is introduced c_u:

$$N(\alpha^- - G_1 V - z_1) - q - u = c_u$$

Finally, we incorporate this equation in our matrix (75):

$$\begin{bmatrix} \alpha^+ \\ c_V \\ c_{z_1} \\ c_{z_2} \\ c_u \end{bmatrix} = \begin{bmatrix} \mathrm{I} & -G_1 & -\mathrm{I} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ -L_2 - M_2 & G_1 L_2 - G_2 & L_2 & -\mathrm{I} & 0 \\ N & -G_1 N & -N & 0 & -\mathrm{I} \end{bmatrix} \begin{bmatrix} \alpha^- \\ V \\ z_1 \\ z_2 \\ u \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ -q \end{bmatrix} \qquad (76)$$

The corresponding complementarity conditions are:

$$\begin{aligned} \alpha^+ \cdot \alpha^- &= 0 \\ c_V \cdot V &= 0 \\ c_{z_1} \cdot z_1 &= 0 \\ c_{z_2} \cdot z_2 &= 0 \\ c_u \cdot u &= 0 \end{aligned} \qquad (77)$$

Now the problem of finding a piecewise linear controller to swing up the pendulum and stabilise it in its inverted (upright) position has been transformed into a Linear Complementarity Problem. Since there are solvers available for LCP's, it is now possible to try to solve the problem of the inverted pendulum with one such solver.

6.3 Solving the LCP's

To generate the LCP as stated in equations (76) and (77) the MatLab function `genlcp` was written. To solve the LCP's, the PATH algorithm is used as it is available through the NEOS-server. (The reader is referred to the section on Software on page 45 for more information on PATH.) The NEOS server accepts LCP's via e-mail in the form of FORTRAN code, therefor function `genlcp` outputs its results in FORTRAN and writes the code to a file. The MatLab code for `genlcp` can be found in appendix B. The MatLab function `lcpsol` is intended to

read the results that are send back by the NEOS-server. Because the NEOS-server has not been found any solution yet, this function has not been tested though.

To test the method an LCP was generated to represent the swing-up problem in its simplest form – meaning that the neighbourhood of the inverted position was decomposed into only eight cells. This problem can actually not be considered as a representation of the swing-up problem. It is more close to the problem of balancing the pendulum in its upright position. The MatLab function produced an LCP with 117 variables. The LCP has been submitted to the solver, but unfortunately the server did not find a solution within its predefined maximum number of iterations (being equal to 500) although a simple linear controller should do the job. Since the NEOS-server did produce solutions for two simple test problems (that were produced without the MatLab function `genlcp`), we know the server is actually functioning.

At this moment it is difficult to determine why no solution is returned. Possible explanations are:

- The derivation for the formulation of the problem as an LCP contains an error;
- The MatLab function `genlcp` is faulty and has produced an LCP which is infeasible or the solution is too hard to find;
- The problem is too complex too be solved within the maximum number of iterations.

The last option does not sound very likely, since it is known that there exists a simple solution to the problem of balancing the pendulum in its inverted position and much more complicated LCP's (thousands of variables) have been solved by others (although it is not known if such problems have been solved by the NEOS-server).

The solver requires an initial point for the first iteration. This point was until now chosen quite arbitrary. A more considered initial point might enable the solver to produce a solution.

In the first place the derivation in this chapter needs more thorough checking and the code of function `genlcp` needs checking.

6.4 Conclusions

The previous chapter showed that it is possible to use an LP solver to search for (piecewise linear) Lyapunov functions to investigate the stability or instability of a neighbourhood of an arbitrary point in the state space. In this chapter this concept was extended. It was shown that it is possible to formulate the problem of finding a piecewise linear controller that makes a point in state space stable, as a linear complementarity problem. This LCP can be presented to a solver for (linear) complementarity problems to find such a controller and at the same time find a Lyapunov function to demonstrate the stability of the point.

A function was implemented in MatLab to transform the problem of swinging up the pendulum and balancing it in its inverted position into an LCP that was suitable for submission to a solver for LCP's. The solver did however not find a solution within its maximum number of iterations. At least the derivation in this chapter for formulating the swing-up problem as an LCP needs to be checked more thoroughly. Besides that, the MatLab code that generates the LCP needs checking too.

7. Conclusions and recommendations

This chapter concludes the report by presenting some conclusion and recommendations. The chapter is subdivided in two sections. The first section focuses on the process of finding piecewise linear Lyapunov functions. The second one deals with the method to find a piecewise linear controller for a piecewise affine approximation of a system.

7.1 Piecewise linear Lyapunov functions

In this report a system – the inverted pendulum – is approximated by a set of piecewise affine expressions, defined upon a state space that is decomposed in disjunct, triangular shaped cells. It is shown how a piecewise linear Lyapunov function can be defined upon this *triangulated* state space. Next to that, the report describes how the conditions stating that the piecewise linear Lyapunov function should have a strictly negative derivative (except for the equilibrium point under consideration), can be formulated as a Linear Program. This LP problem can be presented to a LP solver like PCx to find a piecewise linear Lyapunov function to study the stability of a neighbourhood of an equilibrium point. Using this method (piecewise linear) Lyapunov functions can be found that prove either the stability or instability of such a neighbourhood.

Some numerical examples are presented that prove either the stability or instability of the two equilibrium points – the pending and the inverted position – of the inverted pendulum when using a few simple control strategies. The knowledge of the stability or instability of these control strategies is not new, but the examples illustrate how the Lyapunov functions found by the LP solver can be used to examine stability.

Next to the LP solver, MatLab code has been written to produce the LP problem, and a second MatLab function to plot the Lyapunov function found by the LP solver. The MatLab code is not considerably fast, but the speed is still practical. The generation of the LP problem for a state space that is decomposed in nearly 1000 cells takes less than two hours on a Pentium 75 with 48 MB RAM. Solving is a matter of roughly a minute and the plotting process takes less than 10 minutes. The complexity of the MatLab functions is linear with the number of cells. The generation time of the LP problems can probably be reduced slightly by optimising the code and drastically by rewriting the code in a lower level computer language like C.

The visualisation of the Lyapunov functions is now by means of 3-dimensional plots. This has two disadvantages:

- The 3-dimensional plots are subject to misinterpretations, especially when the plot is printed on paper and can not be rotated anymore like on a computer monitor;
- With the analysis of the stability of the neighbourhood of an equilibrium point, one has special interest in Lyapunov contours, for example to find stability regions.

Therefor the visualisation can be greatly improved by presenting the Lyapunov functions by means of contour plots. The available MatLab functions for producing contour plots are not directly applicable for this purpose though.

7.2 Piecewise linear control

The second part of this report describes how the process of finding a piecewise linear Lyapunov function can be extended to search for a piecewise linear controller that will make the equilibrium point stable. The problem can no longer be expressed in the form of an LP problem. However, it is shown that the problem can be transformed into a Linear Complementarity Problem (LCP). Solvers for Complementarity Problems are available, for example the PATH algorithm. To be able to test this method of finding a piecewise linear controller, a MatLab function has been written to transform a control problem – in this case the swing-up and stabilisation of an inverted pendulum – into an LCP. The LCP is formulated in FORTRAN code, which is the input format of the PATH algorithm as implemented on the NEOS-server. This way the problem can be presented to the solver.

The size of the LCP is fairly big. For the simplest decomposition consisting of only eight cells, an LCP with 117 variables is being generated. Fortunately, this number expands only linear with the number of cells.

The MatLab-function has been tested, but unfortunately the server did not find a solution within its predefined maximum number of iterations (being equal to 500). Neither did it find a solution to (a part of) the swing-up problem, nor did it find a solution to the problem of balancing the pendulum in its inverted position.

Without further research it is not possible to find out why no solution is found. Possible explanations are:

- The MatLab code is faulty and produces either infeasible problems or problems that are just too complex to solve in only 500 iterations;
- The MatLab code is working correct, but the problem is too complex to solve in only 500 iterations.

The last explanation sounds not very likely, since there exists a simple solution to the problem of balancing the pendulum in its inverted position and according to literature quite complex LCP's (several thousands of variables) have been solved already.

Additional research to find the cause of this problem should be done.

Literature

[1] Åström, K.J. and K. Furuta
Swinging up a pendulum by energy control
IFAC 13th World Congress, San Francisco, California, USA, 1996

[2] Chung Choo Chung and John Hauser
Nonlinear control of a swinging pendulum
Automatica, Vol. 31 (1995), No. 6, pp. 851-862
Oxford: Pergamon Press

[3] Cottle, R.W., Jong-Shi Pang, and R.E. Stone
The linear complementarity problem
San Diego: Academic Press: 1992

[4] Julián, P., J. Guivant, and A. Desages
A parametrization of piecewise linear Lyapunov functions via linear programming
Int. J. Control, Vol. 72 (1999), No. 7/8, pp. 702-715
Taylor &Francis Ltd.

[5] Khalil, H.K.
Nonlinear systems
New York: Macmillan Publishing Company: 1992

[6] Murty, K.G.
Linear Complementarity, Linear and Nonlinear Programming, Internet edition
Internet Edition prepared by Feng-Tien Yu: 1997
`http://www.personal.engin.umich.edu/~murty/`
Original edition: Helderman-Verlag, 1988

[7] Qifeng Wei, W.P. Dayawansa and W.S. Levine
Nonlinear controller for an inverted pendulum having restricted travel
Automatica, Vol. 31 (1995), No. 6, pp. 841-850
Oxford: Pergamon Press

[8] Rouche, N., P. Habets and M. Laloy
Stability theory by Lyapunov's direct method
Heidelberg: Springer, 1977

[9] Yoshizawa, T.
Stability theory by Liapunov's second method
Tokyo: The Math. Soc. of Japan, 1966

Software

Numerical mathematics:

MATLAB version 5.0.0.4069 on PCWIN, 22 November 1996
The MathWorks Inc., Natick, MA, USA

MATLAB toolboxes:

Control System Toolbox version 4.0, 15 November 1996

Optimization Toolbox version 1.5.0, 31 October 1996

Simulink version 2.0, 15 November 1996
The MathWorks Inc., Natick, MA, USA

Solver for Linear Programming problems:

PCx for Windows 95/ NT version 1.1, November 1997
Optimization Technology Center at Argonna National Laboratory and Northwestern University
`http://www-fp.mcs.anl.gov/otc/Tools/PCx/`

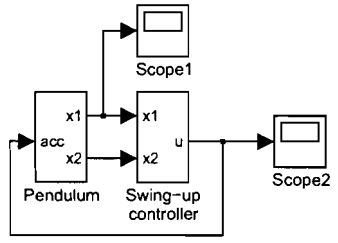
Solver for Linear Complementarity Problems:

NEOS-server, Path version 3.2, 16 February 1998
T.S. Munson, S.P. Dirkse, M.C. Ferris; University of Wisconsin
`http://www.mcs.anl.gov/neos/Server/`

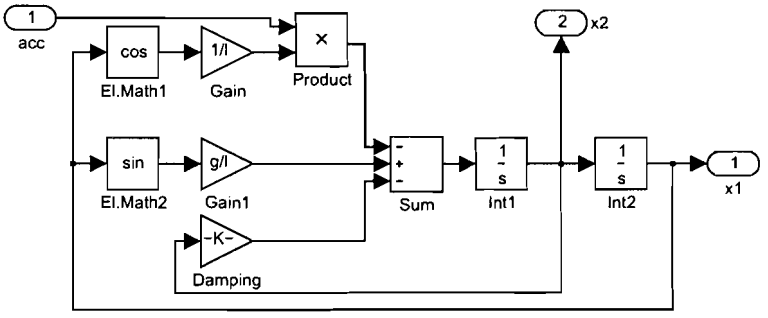
Appendix A: Simulink files

The Simulink files used for the simulations in this report are listed in this appendix. There are two models: The first describes a pendulum with only a swing-up controller. This controller does do the swing-up, but is not capable of keeping the pendulum in its inverted position. The second model has an additional linear controller. This linear controller will stabilise the pendulum in its inverted position.

Module 1: Pendulum with non-linear controller



Pendulum:

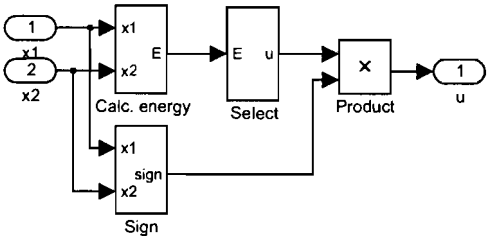


The gain *Damping* has a value of: $\frac{d}{ml^2}$

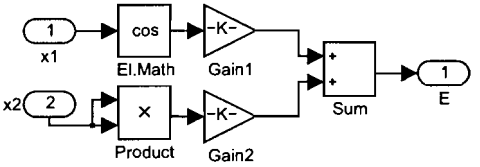
The initial condition of integrator *Int1* is: 0.1 rad s^{-1}

The initial condition of integrator *Int2* is: 0 rad

Swing-up controller:



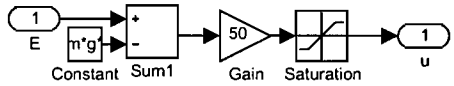
Calc. energy:



The gain *Gain1* has a value of: mgl

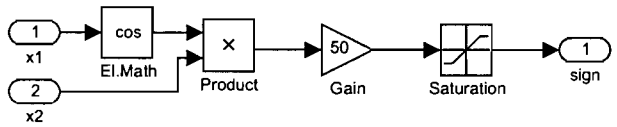
The gain *Gain2* has a value of: $\frac{1}{2}ml^2$

Select:



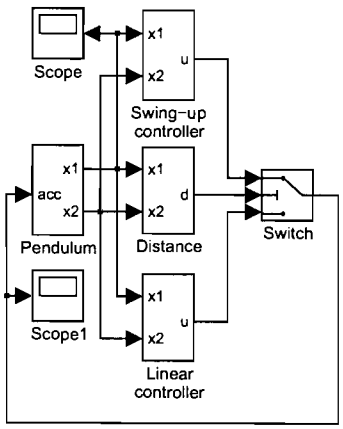
The constant *Constant* has a value of: mgl
The saturation *Saturation* has a lower bound of: $u_{\min}$
and an upper bound of: $u_{\max}$

Sign:

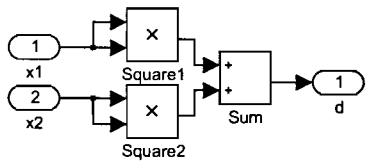


The saturation has a lower bound of: -1
and an upper bound of: $+1$

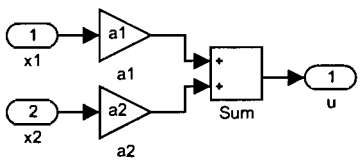
Model 2: Pendulum with swing-up controller and stabiliser



Distance:



Linear controller:



Appendix B: MatLab code

All MatLab functions written for the research described in this report are listed here along with a short description of their purpose, a listing of the inputs, outputs and other functions the described function depends on. The function `setenv` should be invoked before any of the other functions is called to set the configuration to act on. If such configuration does not exist yet, `writecfg` should be invoked right after invoking `setenv`.

Function `barrycnt`

Description:
The function `barrycnt` returns for a point x in which cell m it is lying, the numbers k of the nodes of cell m and the triangular or barrycentric co-ordinates a of point x in the cell.

Inputs:

x	The co-ordinates of point x

Outputs:

m	Number of the cell in which point x is lying
k	Row vector with the numbers of the nodes of cell m
a	Triangular co-ordinates of point x in cell m

Depends on: `odd, even`

Code:

```
function [m,k,a]=barrycnt(x)
%BARRYCNT  Calculates cell number, number of the nodes of the cell
%          and the Barrycentric coordinates of the point in the cell
%
%   [m,k,a]=BARRYCNT(x)
%      Calculates in what cell 'm' the point x lays, the numbers
%      'k' of the nodes of the cell and derives the Barrycentric
%      coordinates 'a' in the cell.

global ENVIROMENT_NAME
if length(ENVIROMENT_NAME)==0
   error('Enviroment name not set. Use SETENV.');
end
if ~exist([ENVIROMENT_NAME '.mat'],'file')
   error('Configuration file does not exist. Use WRITECFG.');
else
   load(ENVIROMENT_NAME);
end

x(1)=mod(x(1),2*pi);

% Calculate horizontal spacing
Dh=2*pi/Nh;

RowNo=2*abs(fix(x(2)/Dv));
if x(2)<0
   RowNo=RowNo+1;
end

BotLeft=(RowNo*Nh)+fix(x(1)/Dh);

% Calculate the number of cells around node 'BotLeft'
sect  = even(fix((BotLeft+Nh)/Nh/2));
eight = (odd(BotLeft) & (~sect)) | ((~odd(BotLeft)) & sect);

%Determine cell number 'm'

if eight
```

```
   xn=coord(BotLeft);
   if (x(2)-xn(2))>(x(1)-xn(1))*Dv/Dh
      m=BotLeft*2;
   else
      m=BotLeft*2+1;
   end
else
   BotRight=BotLeft+1;
   if ~mod(BotRight,Nh)
      BotRight=BotRight-Nh;
   end
   xn=coord(BotRight);
   if (~mod(BotRight,Nh))&(xn(1)==0)
      xn(1)=2*pi;
   end
   if (x(2)-xn(2))>(xn(1)-x(1))*Dv/Dh
      m=BotLeft*2+1;
   else
      m=BotLeft*2;
   end
end

k=c2n(m);
xn=coord(k);
% If point x is in the rightmost column, set the x(1)-coordinates
% of the nodes to the right to 2*pi instead of 0.
if mod(BotLeft,Nh)==Nh-1
   xn(1,xn(1,:)==0)=2*pi;
end
a=[xn(1,:); xn(2,:); ones(1,length(xn(1,:)))]\[x(1); x(2); 1];
```

Function `c2n`

Description:
The function `c2n` returns a row vector with the numbers of the nodes that are the vertices of the cell designated by input *m*.

Inputs:

m	The cell number

Outputs:

k	Row vector with the node numbers

Depends on: `c2n`, `coord`, `even`, `odd`

Code:

```
function k=c2n(m)
%C2N  Calculates the numbers of the nodes of a cell
%
%   k=C2N(m)
%      Returns a vector 'k' containing the numbers of the nodes
%      forming the cell with cell number 'm'.

global ENVIROMENT_NAME
load(ENVIROMENT_NAME);

BotLeft=fix(m/2);
RowNo = fix(BotLeft/Nh);

BotRight=BotLeft+1;
if ~mod(BotRight,Nh)
   BotRight=BotRight-Nh;
end

if RowNo==1
   TopLeft=BotLeft-Nh;
elseif odd(RowNo)
      TopLeft=BotLeft-2*Nh;
    else
      TopLeft=BotLeft+2*Nh;
end

TopRight=TopLeft+1;
```

```
if ~mod(TopRight,Nh)
   TopRight=TopRight-Nh;
end

% Calculate the number of cells around node 'BotLeft'
sect  = even(fix((BotLeft+Nh)/Nh/2));
eight = (odd(BotLeft) & (~sect)) | ((~odd(BotLeft)) & sect);

if eight
   k=[BotLeft TopRight];
   if even(m)
      k=[k TopLeft];
   else
      k=[k BotRight];
   end
else
   k=[TopLeft BotRight];
   if even(m)
      k=[k BotLeft];
   else
      k=[k TopRight];
   end
end

k=sort(k);
```

Function `coord`

Description:
The function `coord` returns a matrix with the cartesian co-ordinates of the nodes designated by input *k*.

Inputs:

k	A row vector with node numbers for which the corresponding co-ordinates must be calculated.

Outputs:

x	A matrix with the co-ordinates of the nodes.

Depends on: `odd`

Code:

```
function x=coord(k)
%COORD  Return the cartesian coordinates of the nodes
%
%   x=COORD(k)
%      Calculates the cartesian coordinates of the nodes defined by
%      rowvector k.

global ENVIROMENT_NAME
load(ENVIROMENT_NAME);

NmbOfNodes=length(k);
x=zeros(2,NmbOfNodes);

for i=1:NmbOfNodes
   x(1,i)=mod(k(i),Nh)*2*pi/Nh;
   RowNo=fix(k(i)/Nh);
   x(2,i)=fix((RowNo+1)/2)*Dv;
   if odd(RowNo)
      x(2,i)=-x(2,i);
   end
end
```

Function `cprt2glb`

Description:
The function `cprt2glb` converts cell numbers according to the numbering system of a partial triangulation to the numbering of the global triangulation.

Inputs:

mPrt	A row vector with (partial) cell numbers to be converted to global numbering.

Outputs:

mGlb	A row vector with (global) cell numbers.

Depends on: —

Code:

```
function mGlb=cprt2glb(mPrt)
%NPRT2GLB  Maps cell numbers from a partial to a global triangulation
%
%   mGlb=CPRT2GLB(mPrt)
%      The function accepts a cell number of a partial triangulation and
%      calcutates the number of the cell in the global triangulation

global ENVIROMENT_NAME
load(ENVIROMENT_NAME);

mGlb=zeros(1,length(mPrt));
for index=1:length(mPrt)
   RowNo=fix(mPrt(index)/(4*Nhp));
   ColNo=mod(mPrt(index),4*Nhp);
   if ColNo<2*Nhp
      mGlb(index)=RowNo*2*Nh+ColNo;
   else
      mGlb(index)=(RowNo+1)*2*Nh+(ColNo-4*Nhp);
   end
end
```

Function `ctrl`

Description:
The function `ctrl` defines the controller.

Inputs:

k	A row vector containing the numbers of the nodes for which the output of the controller should be calculated.

Outputs:

u	A row vector with the values of the output of the controller.

Depends on: —

Code:

```
function u=ctrl(k)

% CTRL  Returns the output of the controller
%
%   u=CTRL(k)
%      Returns the output of the controller if the system is in the state
%      corresponding to node 'k'. (Using global node numbers.)
%
%   u=CTRL(x)
%      Returns the output of the controller if the system is in the state
%      determined by co-ordinates x.

global ENVIROMENT_NAME
if length(ENVIROMENT_NAME)==0
   error('Enviroment name not set. Use SETENV.');
```

```
end
if ~exist([ENVIROMENT_NAME '.mat'],'file')
   error('Configuration file does not exist. Use WRITECFG.');
else
   load(ENVIROMENT_NAME);
end

[row,col]=size(k);

if row==1
   % Warning: No wrapping to negative co-ordinates takes place so far.
   echo('WARNING: No wrapping to negative co-ordinates takes place in module CTRL so
far.');
   x=coord(k);
else
   if row~=2
      error('Input vector should have 1 or 2 rows.')
   end
   x=k;
end

% Uncontrolled
if strcmp(controller,'uncontrolled')
   u=zeros(1,col);
end

% Linearly controlled
if strcmp(controller,'linear')
   u=a*x;
end

% Non-linearly controlled
if strcmp(controller,'non-linear')
   for i=1:col
      E=1/g*x(2,i)*x(2,i)/2+cos(x(1,i));
      if E==1
         u(1,i)=0;
      else
         u(1,i)=sign(x(2,i)*cos(x(1,i)));
         if E>1
            u(1,i)=umax*u(1,i);
         else
            u(1,i)=umin*u(1,i);
         end
      end
   end
end
```

Function `diffeq`

Description:
The function `diffeq` returns the values of the four components of the differential equations for the points designated by input x.

Inputs:

x	A matrix with in each column the co-ordinates of the points.

Outputs:

$f1x$	The value of component $f_1(x)$.
$g1x$	The value of component $g_1(x)$.
$f2x$	The value of component $f_2(x)$.
$g2x$	The value of component $g_2(x)$.

Depends on: —

Code:

```
function [f1x,g1x,f2x,g2x]=diffeq(x)
%DIFFEQ  Returns the result of the differential equations of the system
%
%   [f1x,g1x,f2x,g2x]=DIFFEQ(x)
%      This function calculates the results of the differential equations of the
```

```
%       inverted pendulum. Its inputs are the two states of the system ('x1' and 'x2')
%       representing the angle of the pendulum and the angular velocity.
%       The output is split up in two parts namely the part that is independent of the
%       control input and the part that does depend on the control input like in the
%       following equation:
%
%          dx = f(x) + g(x)*u
%       where 'dx' is the (time-)derivative of the state vector, 'x' is the state vector
%       and 'u' is the control input.
%
%       The four outputs of the system are:
%          - 'f1x' being the result of f1(x1,x2)
%          - 'g1x' being the result of g1(x1,x2)
%          - 'f2x' being the result of f2(x1,x2)
%          - 'g2x' being the result of g2(x1,x2)
%
%       The function assumes the existence of two global variables 'g' and 'l', being
%       the model parameters for the gravitational acceleration and the length of the
%       pendulum.

global ENVIROMENT_NAME
load(ENVIROMENT_NAME);

[NmbCoor,NmbOfPoints]=size(x);

f1x=zeros(1,NmbOfPoints);
g1x=f1x;
f2x=f1x;
g2x=f1x;

for i=1:NmbOfPoints
   f1x(i)=x(2,i);
   g1x(i)=0;
   f2x(i)=g/l*sin(x(1,i));
   g2x(i)=-1/l*cos(x(1,i));
end
```

Function `even`

Description:
The function `even` returns a non-zero value if the integer part of the input value is even, otherwise it returns a zero.

Inputs:

number	The number to be tested.

Outputs:

out	A non-zero number if the integer part of the input is even, otherwise 0.

Depends on: —

Code:

```
function out=even(number)
%EVEN  Returns TRUE if the integer part of its argument is even
out=~mod(fix(number),2);
```

Function `fitcell`

Description:
The function `fitcell` finds the parameters of the piecewise affine approximation for the cell designated by input *m*.

Inputs:

m	The number of the cell for which the fit should be done

plane	Determines if a cell is considered to be in either the left hand plane or the right hand plane. If the string *pos* is provided, the cell is considered to be in the right hand plane, if *neg* is provided the cell is supposed to be in the left hand plane. Default: *pos* Optional

Outputs:

a	Parameter a_m of the piecewise affine approximation.
A	Parameter A_m of the piecewise affine approximation.
b	Parameter b_m of the piecewise affine approximation.

Depends on: `c2n`, `coord`, `diffeq`

Code:

```
function [a,A,b]=fitcell(m,plane)
%FITCELL  Estimates the parameters for the piecewise affine approximation
%
%   [a,A,b]=FITCELL(m,plane)
%      The parameters for the piecewise affine approximation of cell m
%      in the triangulation will be estimated by fitting a plane through
%      the three nodes that are vertices of the cell.

global ENVIROMENT_NAME
load(ENVIROMENT_NAME);

if (nargin<2)|((plane~='pos')&(plane~='neg'))
   negplane=0;
else
   negplane=1;
end

a=zeros(2,1,length(m));
A=zeros(2,2,length(m));
b=zeros(2,1,length(m));
for index=1:length(m)
   k=c2n(m(index));
   x=coord(k);
   if negplane
      select=x(1,:)>=(Nh-Nhp)/Nh*2*pi;
      wrapper=[1; 0]*[1 1 1];
      x(:,select)= x(:,select) - 2*pi*wrapper(:,select);
   else
      if mod(fix(m/2),Nh)==Nh-1
         x(1,(abs(x(1,:))<(pi/Nh)))= 2*pi;
      end
   end
   [f1x,g1x,f2x,g2x]=diffeq(x);
   Paras1 = ...
            [      1  x(1,1)  x(2,1); ...
                   1  x(1,2)  x(2,2); ...
                   1  x(1,3)  x(2,3)] ...
            \ [ f1x(1); ...
                f1x(2); ...
                f1x(3)  ];
   Paras2 = ...
            [      1  x(1,1)  x(2,1); ...
                   1  x(1,2)  x(2,2); ...
                   1  x(1,3)  x(2,3)  ] ...
            \ [ f2x(1); ...
                f2x(2); ...
                f2x(3)  ];
   a(:,:,index)=[Paras1(1);Paras2(1)];
   A(:,:,index)=[Paras1(2) Paras1(3); Paras2(2) Paras2(3)];
   b(2,:,index)=(g2x(1)+g2x(2)+g2x(3))/3;
end
```

Function `genlcp`

Description:
The function `genlcp` generates the LCP to find a piecewise linear controller that makes the inverted position stable. The LCP will be written to an output file.

Inputs:

—

Outputs:

An output file containing the LCP described in Fortran code.

Depends on: `c2n`, `coord`, `fitcell`, `n2c`, `nglb2prt`, `nprt2glb`

Code:

```
function C=genlcp()
% GENLCP  Transforms the swing up of an inverted pendulum into a LCP
%
%   C=GENLCP()
%      The problem of swinging up the inverted pendulum is transformed into a
%      linear complementarity problem. Next the LCP is written to a fortran file
%      which is suitable to be sent to the NEOS server to solve the LCP.

tt=clock;                               % timer for total elapsed time
ct=cputime;                             % timer for used CPU time

global ENVIROMENT_NAME
if length(ENVIROMENT_NAME)==0
   error('Enviroment name not set. Use SETENV.');
end
if ~exist([ENVIROMENT_NAME '.mat'],'file')
   error('Configuration file does not exist. Use WRITECFG.');
else
   load(ENVIROMENT_NAME);
end

% Constants
ModName       = 'GENLCP';               % Module name (max. 8 characters)
Ext           = '.pth';                 % Extension for output file
RgtMar        = 72;                     % Right margin for output file
ProgTimeSlice = 60;                     % Time slice for progression indicator
leader        = 'Module         : ';    % Leader for output messages
spaces        = '                 ';    % Empty leader
epsilon       = 0.0001;                 % Small number

if length(ModName)>8
   ModName=ModName(1:8);
end
leader(8:7+length(ModName))=ModName;

Len=zeros(2,NoNodes);
G=zeros(8,NoNodes,NoNodes);
L=zeros(8,NoNodes,NoNodes);

% GENERATION OF THE INEQUALITIES

disp([leader 'Generating inequalities...']);
t0=clock;   % Initialize clock for progress indication
for kPrt=0:NoNodes-1
   Len(1,kPrt+1)=kPrt;
   kGlb=nprt2glb(kPrt);
   mGlb=n2c(kGlb);
   for i=1:length(mGlb)
      k=c2n(mGlb(i));
      if min(nglb2prt(k)>=0)&(max(nglb2prt(k))<NoNodes)
         Len(2,kPrt+1)=Len(2,kPrt+1)+1;
         X=coord(c2n(mGlb(i)));
         if mod(mGlb(i),Nh*2)>=2*(Nh-Nhp)
            [a,A,b]=fitcell(mGlb(i),'neg');
            select=X(1,:)>=(Nh-Nhp-0.5)/Nh*2*pi;
            wrapper=[1; 0]*[1 1 1];
            X(:,select)= X(:,select) - 2*pi*wrapper(:,select);
         else
            [a,A,b]=fitcell(mGlb(i));
```

```
            end
            Xhat=[X; 1 1 1];
            G(Len(2,kPrt+1),nglb2prt(c2n(mGlb(i)))+1,kPrt+1) = ...
               (inv(Xhat)*[a+A*coord(kGlb);0])';
            L(Len(2,kPrt+1),nglb2prt(c2n(mGlb(i)))+1,kPrt+1) = (inv(Xhat)*[b;0])';
         end
      end
      % Display progress indication
      if etime(clock,t0)>ProgTimeSlice
         disp([spaces num2str(kPrt/(NoNodes-1)*100,'%5.1f') '% of inequalities is
generated.'])
         t0=clock;
      end
   end
   disp([spaces 'Inequalities generated.']);
   Gbar=G+umax*L;
   Ghat=G+umin*L;

   % TRANSFORMATION OF INEQUALITIES TO A LCP

   sl=sum(Len,1);
   G1=zeros(3*NoNodes,NoNodes);
   G2=zeros(sl(2)-3*NoNodes,NoNodes);
   L2=zeros(sl(2)-3*NoNodes,3*NoNodes);
   N=zeros(NoNodes,3*NoNodes);
   row=1;
   disp([leader 'Transforming inequalities to LCP...']);
   t0=clock;   % Initialize clock for progress indication
   for i=1:NoNodes
      G1(3*(i-1)+1,:)=G(1,:,i);
      G1(3*(i-1)+2,:)=Gbar(1,:,i);
      G1(3*(i-1)+3,:)=Ghat(1,:,i);
      L2(row:row+3*(Len(2,i)-1)-1,3*(i-1)+1)=1;
      L2(row:row+3*(Len(2,i)-1)-1,3*(i-1)+2)=1;
      L2(row:row+3*(Len(2,i)-1)-1,3*(i-1)+3)=1;
      for j=2:Len(2,i)
         G2(row+j-2,:)=G(j,:,i);
         G2(row+(Len(2,i)-1)+j-2,:)=Gbar(j,:,i);
         G2(row+2*(Len(2,i)-1)+j-2,:)=Ghat(j,:,i);
         row=row+1;
      end
      row=row+2*(Len(2,i)-1); % Skip over the rows filled with entries from Gbar and Ghat
      N(i,3*(i-1)+1)=1;
      N(i,3*(i-1)+2)=1;
      N(i,3*(i-1)+3)=1;
      % Display progress indication
      if etime(clock,t0)>ProgTimeSlice
         disp([spaces num2str(kPrt/(NoNodes-1)*100,'%5.1f') '% of inequalities is
transformed.'])
         t0=clock;
      end
   end
   disp([spaces 'Inequalities transformed.']);
   M2=-L2;
   q=-epsilon*ones(NoNodes,1);

   % OUTPUT RESULTS TO FORTRAN FILE

   [rg1,cg1]=size(G1);
   [rg2,cg2]=size(G2);
   [rl2,cl2]=size(L2);
   [rlm,clm]=size(L2-M2);
   [rglg,cglg]=size(-G2+L2*G1);
   [rn,cn]=size(N);
   [rng,cng]=size(-N*G1);

   C=[eye(rg1,3*NoNodes)      -G1      -eye(rg1,cl2)     zeros(rg1,rl2)
   zeros(rg1,NoNodes); ...
      zeros(NoNodes, 3*NoNodes + NoNodes + cl2 + rl2 + NoNodes)
   ; ...
      zeros(cl2, 3*NoNodes + NoNodes + cl2 + rl2 + NoNodes)
   ; ...
            -L2-M2          -G2+L2*G1         L2            -eye(rl2,rl2)
   zeros(rl2,NoNodes); ...
              N               -N*G1           -N          zeros(NoNodes,rl2)   -
   eye(NoNodes,NoNodes)];

   [rC,cC]=size(C);

   l=[zeros(rg1+NoNodes+cl2+rl2,1);-q];
```

```
%Write output to file

fid=fopen([ENVIROMENT_NAME Ext],'w');
if fid==-1
    error(['Unable to open file ' ENVIROMENT_NAME Ext])
end

disp([leader 'Writing problem to file ' ENVIROMENT_NAME Ext]);

fprintf(fid,'TYPE CP\n');
fprintf(fid,'SOLVER PATH\n\n');

%Write Initial Point
fprintf(fid,'BEGIN.INITPT\n');
fprintf(fid,'      subroutine initpt(n,x)\n');
fprintf(fid,'      integer n\n');
fprintf(fid,'      double precision x(n)\n');
%All variables are initially set to 1.0
fprintf(fid,'      do i=1,n\n');
fprintf(fid,'      x(i) = 1.0d0\n');
fprintf(fid,'      enddo\n');
fprintf(fid,'      return\n');
fprintf(fid,'      end\n');
fprintf(fid,'END.INITPT\n\n');

t0=clock;
%Write function
fprintf(fid,'BEGIN.FCN\n');
fprintf(fid,'      subroutine fcn(n,x,f)\n');
fprintf(fid,'      integer n\n');
fprintf(fid,'      double precision x(n)\n');
fprintf(fid,'      double precision f(*)\n');
CLmax=0;
for i=1:rC
   fprintf(fid,'      f(%1d) =',i);
   column=13+floor(log10(i)); % Tracks the column number of the cursor in the output
   CL=0;                      % Tracks the number of continuation lines
   found=0;
   for j=1:cC
      if C(i,j)
         exponent=floor(log10(abs(C(i,j))));
         mantisse=C(i,j)/(10^exponent);
         if found
            if C(i,j)>=0
               AppStr=' +';
            else
               AppStr=' -';
            end

AppStr=strcat(AppStr,num2str(abs(mantisse),'%1.7f'),'d',int2str(exponent));
               AppStr=strcat(AppStr,' * x(',int2str(j),')');
         else
            AppStr=strcat([' ' num2str(mantisse,'%1.7f')],'d',int2str(exponent));
            AppStr=strcat(AppStr,' * x(',int2str(j),')');
            found=1;
         end
         if (column+length(AppStr))>RgtMar
            fprintf(fid,'\n     +');
            column=6;
            CL=CL+1;
            CLmax=max(CLmax,CL);
         end
         fprintf(fid,AppStr);
         column=column+length(AppStr);
      end
   end
   if l(i)
      if l(i)<0
         AppStr=' -';
      else
         if found
            AppStr=' +';
         else
            AppStr=' ';
         end
      end
      exponent=floor(log10(abs(l(i))));
      mantisse=l(i)/(10^exponent);
      AppStr=strcat(AppStr,num2str(abs(mantisse),'%1.7f'),'d',int2str(exponent));
      if (column+length(AppStr))>RgtMar
         fprintf(fid,'\n     +');
```

```
            column=6;
            CL=CL+1;
            CLmax=max(CLmax,CL);
         end
         fprintf(fid,'%s\n',AppStr);
      else
         if found
            fprintf(fid,'\n');
         else
            if (column+11)>RgtMar
               fprintf(fid,'\n     +');
               column=6;
               CL=CL+1;
               CLmax=max(CLmax,CL);
            end
            fprintf(fid,'%1.7fd0\n',0);
            column=0;
         end
      end
      % Display progress indication
      if etime(clock,t0)>ProgTimeSlice
         disp([spaces num2str(i/(rg1+rg2+5*NoNodes)*100,'%5.1f') '% written.'])
         t0=clock;
      end
   end
   fprintf(fid,'      end\n');
   fprintf(fid,'END.FCN\n\n');

   %Write bounds
   fprintf(fid,'BEGIN.XBOUND\n');
   fprintf(fid,'      subroutine xbound(n,xl,xu)\n');
   fprintf(fid,'      integer n\n');
   fprintf(fid,'      double precision xl(n), xu(n)\n');
   fprintf(fid,'      integer i\n');
   %Lower bound is set to 0. Higher limit is set to infinity.
   fprintf(fid,'      do i=1,n\n');
   fprintf(fid,'      xl(i) = 0.0d0\n');
   fprintf(fid,'      enddo\n');
   fprintf(fid,'      return\n');
   fprintf(fid,'      end\n');
   fprintf(fid,'END.XBOUND\n\n');

   %Write trailer
   fprintf(fid,'N = %1d\n\n',rg1+rg2+5*NoNodes);
   fprintf(fid,'BEGIN.PATHOPT\n');
   fprintf(fid,'END.PATHOPT\n\n');
   fprintf(fid,'BEGIN.COMMENT\n');
   fprintf(fid,'Generation time: %s\n',datestr(now,0));
   fprintf(fid,'Case name: %s\n',ENVIROMENT_NAME);
   fprintf(fid,'END.COMMENT\n\n');
   fprintf(fid,'END-SERVER-INPUT\n');

   fclose(fid);

   disp([spaces 'Output written to file ' ENVIROMENT_NAME Ext]);
   disp([spaces num2str(rC) ' values written.']);
   if CLmax>19
      disp([leader 'WARNING! The output file contains lines with more than 19 continuation
   lines!']);
   end
   disp([leader 'Done']);
   disp([leader 'CPU time    : ' num2str(cputime-ct,'%3.1f') ' seconds']);
   disp([leader 'Elapsed time: ' num2str(etime(clock,tt),'%3.1f') ' seconds']);
   fprintf(1,'\n');
```

Function `lyadown`

Description:
The function `lyadown` generates an LP problem to find a Lyapunov function to prove either stability or instability of the pending position of the pendulum. The LP problem will be written to an output file.

Inputs:

filetype	Defines the file format of the output file. The string *mps* will instruct lyadown to produce an MPS file. The string *lingo* will make the code generate a file compatible with Lingo.
soltype	Type of Lyapunov function to be found. If a string starting with an *s* is supplied an LP is generated to find a Lyapunov function that proves stability. A string starting with an *i* will make the LP-solver to try to prove instability. Default: *s* Optional

Outputs:

An output file containing an LP problem. The format of the output file can be set by means of the input *filetype*.

Depends on: `c2n`, `coord`, `cprt2glb`, `ctrl`, `fitcell`, `mat2ling`, `mat2mps`, `nglb2prt`, `nprt2glb`

Code:

```
function lyadown(filetype,soltype)
%LYADOWN   Tries to find a Lyapunov function
%
%   LYADOWN(filetype,soltype)
%      This function transforms the problem of finding a Lyapunov function
%      to a linear programming problem by defining a triangulation on the
%      state space. The linear programming problem can then be solved by
%      an appropriate solver.
%
%      FILETYPE
%      The variable 'filetype' is a string, which determines the type of
%      file the linear programming problem should be written to. If
%      filetype equals 'mps', the output is written in MPS format, so it
%      can be read by several solvers, among which PCx. If the filetype
%      string equals 'lingo', then a Lingo input file is generated.
%
%      SOLTYPE
%      The parameter 'soltype' determines the type of solution that is
%      being looked for. It can be either a solution to determine
%      stability of the region, denoted by the a string starting with an
%      's' or a solution to determine instability, denoted by a string
%      starting with an 'i'.

tt=clock;                              % timer for total elapsed time

global ENVIROMENT_NAME
if length(ENVIROMENT_NAME)==0
   error('Enviroment name not set. Use SETENV.');
end
if ~exist([ENVIROMENT_NAME '.mat'],'file')
   error('Configuration file does not exist. Use WRITECFG.');
else
   load(ENVIROMENT_NAME);
end

% Constants
ModName       = 'LyaDown';             % Module name (max. 8 characters)
ProgTimeSlice = 60;                    % Number of seconds between progress feedback
leader        = 'Module         : ';   % Leader for output messages
spaces        = '                 ';   % Empty leader

if length(ModName)>8
   ModName=ModName(1:8);
end
leader(8:7+length(ModName))=ModName;

disp([leader 'Calculating LMI matrix...'])

Node0Rows=zeros(3*NoCells,1);
NodeList=zeros(1,NoNodes);
C=sparse(3*NoCells,NoNodes);
t0=clock;
for mPrt=0:NoCells-1
   m=cprt2glb(mPrt);
   % Correct global cell number for the fact that not the neighbourhood
```

```
   % of (0,0) is examined, but the neighbourhood of (pi,0).
   if mod(m,Nh)>=Nh/2
      m=m-Nh;
   else
      m=m+Nh;
   end
   k=c2n(m);
   Node0Rows(mPrt*3+1:mPrt*3+3,1)=(k==Nh/2*ones(1,3))';
   X=coord(k);
   [a,A,b]=fitcell(m);
   Xhat=[X; 1 1 1];
   invXhat=(eye(size(Xhat))/Xhat)';
   u=ctrl(X);
   % Calculate node number for the partial numbering system
   kPrt=zeros(1,3);
   for i=1:3
      RowNo=fix(k(i)/Nh);
      ColNo=mod(k(i),Nh);
      if ColNo>=Nh/2
         kPrt(i)=RowNo*(2*Nhp+1)+ColNo-Nh/2;
      else
         kPrt(i)=RowNo*(2*Nhp+1)+(Nhp+1)+ColNo-(Nh/2-Nhp);
      end
   end

   NodeList(1,[kPrt+1])=k;

   C(mPrt*3+1:mPrt*3+3,kPrt+1)= [ones(3,1)*a'+X'*A'+u'*b' [0;0;0]] * invXhat;

   % Display progress indication
   if etime(clock,t0)>ProgTimeSlice
      disp([spaces num2str(mPrt/(NoCells-1)*100,'%5.1f') '% of LMI matrix is
calculated.'])
      t0=clock;
   end
end

disp([leader 'LMI matrix calculated.'])
disp([leader 'Preparing to export matrix...'])

% Delete rows which contain node number 0
C=C(~Node0Rows,:);

% Delete leftmost column, to force V0 to be zero.
C(:,[1])=[];

% Delete centre node from NodeList
NodeList=NodeList(1,2:NoNodes);

if length(filetype)==3
   if ( (filetype=='mps') | (filetype=='MPS') )
      % Write to MPS file to solve with linear programming problem solver
      disp([leader 'Invoking module Mat2MPS to write data to MPS file'])
      mat2mps([ENVIROMENT_NAME '.mps'],C,NodeList,soltype)
   else
      error([filetype ': unknown file format.'])
   end

elseif length(filetype)==5
   if ( (filetype=='lingo') | (filetype=='LINGO') )
      % Write to Lingo file to solve with external solver
      disp([leader 'Invoking module Mat2Ling to write data to LINGO file'])
      mat2ling([ENVIROMENT_NAME '.lng'],C,NodeList,soltype)
   else
      error([filetype ': unknown file format.'])
   end
else
   error([filetype ': unknown file format.'])
end

disp([leader 'Done.   (' num2str(etime(clock,tt),'%3.1f') ' seconds elapsed)']);
fprintf(1,'\n');
```

Function `lyapunov`

Description:
The function `lyapunov` generates an LP problem to find a Lyapunov function to prove either stability or instability of the inverted position of the pendulum. The LP problem will be written to an output file.

Inputs:

filetype	Defines the file format of the output file. The string *mps* will instruct lyadown to produce an MPS file. The string *lingo* will make the code generate a file compatible with Lingo.
soltype	Type of Lyapunov function to be found. If a string starting with an *s* is supplied an LP is generated to find a Lyapunov function that proves stability. A string starting with an *i* will make the LP-solver to try to prove instability. Default: *s* Optional

Outputs:

An output file containing an LP problem. The format of the output file can be set by means of the input *filetype*.

Depends on: `c2n`, `coord`, `cprt2glb`, `ctrl`, `fitcell`, `mat2ling`, `mat2mps`, `nglb2prt`, `nprt2glb`

Code:

```
function lyapunov(filetype,soltype)
%LYAPUNOV   Tries to find a Lyapunov function
%
%   LYAPUNOV(filetype)
%      This function transfers the problem of finding a Lyapunov function
%      to a linear programming problem by defining a triangulation on the
%      state space. The linear programming problem can then be solved by
%      an appropriate solver.
%
%      FILETYPE
%      The variable 'filetype' is a string, which determines the type of
%      file the linear programming problem should be written to. If
%      filetype equals 'mps', the output is written in MPS format, so it
%      can be read by several solvers, among which PCx. If the filetype
%      string equals 'lingo', then a Lingo input file is generated.

tt=clock;                              % timer for total elapsed time

global ENVIROMENT_NAME
if length(ENVIROMENT_NAME)==0
   error('Enviroment name not set. Use SETENV.');
end
if ~exist([ENVIROMENT_NAME '.mat'],'file')
   error('Configuration file does not exist. Use WRITECFG.');
else
   load(ENVIROMENT_NAME);
end

% Constants
ModName       = 'Lyapunov';            % Module name (max. 8 characters)
ProgTimeSlice = 60;                    % Number of seconds between progress feedback
leader        = 'Module         : ';   % Leader for output messages
spaces        = '                 ';   % Empty leader

if length(ModName)>8
   ModName=ModName(1:8);
end
leader(8:7+length(ModName))=ModName;

disp([leader 'Calculating LMI matrix...'])

Node0Rows=zeros(3*NoCells,1);
NodeList=nprt2glb(1:NoNodes-1);
C=sparse(3*NoCells,NoNodes);
```

```
45    t0=clock;
      for mPrt=0:NoCells-1
         m=cprt2glb(mPrt);
         k=c2n(m);
         Node0Rows(mPrt*3+1:mPrt*3+3,1)=-k';
50       X=coord(k);
         if mod(m,Nh*2)>=2*(Nh-Nhp)
            [a,A,b]=fitcell(m,'neg');
            select=X(1,:)>=(Nh-Nhp-0.5)/Nh*2*pi;
            wrapper=[1; 0]*[1 1 1];
55          X(:,select)= X(:,select) - 2*pi*wrapper(:,select);
         else
            [a,A,b]=fitcell(m);
         end
         Xhat=[X; 1 1 1];
60       invXhat=(eye(size(Xhat))/Xhat)';
         u=ctrl(X);
         C(mPrt*3+1:mPrt*3+3,nglb2prt(k)+1)= [ones(3,1)*a'+X'*A'+u'*b' [0;0;0]] * invXhat;
         % Display progress indication
         if etime(clock,t0)>ProgTimeSlice
65          disp([spaces num2str(mPrt/(NoCells-1)*100,'%5.1f') '% of LMI matrix is
      calculated.'])
            t0=clock;
         end
      end
70
      disp([leader 'LMI matrix calculated.'])
      disp([leader 'Preparing to export matrix...'])

      % Delete rows which contain node number 0
75    C=C(-Node0Rows,:);

      % Delete leftmost column, to force V0 to be zero.
      C(:,[1])=[];

80    if length(filetype)==3
         if ( (filetype=='mps') | (filetype=='MPS') )
            % Write to MPS file to solve with linear programming problem solver
            disp([leader 'Invoking module Mat2MPS to write data to MPS file'])
            mat2mps([ENVIROMENT_NAME '.mps'],C,NodeList,soltype)
85       else
            error([filetype ': unknown file format.'])
         end

      elseif length(filetype)==5
90       if ( (filetype=='lingo') | (filetype=='LINGO') )
            % Write to Lingo file to solve with external solver
            disp([leader 'Invoking module Mat2Ling to write data to LINGO file'])
            mat2ling([ENVIROMENT_NAME '.lng'],C,NodeList,soltype)
         else
95          error([filetype ': unknown file format.'])
         end
      else
         error([filetype ': unknown file format.'])
      end
100
      disp([leader 'Done.    (' num2str(etime(clock,tt),'%3.1f') ' seconds elapsed)']);
      fprintf(1,'\n');
```

Function `lcpsol`

Description:
The function `lcpsol` reads the results returned by the NEOS server.

Inputs:

The file containing the results returned by the NEOS-server. The extension of the file is supposed to be *.sol*.

Outputs:

aplhap	The vector α^+.
alphan	The vector α^-.
V	The vector with values of the piecewise linear Lyapunov function.

x	The vector with the values of the solution of the unknown variable from the LCP.
f	The vector with the function values of the solution of the LCP.

Depends on: —

Code:

```
function [alphap,alphan,V,x,f]=lcpsol()
% LCPSOL  Reads the solution returned by the NEOS server
%
%   LCPSOL
%      Reads the solution to the problem of the swing up of the inverted pendulum as
%      returned by the NEOS server.

global ENVIROMENT_NAME
if length(ENVIROMENT_NAME)==0
   error('Enviroment name not set. Use SETENV.');
end
if ~exist([ENVIROMENT_NAME '.mat'],'file')
   error('Configuration file does not exist. Use WRITECFG.');
else
   load(ENVIROMENT_NAME);
end

% Constants
ModName       = 'LCPSOL';            % Module name (max. 8 characters)
Ext           = '.sol';               % Extension for output file
ProgTimeSlice = 60;                   % Time slice for progression indicator
leader        = 'Module         : ';  % Leader for output messages
spaces        = '                 ';  % Empty leader

if length(ModName)>8
   ModName=ModName(1:8);
end
leader(8:7+length(ModName))=ModName;

fid=fopen([ENVIROMENT_NAME Ext],'r');
if fid<0
   error(['Can not open file ' ENVIROMENT_NAME Ext])
end

line='dummy';
while ~strcmp(line,'***PATH Output***')&~feof(fid)
   line=fgetl(fid);
end

if ~strcmp(line,'***PATH Output***')&feof(fid)
   error('Premature end-of-file detected.');
end

while ~strcmp(line,'SOLUTION FOUND.')&~feof(fid)
   line=fgetl(fid);
end

if ~strcmp(line,'SOLUTION FOUND.')&feof(fid)
   error('The server did not find any solution.');
end

while ~strcmp(line,'COMPLEMENTARY SOLUTION:')&~feof(fid)
   line=fgetl(fid);
end

if ~strcmp(line,'COMPLEMENTARY SOLUTION:')&feof(fid)
   error('Premature end-of-file detected.');
end

disp([leader 'Reading solution from file ' ENVIROMENT_NAME Ext])
t0=clock;   % Initialize clock for progress indication
junk=fscanf(fid,'%c',2);
while (~feof(fid))&strcmp(junk,'x(')
   SeqNr=fscanf(fid,'%d',1);
   junk=fscanf(fid,'%c',1);
   while ~strcmp(junk,':')
      junk=fscanf(fid,'%c',1);
   end
   junk=fscanf(fid,'%c',1);
   x(SeqNr)=fscanf(fid,'%e',1);
   junk=fscanf(fid,'%c',1);
   while ~strcmp(junk,':')
```

```
      junk=fscanf(fid,'%c',1);
   end
   junk=fscanf(fid,'%c',1);
   f(SeqNr)=fscanf(fid,'%e',1);
   line=fgetl(fid);
   junk=fscanf(fid,'%c',2);
   % Display progress indication
   if etime(clock,t0)>ProgTimeSlice
      disp([spaces SeqNr ' values have been read.'])
      t0=clock;
   end
end

alphap = f(1:3*NoNodes);
alphan = x(1:3*NoNodes);
V      = x(3*NoNodes+1:4*NoNodes);

fclose(fid);
disp([leader 'Done.'])
```

Function `mat2ling`

Description:
The function `mat2ling` is used by the functions `lyadown` and `lyapunov` to do the actual writing to a Lingo file.

Inputs:

LINGOname	The file name of the output file.
C	The matrix containing the LP problem.
NodeList	A list with the (global) numbers of the nodes.
Soltype	Defines if the solver has to find a Lyapunov function to prove either stability or instability. A string beginning with *s* makes the solver look for stability; a string starting with *i* forces the solver to attempt to prove instability.

Outputs:

A Lingo file containing the LP problem representing the problem of finding an appropriate Lyapunov function.

Depends on: —

Code:

```
function mat2ling(LINGOname,C,NodeList,soltype)
% MAT2LING  Write matrix to LINGO file to solve with linear programming tool

% Constants
ModName       = 'Mat2Ling';           % Module name (max. 8 characters)
ProgTimeSlice = 60;                   % Number of seconds between progress feedback
upbound       = 250;                  % Boundary for values of Lyapunov function
leader        = 'Module         : ';  % Leader for output messages
spaces        = '                 ';  % Empty leader

if length(ModName)>8
   ModName=ModName(1:8);
end
leader(8:7+length(ModName))=ModName;

if soltype(1)=='s'
   % Find Lyapunov function to proof stability
   lowbound=0;
else
   % Find Lyapunov function to proof instability
   lowbound=-upbound;
end

[rows,nodes]=size(C);

% Open file
fid=fopen(LINGOname, 'w');
if fid==-1
    error(['Unable to open file ' LINGOname])
```

```
30    end

      disp([leader 'Writing to LINGO file ' LINGOname ' ...'])

      % Write header
35    fprintf(fid,['MODEL:\r\n']);

      % Write cost equation
      fprintf(fid,'   [COST]  max = z;\r\n');

40    disp([leader '   Writing variable bounds'])

      % Write bounds
      t0=clock;
      fprintf(fid,'   @BND(0,z,1);\r\n');
45    for k=1:nodes
         fprintf(fid,['   @BND(' num2str(lowbound) ',V' num2str(NodeList(k))  ','
      num2str(upbound) ');\r\n']);
         % Display progress indication
         if etime(clock,t0)>ProgTimeSlice
50          disp([spaces '   ' num2str(k/nodes*100,'%5.1f') '% of bounds is written.'])
            t0=clock;
         end
      end

55    disp([leader '   Writing inequalities'])

      % Write inequalities
      t0=clock;
      for mm=1:rows
60       found=0;
         for k=1:nodes
            if C(mm,k)
               if found
                  if C(mm,k)<0
65                   fprintf(fid,' - ');
                  else
                     fprintf(fid,' + ');
                  end
               else
70                fprintf(fid,['   [R' num2str(mm) ']  ']);
                  if C(mm,k)<0
                     fprintf(fid,'-');
                  end
                  found=1;
75             end
               fprintf(fid,[num2str(abs(C(mm,k))) '*V' num2str(NodeList(k))]);
            end
         end
         if found
80          fprintf(fid,' + z < 0;\r\n');
         end
         % Display progress indication
         if etime(clock,t0)>ProgTimeSlice
            disp([spaces '   ' num2str(mm/rows*100,'%5.1f') '% of inequalities is written.'])
85          t0=clock;
         end
      end
      fprintf(fid,'END');

90    % Close data file !!
      fclose(fid);

      disp([leader 'Data written to LINGO file ' LINGOname '.'])
```

Function `mat2mps`

Description:
The function `mat2mps` is used by the functions `lyadown` and `lyapunov` to do the actual writing to a MPS file.

Inputs:

MPSname The file name of the output file.

C	The matrix containing the LP problem.
NodeList	A list with the (global) numbers of the nodes.
Soltype	Defines if the solver has to find a Lyapunov function to prove either stability or instability. A string beginning with *s* makes the solver look for stability; a string starting with *i* forces the solver to attempt to prove instability.

Outputs:

A MPS file containing the LP problem representing the problem of finding an appropriate Lyapunov function.

Depends on: —

Code:

```
function mat2mps(MPSname,C,NodeList,soltype)
% MAT2MPS  Write matrix to MPS file to solve with linear programming tool

% Constants
ModName       = 'Mat2MPS';            % Module name (max. 8 characters)
ProgTimeSlice = 60;                   % Number of seconds between progress feedback
upbound       = 100000;               % Boundary for values of Lyapunov function
leader        = 'Module         : ';  % Leader for output messages
spaces        = '                 ';  % Empty leader

if length(ModName)>8
   ModName=ModName(1:8);
end
leader(8:7+length(ModName))=ModName;

if soltype(1)=='s'
   % Find Lyapunov function to proof stability
   lowbound=0;
else
   % Find Lyapunov function to proof instability
   lowbound=-upbound;
end

[rows,nodes]=size(C);

% Open file
fid=fopen(MPSname,'w');
if fid==-1
    error('Unable to open file')
end

disp([leader 'Writing to MPS file ' MPSname ' ...'])

% Write NAME section
s=blanks(14);
s(1:4)='NAME';
fprintf(fid,[s 'Find Lyapunov function\n']);

disp([leader 'Writing ROWS section'])

% Write ROWS section
t0=clock;
fprintf(fid,'ROWS\n');
for mm=1:rows
   fprintf(fid,[' L  RN' num2str(mm,6) '\n']);
   % Display progress indication
   if etime(clock,t0)>ProgTimeSlice
      disp([spaces num2str(mm/rows*100,'%5.1f') '% of rows written.'])
      t0=clock;
   end
end
fprintf(fid,[' N  COST\n']);

disp([leader 'Writing COLUMNS section'])

% Write COLUMNS section
t0=clock;
fprintf(fid,'COLUMNS\n');
for k=1:nodes
   for mm=1:rows
      if C(mm,k)
         clab=['V' num2str(NodeList(k),7)];
         rlab=['RN' num2str(mm,6)];
```

```
            s=blanks(24);
            s(5:4+length(clab))=clab;
            s(15:14+length(rlab))=rlab;
            fprintf(fid,[s '%1.8f\n'],C(mm,k));
         end
      end
      % Display progress indication
      if etime(clock,t0)>ProgTimeSlice
         disp([spaces num2str(k/nodes*100,'%5.1f') '% of columns written.'])
         t0=clock;
      end
   end
   for mm=1:rows
      clab=['Z'];
      rlab=['RN' num2str(mm,6)];
      s=blanks(24);
      s(5:4+length(clab))=clab;
      s(15:14+length(rlab))=rlab;
      fprintf(fid,[s '%1.8f\n'],1);
   end
   clab=['Z'];
   s=blanks(24);
   s(5:4+length(clab))=clab;
   s(15:18)='COST';
   fprintf(fid,[s '%1.8f\n'],-1);

   disp([leader 'Writing RHS section'])

   % Write RHS section
   t0=clock;
   fprintf(fid,'RHS\n');
   for mm=1:rows
       s=blanks(24);
       s(5)='B';
       rlab=['RN' num2str(mm,6)];
       s(15:14+length(rlab))=rlab;
       fprintf(fid,[s num2str(0.0,'%1.8f') '\n']);
      % Display progress indication
      if etime(clock,t0)>ProgTimeSlice
         disp([spaces num2str(mm/rows*100,'%5.1f') '% of RHS section is written.'])
         t0=clock;
      end
   end

   disp([leader 'Writing BOUNDS section'])

   % Write BOUNDS section
   t0=clock;
   fprintf(fid,'BOUNDS\n');
   for k=1:nodes
      clab=['V' num2str(NodeList(k),7)];
      s=blanks(24);
      s(2:3)='LO';
      s(5:9)='BOUND';
      s(15:14+length(clab))=clab;
      fprintf(fid,[s '%1.8f\n'],lowbound);
      s(2:3)='UP';
      fprintf(fid,[s '%1.8f\n'],upbound);
      % Display progress indication
      if etime(clock,t0)>ProgTimeSlice
         disp([spaces num2str(k/nodes*100,'%5.1f') '% of bounds is written.'])
         t0=clock;
      end
   end
      s=blanks(24);
      s(2:3)='LO';
      s(5:9)='BOUND';
      s(15)='Z';
      fprintf(fid,[s '%1.8f\n'],0);
      s(2:3)='UP';
      fprintf(fid,[s '%1.8f\n'],1);

   % Write ENDATA
   fprintf(fid,'ENDATA\n');

   % Close data file !!
   fclose(fid);

   disp([leader 'Data written to MPS file ' MPSname '.'])
```

Function `n2c`

Description:
The function `n2c` returns a row vector with the numbers of the cell that are bordering the node designated by input *k*.

Inputs:

k	The node number

Outputs:

m	Row vector with the cell numbers

Depends on: `odd, even`

Code:

```
function m=n2c(k)
%NODE2CELLS  Calculates the numbers of the cells encompassing a node
%
%   m=N2C(k)
%      Returns a row vector m containing all the cells that
%      encompass node k.

global ENVIROMENT_NAME
load(ENVIROMENT_NAME);

RowNo = fix(k/Nh);

% Calculate node number of the node below the current one
if RowNo==0
   NodeBelow=k+Nh;
elseif odd(RowNo)
   NodeBelow=k+2*Nh;
else
   NodeBelow=k-2*Nh;
end

% Calculate the numbers of the nodes directly to the left of the
% current node and the node below this one

NodeLeft=k-1;
if mod(NodeLeft,Nh)==Nh-1
   NodeLeft=NodeLeft+Nh;
end

NodeBelowLeft=NodeBelow-1;
if mod(NodeBelowLeft,Nh)==Nh-1
   NodeBelowLeft=NodeBelowLeft+Nh;
end

% Calculate the number of cells around node 'k'
sect  = even(fix((k+Nh)/Nh/2));
eight = (odd(k) & (~sect)) | ((~odd(k)) & sect);

% from this information we can calculate the number of the
% adjacent cells

m=[k*2 NodeLeft*2+1 NodeBelow*2 NodeBelowLeft*2+1];

if eight
   m=[m k*2+1 NodeLeft*2 NodeBelow*2+1 NodeBelowLeft*2];
end
m=sort(m);
```

Function `nglb2prt`

Description:
The function `nglb2prt` converts node numbers according to the numbering system of the global triangulation to the numbering of a partial triangulation.

Inputs:

kGlb	A row vector with (global) node numbers to be converted to partial numbering.

Outputs:

kPrt	A row vector with (partial) cell numbers.

Depends on: —

Code:

```
function kPrt=nglb2prt(kGlb)
%NGLB2PRT  Maps node numbers from a global to a partial triangulation
%
%   kPrt=NGLB2PRT(kGlb)
%      The function accepts a node number of a global triangulation and
%      calcutates the number of the node in the partial triangulation. If
%      the node number is not a part of the partial triangulation, a value
%      of -1 will be returned.

global ENVIROMENT_NAME
load(ENVIROMENT_NAME);

kPrt=zeros(1,length(kGlb));
for index=1:length(kGlb)
   RowNo=fix(kGlb(index)/Nh);
   ColNo=mod(kGlb(index),Nh);
   if ColNo<=Nhp
      kPrt(index)=RowNo*(2*Nhp+1)+ColNo;
   elseif ColNo>=(Nh-Nhp)
      kPrt(index)=(RowNo+1)*(2*Nhp+1)+(ColNo-Nh);
   else
      kPrt(index)=-1;
   end
end
```

Function `nprt2glb`

Description:
The function `nprt2glb` converts node numbers according to the numbering system of a partial triangulation to the numbering of the global triangulation.

Inputs:

kPrt	A row vector with (partial) node numbers to be converted to global numbering.

Outputs:

kGlb	A row vector with (global) node numbers.

Depends on: —

Code:

```
function kGlb=nprt2glb(kPrt)
%NPRT2GLB  Maps node numbers from a partial to a global triangulation
%
%   kGlb=NPRT2GLB(kPrt)
%      The function accepts a node number of a partial triangulation and
%      calcutates the number of the node in the global triangulation

global ENVIROMENT_NAME
```

```
load(ENVIROMENT_NAME);

kGlb=zeros(1,length(kPrt));
for index=1:length(kPrt)
   RowNo=fix(kPrt(index)/(2*Nhp+1));
   ColNo=mod(kPrt(index),2*Nhp+1);
   if ColNo<=Nhp
      kGlb(index)=RowNo*Nh+ColNo;
   else
      kGlb(index)=(RowNo+1)*Nh+(ColNo-(2*Nhp+1));
   end
end
```

Function `plc`

Description:
The function `plc` calculates the response of the piecewise linear controller in a point.

Inputs:

x	Co-ordinates of the point for which the output of the piecewise linear controller should be calculated

Outputs:

u	Output of the piecewise linear controller for input *x*

Depends on: `barrycnt`, `coord`, `ctrl`

Code:

```
function u=plc(x)
%PLC  Calculate the response of the piecewise linear controller in a point
%
%   u=PLC(x)
%      The response of the piecewise linear controller for the input defined
%      by the co-ordinates 'x'.

global ENVIROMENT_NAME
if length(ENVIROMENT_NAME)==0
   error('Enviroment name not set. Use SETENV.');
end
if ~exist([ENVIROMENT_NAME '.mat'],'file')
   error('Configuration file does not exist. Use WRITECFG.');
else
   load(ENVIROMENT_NAME);
end

if length(x)~=2
   error('Input must have exactly 2 rows');
end

[m,k,a]=barrycnt(x);
xk=coord(k);
uk=ctrl(xk);
u=uk*a;
```

Function `plotlya`

Description:
The function `plotlya` plots the piecewise linear Lyapunov function.

Inputs:

V	A column vector with the values of the Lyapunov function in the nodes.
NodeList	A column vector with the numbers of the nodes.
TitleStr	A string with an optional title for the graph Default: *Lyapunov function* Optional

Outputs:

A 3-dimensional graph showing the Lyapunov function.

Depends on: `c2n`, `coord`, `cprt2glb`, `nglb2prt`

Code:

```
function plotlya(V,NodeList,TitleStr)
%PLOTLYA  Plots a piecewise linear Lyapunov function
%
%   PLOTLYA(V,NodeList,TitleStr)
%      Plots the piecewise linear Lyapunov function defined in vector V.
%      NodeList contains a list with the global numbers of the nodes and
%      the optional string TitleStr can be used to give the plot a title.

tt=clock;                              % timer for total elapsed time

global ENVIROMENT_NAME
if length(ENVIROMENT_NAME)==0
   error('Enviroment name not set. Use SETENV.');
end
if ~exist([ENVIROMENT_NAME '.mat'],'file')
   error('Configuration file does not exist. Use WRITECFG.');
else
   load(ENVIROMENT_NAME);
end

% Constants
ModName       = 'PlotLya';             % Module name (max. 8 characters)
ProgTimeSlice = 60;                    % Number of seconds between progress feedback
leader        = 'Module         : ';   % Leader for output messages
spaces        = '                 ';   % Empty leader

if length(ModName)>8
   ModName=ModName(1:8);
end
leader(8:7+length(ModName))=ModName;

if NodeList(1)==0
   % Examining around the inverted position
   EquiPnt=0;
else
   % Examining around the pending position
   EquiPnt=pi;
end

disp([leader 'Initialising data...']);
tri=zeros(NoCells,3);
x=zeros(NoNodes,1);
y=zeros(NoNodes,1);
z=zeros(NoNodes,1);

disp([leader 'Calculating x- and y-coordinates...']);
t0=clock;
for kPrt=0:NoNodes-1
   kGlb=NodeList(kPrt+1);
   X=coord(kGlb);
   if EquiPnt==0
      % Examining around the inverted position so apply wrapping
      select=X(1,:)>=(Nh-Nhp-0.5)/Nh*2*pi;
      wrapper=[1; 0]*[1 1 1];
      X(:,select)= X(:,select) - 2*pi*wrapper(:,select);
   end
   x(kPrt+1)=X(1);
   y(kPrt+1)=X(2);
   % Display progress indication
   if etime(clock,t0)>ProgTimeSlice
      disp([spaces '   ' num2str(kPrt/NoNodes*100,'%5.1f') '% of x- and y-coordinates
calculated.'])
      t0=clock;
   end
end

disp([leader 'Calculating surface...'])
t0=clock;
for mPrt=0:NoCells-1
   m=cprt2glb(mPrt);
   if EquiPnt~=0
      % Correct global cell number for the fact that not the neighbourhood
      % of (0,0) is examined, but the neighbourhood of (pi,0).
```

```
        if mod(m,Nh)>=Nh/2
            m=m-Nh;
        else
            m=m+Nh;
        end
        k=c2n(m);
        % Calculate node number for the partial numbering system
        kPrt=zeros(1,3);
        for i=1:3
            RowNo=fix(k(i)/Nh);
            ColNo=mod(k(i),Nh);
            if ColNo>=Nh/2
                kPrt(i)=RowNo*(2*Nhp+1)+ColNo-Nh/2;
            else
                kPrt(i)=RowNo*(2*Nhp+1)+(Nhp+1)+ColNo-(Nh/2-Nhp);
            end
        end
        tri(mPrt+1,:)=kPrt+1;
    else
        k=c2n(m);
        tri(mPrt+1,:)=nglb2prt(k)+1;
    end
    % Display progress indication
    if etime(clock,t0)>ProgTimeSlice
        disp([spaces '    ' num2str(mPrt/NoCells*100,'%5.1f') '% calculated.'])
        t0=clock;
    end
end

disp([leader 'Displaying surface...'])
colormap([linspace(0.3,1,32)'*[1 1 1]]);
trisurf(tri,x,y,V');
xlabel('\theta');
ylabel('\theta''');
if nargin<3
    title('Lyapunov function');
else
    title(TitleStr);
end
rotate3d;

disp([leader 'Done.    (' num2str(etime(clock,tt),'%3.1f') ' seconds elapsed)']);
fprintf(1,'\n');
```

Function odd

Description:
The function odd returns a non-zero value if the integer part of the input value is odd, otherwise it returns a zero.

Inputs:

number	The number to be tested.

Outputs:

out	A non-zero number if the integer part of the input is odd, otherwise 0.

Depends on: —

Code:

```
function out=odd(number)
%ODD  Returns TRUE if the integer part of its argument is odd
out=mod(fix(number),2);
```

Function `readling`

Description:
The function `readling` reads the results from Lingo and puts the values of the Lyapunov function in a vector.

Inputs:

The output file of Lingo containing the solution to the LP problem. The extension of the file is supposed to be *.lou* .

Outputs:

V	A column vector with the values of the Lyapunov function in the nodes.
NodeList	A column vector with the numbers of the nodes.

Depends on: —

Code:

```
function [V,NodeList]=readling()
% READLING  Reads the output generated by LINGO

global ENVIROMENT_NAME
if length(ENVIROMENT_NAME)==0
   error('Enviroment name not set. Use SETENV.');
end
if ~exist([ENVIROMENT_NAME '.mat'],'file')
   error('Configuration file does not exist. Use WRITECFG.');
else
   load(ENVIROMENT_NAME);
end

% Constants
Ext           = '.lou';
ModName       = 'ReadLing';           % Module name (max. 8 characters)
ProgTimeSlice = 60;                   % Number of seconds between progress feedback
leader        = 'Module         : ';  % Leader for output messages
spaces        = '                 ';  % Empty leader

fid=fopen([ENVIROMENT_NAME Ext],'r');
if fid<0
   error(['Can not open file ' ENVIROMENT_NAME Ext])
end

% Skip header lines
while (~strcmp(fscanf(fid,'%c',2),'  '))&(~feof(fid))
   fgetl(fid);
end
fgetl(fid);

index=2;
NodeList=[0];
V=[0];
Nneg=0;
Nzero=0;
Npos=0;
MinV=0;
MaxV=0;
disp([leader 'Reading LINGO output file ' ENVIROMENT_NAME Ext])
t0=clock;
while (~feof(fid))&(fscanf(fid,'%c',1)==' ')
   charact=fscanf(fid,'%c',1);
   while strcmp(charact,' ')
      charact=fscanf(fid,'%c',1);
   end
   if charact~='Z'
      NodeList(index)=fscanf(fid,'%d',1);
      V(index)=fscanf(fid,'%e',1);
      if V(index)<0
         Nneg=Nneg+1;
      elseif V(index)==0
         Nzero=Nzero+1;
      else
         Npos=Npos+1;
      end
```

```
      MinV=min(MinV,V(index));
      MaxV=max(MaxV,V(index));
   end
   line=fgetl(fid);
   % Display progress indication
   if etime(clock,t0)>ProgTimeSlice
      disp([spaces num2str(index/(NoNodes-1)*100,'%5.1f') '% read.'])
      t0=clock;
   end
   index=index+1;
end

% Correcting global number of first node if examining neighbourhood of pi.
if length(NodeList)>1
   if NodeList(2)~=1
      NodeList(1)=Nh/2;
   end
end

fclose(fid);

disp([leader 'Statistics:']);
disp([spaces num2str(length(V)) ' values read from file ' ENVIROMENT_NAME Ext]);
disp([spaces num2str(Nneg) ' negative values with minimum ' num2str(MinV)]);
disp([spaces num2str(Nzero) ' zeros']);
disp([spaces num2str(Npos) ' positive values with maximum ' num2str(MaxV)]);
disp([leader 'Done.']);
fprintf(1,'\n');
```

Function `readmps`

Description:
The function `readmps` reads the results from PCx and puts the values of the Lyapunov function in a vector.

Inputs:

The output file of PCx containing the solution to the LP problem.

Outputs:

V	A column vector with the values of the Lyapunov function in the nodes.
NodeList	A column vector with the numbers of the nodes.

Depends on: —

Code:

```
function [V,NodeList]=readmps()
% READMPS  Reads the output generated by PCx
%
% [V,NodeList]=READMPS()
%   Reads the solution provided by the LP solver PCx

global ENVIROMENT_NAME
if length(ENVIROMENT_NAME)==0
   error('Enviroment name not set. Use SETENV.');
end
if ~exist([ENVIROMENT_NAME '.mat'],'file')
   error('Configuration file does not exist. Use WRITECFG.');
else
   load(ENVIROMENT_NAME);
end

% Constants
Ext           = '.out';
ModName       = 'ReadMPS';            % Module name (max. 8 characters)
ProgTimeSlice = 60;                   % Number of seconds between progress feedback
leader        = 'Module         : ';  % Leader for output messages
spaces        = '                 ';  % Empty leader

if length(ModName)>8
   ModName=ModName(1:8);
end
leader(8:7+length(ModName))=ModName;
```

```
fid=fopen([ENVIROMENT_NAME Ext],'r');
if fid<0
   error(['Can not open file ' ENVIROMENT_NAME Ext])
end

% Skip first three lines
line=fgetl(fid);
line=fgetl(fid);
line=fgetl(fid);

index=2;
NodeList=[0];
V=[0];
Nneg=0;
Nzero=0;
Npos=0;
MinV=0;
MaxV=0;
disp([leader 'Reading PCx output file ' ENVIROMENT_NAME Ext])
done=0;
t0=clock;
while (~feof(fid))&~done
   SeqNr=fscanf(fid,'%d',1);
   spaces=fscanf(fid,'%c',2);
   if fscanf(fid,'%c',1)~='Z'
      NodeList(index)=fscanf(fid,'%d',1);
      V(index)=fscanf(fid,'%e',1);
      if V(index)<0
         Nneg=Nneg+1;
      elseif V(index)==0
         Nzero=Nzero+1;
      else
         Npos=Npos+1;
      end
      MinV=min(MinV,V(index));
      MaxV=max(MaxV,V(index));
      index=index+1;
   else
      done=1;
   end
   line=fgetl(fid);
   % Display progress indication
   if etime(clock,t0)>ProgTimeSlice
      disp([spaces num2str(index/(NoNodes-1)*100,'%5.1f') '% read.'])
      t0=clock;
   end
end
% Correcting global number of first node if examining neighbourhood of pi.
if length(NodeList)>1
   if NodeList(2)~=1
      NodeList(1)=Nh/2;
   end
end

fclose(fid);

disp([leader 'Statistics:']);
disp([spaces num2str(length(V)) ' values read from file ' ENVIROMENT_NAME Ext]);
disp([spaces num2str(Nneg) ' negative values with minimum ' num2str(MinV)]);
disp([spaces num2str(Nzero) ' zeros']);
disp([spaces num2str(Npos) ' positive values with maximum ' num2str(MaxV)]);
disp([leader 'Done.']);
fprintf(1,'\n');
```

Function `setenv`

Description:

The function `setenv` sets the global variable ENVIROMENT_NAME to the name of the configuration that should be used by all MatLab functions. It should be invoked before any of the other functions is invoked. The name of the configuration is used to store a file with global variables that determine the configuration and most names of files used by the MatLab functions will be derived from the configuration name.

Inputs:

envname A string with the name of the configuration that should be used.

Outputs:

out A non-zero number if the integer part of the input is odd, otherwise 0.

Depends on: —

Code:

```
function setenv(envname)
%SETENV  Sets the name of the enviroment to use for the calculations
%
%   SETENV(envname)
%      Stores the name of the enviroment in the global variable
%      ENVIROMENT_NAME. This name is used to retrieve a set of
%      global variables from a configuration file, which names
%      is constructed from the enviroment name.

% Constants
DefaultPath='c:\Home\Groen\triang\';  % Set default path
PathDefined=0;

envlength=length(envname);
if envlength>=2
   % Enviroment name starts with drive specification
   PathDefined=PathDefined|(envname(2)==':');
end

i=1;
while (i<=envlength)&(~PathDefined)
   PathDefined=PathDefined|(envname(i)=='/')|(envname(i)=='\');
   i=i+1;
end

%Strip extension if necessary
Stripped=0;
i=max(1,envlength-3);
while (i<=envlength)&~Stripped
   if envname(i)=='.'
      if i==envlength
         envname=envname(1:envlength-1);
         stripped=1;
      else
         if (envname(min(i+1,envlength))=='/')|(envname(min(i+1,envlength))=='\') ...
            |(envname(min(i+2,envlength))=='/')|(envname(min(i+2,envlength))=='\')
            envname=envname(1:max(1,i-1));
            Stripped=1;
         end
      end
   end
   i=i+1;
end

if ~PathDefined
   envname=fullfile(DefaultPath,envname);
end

if exist('ENVIROMENT_NAME','var')
   clear global ENVIROMENT_NAME;
end

global ENVIROMENT_NAME;
ENVIROMENT_NAME=envname;

if exist([ENVIROMENT_NAME '.mat'],'file')
   disp(['Configuration file present.']);
else
   disp(['Configuration file does not exist.']);
end
fprintf(1,'\n');
```

Function `testfit`

Description:
The function `testfit` calculates the differences between the output of the differential equations and the piecewise affine approximation to give an idea of the accurateness of the approximation. Either the total triangulation is examined or the neighbourhood of the inverted position. The last case is true if in `writecfg` a non-zero number is entered for the number of horizontal nodes to examine. If a zero was entered, `testfit` examines the total triangulation.

Inputs:

NoRndPoints	The number of random samples to examine in each cell. Default: 3 Optional
PermRelErr	The minimum relative deviation between the value of the differential equations and the approximation before the approximation is considered to be erroneous. Default: 2 (%) Optional
SmallNmbr	Numbers smaller than this parameter are considered to be zero. This parameter prevents a relative error to become infinitely large if the approximation is not exactly zero and the differential equations are. Default: 10^{-14} Optional

Outputs:

An output file with the results for each cell and a summary on the terminal.

Depends on: `c2n`, `coord`, `cprt2glb`, `ctrl`, `diffeq`, `fitcell`, `nglb2prt`

Code:

```
function testfit(NoRndPoints,PermRelErr,SmallNmbr)
%TESTFIT  Tests if the fit of the triangulation is adequate
%
%   TESTFIT(n,PermRelErr,SmallNmbr)
%      This module compares the values the fitted process with the values
%      produced by the differential equations and outputs the differences
%      to a file (extension .tst). The parameter 'n' is optional and
%      represents the number of random samples to be calculated in each
%      cell. The parameters 'PermRelErr' and 'SmallNmbr' are also optional
%      and define the relative error that is permitted to occur and a
%      limit that defines when a small number is considered to be equal to
%      zero. The module will produce a summary of the test on the terminal.

tt=clock;                              % timer for total elapsed time

global ENVIROMENT_NAME
if length(ENVIROMENT_NAME)==0
   error('Enviroment name not set. Use SETENV.');
end
if ~exist([ENVIROMENT_NAME '.mat'],'file')
   error('Configuration file does not exist. Use WRITECFG.');
else
   load(ENVIROMENT_NAME);
end

% Constants
ModName       = 'TestFit';             % Module name (max. 8 characters)
ProgTimeSlice = 60;                    % Number of seconds between progress feedback
leader        = 'Module         : ';   % Leader for output messages
spaces        = '                 ';   % Empty leader
ext           = '.tst';                % Extension for the output file

if length(ModName)>8
   ModName=ModName(1:8);
end
leader(8:7+length(ModName))=ModName;
```

```
if nargin<1
   NoRndPoints = 3;      % Number of random points to be examined
else
   NoRndPoints = max(0,fix(NoRndPoints));
end
if nargin<2
   PermRelErr  = 2;      % Permitted relative error (used for summary)
else
   PermRelErr  = abs(PermRelErr);
end
if nargin<3
   SmallNmbr   = 1e-14;  % Smaller numbers are considered 0 for summary
else
   SmallNmbr   = abs(SmallNmbr);
end

% fid=1;  % Send output to Standard Output
fid=fopen([ENVIROMENT_NAME ext],'w');
if fid ==-1
   error('Could not open output file')
end

disp([leader 'Testing quality of fit...']);

% Initialize values for summary report
  % All errors regardless of the occurence of near-zero errors
   NoErr=0;          % Initialize maximum error
   OutOfRng=0;       % Number of errors out of permitted range
   SgnErr=0;         % Number of wrong signs
   % Errors corrected for the occurence of near-zero errors
   NoErrCor=0;       % Initialize maximum error
   OutOfRngCor=0;    % Number of errors out of permitted range
   SgnErrCor=0;      % Number of wrong signs

%Initialise the counters for the errors
%  The matrix ErrUn contains the counters for all errors regardless
%  the presence of near-zero errors. The matrix ErrCr contains the
%  same counters, but is corrected for the occurrence of near-zero
%  errors. The first row contains the counters applying to the
%  results of the vaue of the first DE, the second row the counters
%  for the second DE. There are separate counters for points that
%  are nodes and for random points.
%  The columns contain the next values:
%     1 nodes:     # samples
%     2            comm. of abs. value of error
%     3            comm. of square of error
%     4            max. of error
%     5            # errors  exceeding threshold
%     6            # relative errors = undefined
%     7 non-nodes: # samples
%     8            comm. of abs. value of error
%     9            comm. of square of error
%    10            max. of error
%    11            # errors exceeding threshold
%    12            # relative errors = undefined
%    13 # wrong signs
ErrUn=zeros(2,13);
ErrCr=zeros(2,13);

t0=clock;   % Initialize clock for progress indication
for mPrt=0:NoCells-1
   if Nhp==0
      mGlb=mPrt;
   else
      mGlb=cprt2glb(mPrt);
   end
   fprintf(fid,'\nCell number: %d (global: %d)\n',mPrt,mGlb);
   [a,A,b]=fitcell(mGlb);
   fprintf(fid,'   Fit parameters:\n');
   fprintf(fid,'      a[1]  =%14d,   a[2]  =%14d\n',a(1),a(2));
   fprintf(fid,'      A[1,1]=%14d,   A[1,2]=%14d\n',A(1,1),A(1,2));
   fprintf(fid,'      A[2,1]=%14d,   A[2,2]=%14d\n',A(2,1),A(2,2));
   fprintf(fid,'      b[1]  =%14d,   b[2]  =%14d\n',b(1),b(2));
   kGlb=c2n(mGlb);
   kPrt=nglb2prt(kGlb);
   fprintf(fid,'   Node numbers:\n');
   fprintf(fid,'      %6d, %6d, %6d ',kPrt(1),kPrt(2),kPrt(3));
   fprintf(fid,'(global: %d, %d, %d\n',kGlb(1),kGlb(2),kGlb(3));
   x=coord(kGlb);
   % Adjust coordinates if cell is in the last column
```

```
   if mod(fix(mGlb/2),Nh)==Nh-1
      x(1,(abs(x(1,:))<(pi/Nh)))= 2*pi;
   end
   fprintf(fid,'   Node coordinates:\n');
   fprintf(fid,'      %14d, %14d, %14d\n',x(1,1),x(1,2),x(1,3));
   fprintf(fid,'      %14d, %14d, %14d\n',x(2,1),x(2,2),x(2,3));
   rndbarry=rand(3,NoRndPoints);
   rndbarry=rndbarry./(ones(3,3)*rndbarry);
   randpoints=x*rndbarry;
   fprintf(fid,'   Coordinates of random points:\n');
   if NoRndPoints>0
      fprintf(fid,'      %14d',randpoints(1,1));
      for i=2:NoRndPoints
         fprintf(fid,', %14d',randpoints(1,i));
      end
      fprintf(fid,'\n');
      fprintf(fid,'      %14d',randpoints(2,1));
      for i=2:NoRndPoints
         fprintf(fid,', %14d',randpoints(2,i));
      end
      fprintf(fid,'\n');
   end
   fprintf(fid,'   Values of the fitted model and the differential equations in the
above points:\n');
   fprintf(fid,'              Fitted model          |    Differential equation    |
Absolute error (Fit-Dif)   |   Rel.error (%%)   \n');
   fprintf(fid,'        value 1    |   value 2    |   value 1    |   value 2    |
value 1    |   value 2    | value 1 | value 2 \n');
   fprintf(fid,'      ------------+-------------+-------------+-------------+-------
-------+-------------+---------+---------\n');
   x=[x randpoints];
   for i=1:3+NoRndPoints
      if i<=3
         % The piecewise linear approximation of the controller will be exact
         % in the nodes, so call the controller directly instead of the
         % approximation to prevent for errors introduced by MatLab
         u=ctrl(x(:,i));
      else
         % Calculate value of piecewise linear controller
         u=plc(x(:,i));
      end
      y=a+A*x(:,i)+b*u;
      [f1x,g1x,f2x,g2x]=diffeq(x(:,i));
      z(1)=f1x+g1x*u;
      z(2)=f2x+g2x*u;
      fprintf(fid,'      %12.4e | %12.4e | %12.4e | %12.4e | ',y(1),y(2),z(1),z(2));
      fprintf(fid,'%12.4e | %12.4e | ',y(1)-z(1),y(2)-z(2));
      if i<=3  % Is point a node ?
         os  = 0; % set offset for uncorrected values
         os2 = 0; % ditto for corrected vaues
      else
         os  = 6; % ditto
         os2 = 3; % ditto
      end
      for j=1:2
         if z(j)
            % Value of DE is non-zero
            RelErr=y(j)/z(j)-1;
            ErrUn(j,1+os)=ErrUn(j,1+os)+1;                % # samples
            ErrUn(j,2+os)=ErrUn(j,2+os)+abs(RelErr);      % comm. abs. value of error
            ErrUn(j,3+os)=ErrUn(j,3+os)+RelErr*RelErr;    % comm. of squares
            ErrUn(j,4+os)=max(ErrUn(j,4+os),RelErr);      % maximum value
            if (y(j)>SmallNmbr)|(z(j)>SmallNmbr)
               ErrCr(j,1+os)=ErrCr(j,1+os)+1;
               ErrCr(j,2+os)=ErrCr(j,2+os)+abs(RelErr);
               ErrCr(j,3+os)=ErrCr(j,3+os)+RelErr*RelErr;
               ErrCr(j,4+os)=max(ErrCr(j,4+os),RelErr);
            end
            if abs(RelErr)*100>PermRelErr
               ErrUn(j,5+os)=ErrUn(j,5+os)+1;             % # errors exceeding limit
               if (y(j)>SmallNmbr)|(z(j)>SmallNmbr)
                  ErrCr(j,5+os)=ErrCr(j,5+os)+1;
               end
            end
            if j==1
               fprintf(fid,'%7.2f | ',RelErr*100);
            else
               fprintf(fid,'%7.2f\n',RelErr*100);
            end
         else
            % Value of DE is zero
```

```
            if y(j)
200            % Value of fit is non-zero ->
               % Relative error is undefined
               ErrUn(j,6+os)=ErrUn(j,6+os)+1;
               if y(j)>SmallNmbr
                  ErrCr(j,6+os)=ErrCr(j,6+os)+1;
205            end
               if j==1
                  fprintf(fid,'        | ');
               else
                  fprintf(fid,'\n');
210            end
            else
               % Value of fit is zero ->
               % Relative error is zero
               ErrUn(j,1+os)=ErrUn(j,1+os)+1;
215            ErrCr(j,1+os)=ErrCr(j,1+os)+1;
               if j==1
                  fprintf(fid,'   0.00 | ');
               else
                  fprintf(fid,'   0.00\n');
220            end
            end
         end
         if y(j)*z(j)<0
            % Wrong sign
225         ErrUn(j,13)=ErrUn(j,13)+1;
            if (y(j)>SmallNmbr)|(z(j)>SmallNmbr)
               ErrCr(j,13)=ErrCr(j,13)+1;
            end
         end
230   end
   end
   % Display progress indication
   if etime(clock,t0)>ProgTimeSlice
      disp([spaces num2str(mPrt/(NoCells-1)*100,'%5.1f') '% is done.'])
235   t0=clock;
   end
end

disp([leader 'Summary of the results:']);
240 fprintf(fid,'\nSummary of the results:\n');

for fd=1:fid-1:fid
   fprintf(fd,'\n');
   fprintf(fd,'Excessive error: >%6.2f%%',PermRelErr);
245 fprintf(fd,'   ||          value 1        |          value 2        \n');
   fprintf(fd,'Small number   : %9.2e',SmallNmbr);
   fprintf(fd,'  || uncorrected | after cor | uncorrected | after cor \n');
   fprintf(fd,'-----------------------------+------------+-----------+-------------+---
--------\n');
250 fprintf(fd,'nodes                        |            |           |
|\n');
   fprintf(fd,'   mean of |rel.error|  (%%)  | ');
   fprintf(fd,'%11.2f | %9.2f | ',ErrUn(1,2)/ErrUn(1,1)*100,ErrCr(1,2)/ErrCr(1,1)*100);
   fprintf(fd,'%11.2f | %9.2f\n',ErrUn(2,2)/ErrUn(2,1)*100,ErrCr(2,2)/ErrCr(2,1)*100);
255 fprintf(fd,'   variance             (%%)  | ');
   fprintf(fd,'%11.2f | %9.2f | ',(ErrUn(1,3)-
ErrUn(1,2)*ErrUn(1,2)/ErrUn(1,1))/(ErrUn(1,1)-1)*10,...
      (ErrCr(1,3)-ErrCr(1,2)*ErrCr(1,2)/ErrCr(1,1))/(ErrCr(1,1)-1)*10);
   fprintf(fd,'%11.2f | %9.2f\n',(ErrUn(2,3)-
260 ErrUn(2,2)*ErrUn(2,2)/ErrUn(2,1))/(ErrUn(2,1)-1)*10,...
      (ErrCr(2,3)-ErrCr(2,2)*ErrCr(2,2)/ErrCr(2,1))/(ErrCr(2,1)-1)*10);
   fprintf(fd,'   maximum              (%%)  | ');
   fprintf(fd,'%11.2f | %9.2f | ',ErrUn(1,4)*100,ErrCr(1,4)*100);
   fprintf(fd,'%11.2f | %9.2f\n',ErrUn(2,4)*100,ErrCr(2,4)*100);
265 fprintf(fd,'   excessive            (#)  | ');
   fprintf(fd,'%11d | %9d | ',ErrUn(1,5),ErrCr(1,5));
   fprintf(fd,'%11d | %9d\n',ErrUn(2,5),ErrCr(2,5));
   fprintf(fd,'   rel. error undefined (#)  | ');
   fprintf(fd,'%11d | %9d | ',ErrUn(1,6),ErrCr(1,6));
270 fprintf(fd,'%11d | %9d\n',ErrUn(2,6),ErrCr(2,6));
   if NoRndPoints>0
      fprintf(fd,'non-nodes   %15d',NoRndPoints);
      fprintf(fd,'  |              |           |             |\n');
      fprintf(fd,'   mean of |rel.error|  (%%)  | ');
275   fprintf(fd,'%11.2f | %9.2f |
',ErrUn(1,8)/ErrUn(1,7)*100,ErrCr(1,8)/ErrCr(1,7)*100);
      fprintf(fd,'%11.2f |
%9.2f\n',ErrUn(2,8)/ErrUn(2,7)*100,ErrCr(2,8)/ErrCr(2,7)*100);
      fprintf(fd,'   variance             (%%)  | ');
```

```
280         fprintf(fd,'%11.2f | %9.2f | ',(ErrUn(1,9)-
ErrUn(1,8)*ErrUn(1,8)/ErrUn(1,7))/(ErrUn(1,7)-1)*10,...
               (ErrCr(1,9)-ErrCr(1,8)*ErrCr(1,8)/ErrCr(1,7))/(ErrCr(1,7)-1)*10);
            fprintf(fd,'%11.2f | %9.2f\n',(ErrUn(2,9)-
ErrUn(2,8)*ErrUn(2,8)/ErrUn(2,7))/(ErrUn(2,7)-1)*10,...
285            (ErrCr(2,9)-ErrCr(2,8)*ErrCr(2,8)/ErrCr(2,7))/(ErrCr(2,7)-1)*10);
            fprintf(fd,'   maximum                (%%)   | ');
            fprintf(fd,'%11.2f | %9.2f | ',ErrUn(1,10)*100,ErrCr(1,10)*100);
            fprintf(fd,'%11.2f | %9.2f\n',ErrUn(2,10)*100,ErrCr(2,10)*100);
            fprintf(fd,'   excessive              (#)   | ');
290         fprintf(fd,'%11d | %9d | ',ErrUn(1,11),ErrCr(1,11));
            fprintf(fd,'%11d | %9d\n',ErrUn(2,11),ErrCr(2,11));
            fprintf(fd,'   rel. error undefined (#)   | ');
            fprintf(fd,'%11d | %9d | ',ErrUn(1,12),ErrCr(1,12));
            fprintf(fd,'%11d | %9d\n',ErrUn(2,12),ErrCr(2,12));
295      end
         fprintf(fd,'wrong signs              (#)   | ');
         fprintf(fd,'%11d | %9d | ',ErrUn(1,13),ErrCr(1,13));
         fprintf(fd,'%11d | %9d\n',ErrUn(2,13),ErrCr(2,13));
         fprintf(fd,'\n');
300   end

      fclose(fid);

      disp([leader 'Results written to file ' ENVIROMENT_NAME ext])
305   disp([leader 'Done.   (' num2str(etime(clock,tt),'%3.1f') ' seconds elapsed)']);
      fprintf(1,'\n');
```

Function `writecfg`

Description:

The function `writecfg` creates or edits the configuration file that contains all global variables. If this function is invoked when no configuration file exists yet, the function will present the user with its internal defaults for all values. If the configuration file does exist, the user is asked to either leave existing values unchanged or supply new values. Optionally the function can be invoked with a configuration name as argument. The defaults will then be taken from that configuration and the user can change the values that do not suit him/ her.

Inputs:

dftfile	A string with the name of the configuration that should be used for the default values. Optional

Outputs:

A configuration file if it is not already present.

Depends on: —

Code:

```
function writecfg(dftfile)
%WRITECFG  Writes parameters to configuration file
%
%   WRITECFG(defaultfile)
5   %      Writes parameters to the configuration file defined by the file name in
%      variable 'configfile'. Optionally the file 'defaultfile' can be specified.
%      If this file is specified, not the internal defaults will be suggested, but
%      the parameters from this configuration file will be used as defaults.

10  global ENVIROMENT_NAME
    DefaultPath='d:\swingup\triang\';  % Set default path

    if nargin<1
       if exist([ENVIROMENT_NAME '.mat'],'file')
15        % CURRENT CONFIGURATION
          disp('Reading defaults from current external configuration file');
          load(ENVIROMENT_NAME);
       else
          % INTERNAL DEFAULTS
20        disp('Reading internal defaults for configuration');
```

```
        % Physical parameters
        g=9.81;       % gravitational acceleration
        l=1;          % pendulum length
        % Triangulation parameters
25      Dv=0.025;     % vertical spacing of triangulation
        Nh=252;       % total number of nodes in horizontal direction from 0 to 2pi
        Nv=3;         % number of nodes in vertical direction
        Nhp=1;        % horizontal bound of partial triangulation
        % Controller
30      controller='uncontrolled';
        % Controller parameters
        a=[15 3];     % parameters of linear controller
        umin=-0.25;   % lower bound of non-linear controller output
        umax=0.25;    % upper bound of non-linear controller output
35    end
    else
      PathDefined=0;
      %Check if filename contains path description
      filenamelength=length(dftfile);
40    if filenamelength>=2
        % File name starts with drive specification
        PathDefined=PathDefined|(dftfile(2)==':');
      end
      i=1;
45    while (i<=filenamelength)&(~PathDefined)
        PathDefined=PathDefined|(dftfile(i)=='/')|(dftfile(i)=='\');
        i=i+1;
      end
      %Strip extension if necessary
50    Stripped=0;
      i=max(1,filenamelength-3);
      while (i<=filenamelength)&~Stripped
        if dftfile(i)=='.'
          if i==filenamelength
55          dftfile=dftfile(1:filenamelength-1);
            stripped=1;
          else
            if
    (dftfile(min(i+1,filenamelength))=='/')|(dftfile(min(i+1,filenamelength))=='\') ...
60
    |(dftfile(min(i+2,filenamelength))=='/')|(dftfile(min(i+2,filenamelength))=='\')
              dftfile=dftfile(1:max(1,i-1));
              Stripped=1;
            end
65        end
        end
        i=i+1;
      end
      if ~PathDefined
70      % Set path to default path if it is not defined
        dftfile=fullfile(DefaultPath,dftfile);
      end
      disp('Reading defaults from external configuration file');
      % Load defaults from external configuration file
75    load(dftfile);
    end

    % Ask to adjust physical parameters
    fprintf(1,'\n');
80  disp('Physical parameters');
    disp('-------------------');
    disp(['   gravitational acceleration, g [m s-2]: ' num2str(g,'%5.2f')]);
    disp(['   pendulum length,            l [m]    : ' num2str(l,'%5.2f')]);
    feedback=input('\nAccept these physical parameters (Y/n)? ','s');
85
    if length(feedback)>0
      if (feedback(1)=='n')|(feedback(1)=='N')
        fprintf(1,'\n');
        disp('Enter new values for physical parameters.');
90      disp('Note: Minus sign will be ignored.');
        disp('Just hitting <ENTER> accepts the original value.');
        feedback=input('   gravitational acceleration,   g [m s-2]: ');
        if abs(feedback)>0
          g=abs(feedback);
95      else
          disp('   Default value retained');
        end
        feedback=input('   pendulum length,              l [m]    : ');
        if abs(feedback)>0
100       l=abs(feedback);
        else
```

```
         disp('   Default value retained');
      end
   end
end

% Ask to adjust triangulation parameters
fprintf(1,'\n');
disp('Triangulation parameters');
disp('------------------------');
disp(['   horizontal spacing,                      Dh  [rad]    : '
num2str(2*pi/Nh,'%6.4f')]);
disp(['   number of nodes in horizontal direction  Nh           : '
num2str(2*Nhp+1,'%6d')]);
disp(['   vertical spacing,                        Dv  [rad s-1]: '
num2str(Dv,'%6.4f')]);
disp(['   number of nodes in vertical direction    Nv           : ' num2str(Nv,'%6d')]);
feedback=input('\nAccept these triangulation parameters (Y/n)? ','s');

if length(feedback)>0
   if (feedback(1)=='n')|(feedback(1)=='N')
      fprintf(1,'\n');
      disp('Enter new values for triangulation parameters.');
      disp('Some parameters can only take certain values.');
      disp('Improper entries will be adjusted to the nearest proper value.');
      disp('Just hitting <ENTER> accepts the original value.');
      feedback=input('\n   horizontal spacing,              Dh [rad] : ');
      newvalue=abs(feedback);
      if newvalue>0
        if newvalue>pi
          Nh=2;
        else
          Nh=fix(pi/newvalue)*2;
        end
      else
         disp('   Default value retained');
      end
      fprintf(1,'\n');
      disp('   number of nodes in horizontal direction (odd)');
      feedback=input('   ''all'' will include all nodes from 0 to 2pi : ');
      if isstr(feedback)
         if cmpstr(feedback,'all')
            Nhp=0;
         else
                  disp('   Default value retained');
             end
      else
         if abs(feedback)>2
            Nhp=abs(feedback);
            if even(Nhp)
               Nhp=Nhp-1;
            end
            Nhp=fix(Nhp/2);
         else
            disp('   Default value retained');
         end
      end
      feedback=input('\n   vertical spacing,              Dv [rad s-1]: ');
      newvalue=abs(feedback);
      if newvalue>0
         Dv=newvalue;
      else
         disp('   Default value retained');
      end
      feedback=input('\n   number of nodes in vertical direction (odd): ');
      if abs(feedback)>2
         Nv=abs(feedback);
         if even(Nv)
            Nv=Nv+1;
         end
      else
         disp('   Default value retained');
      end
   end
end

% Ask to chance controller
fprintf(1,'\n');
disp('Controller');
disp('----------');
disp(['   controller                          : ' controller]);
feedback=input('\nAccept this controller (Y/n)? ','s');
```

```
if length(feedback)>0
   if (feedback(1)=='n')|(feedback(1)=='N')
      fprintf(1,'\n');
      disp('Choose new controller, e.g.');
      disp('uncontrolled, linear, non-linear');
      disp('Just hitting <ENTER> accepts the original value.');
      feedback=input('   controller                              : ','s');
      if length(feedback)>0
         controller=feedback;
      else
         disp('   Default value retained');
      end
   end
end

if strcmp(controller,'linear')
   % Ask to adjust controller parameters
   fprintf(1,'\n');
   disp('Controller parameters');
   disp('--------------------');
   disp(['   gain of first state                 : ' num2str(a(1),'%5.2f')]);
   disp(['   gain of second state                : ' num2str(a(2),'%5.2f')]);
   feedback=input('\nAccept these controller parameters (Y/n)? ','s');

   if length(feedback)>0
      if (feedback(1)=='n')|(feedback(1)=='N')
         fprintf(1,'\n');
         disp('Enter new values for parameters of the linear controller.');
         disp('Just hitting <ENTER> accepts the original value.');
         feedback=input('   gain of first state                    : ');
         if abs(feedback)>0
            a(1)=feedback;
         else
            disp('   Default value retained');
         end
         feedback=input('   gain of second state                   : ');
         if abs(feedback)>0
            a(2)=feedback;
         else
            disp('   Default value retained');
         end
      end
   end
end

if strcmp(controller,'non-linear')
   % Ask to adjust controller parameters
   fprintf(1,'\n');
   disp('Controller parameters');
   disp('--------------------');
   disp(['   lower bound controller,     umin     : ' num2str(umin,'%5.2f')]);
   disp(['   upper bound controller,     umax     : ' num2str(umax,'%5.2f')]);
   feedback=input('\nAccept these controller parameters (Y/n)? ','s');

   if length(feedback)>0
      if (feedback(1)=='n')|(feedback(1)=='N')
         fprintf(1,'\n');
         disp('Enter new values for parameters of the non-linear controller.');
         disp('Just hitting <ENTER> accepts the original value.');
         feedback=input('   lower bound controller        umin     : ');
         if abs(feedback)>0
            umin=-abs(feedback);
         else
            disp('   Default value retained');
         end
         feedback=input('   upper bound controller        umax     : ');
         if abs(feedback)>0
            umax=abs(feedback);
         else
            disp('   Default value retained');
         end
      end
   end
end

% Calculate extra parameters
K=Nv*Nh;                % Total number of nodes
M=2*(Nv-1)*Nh;          % Total number of cells

% Parameters for partial triangulation
```

```
    if Nhp==0
265     NoNodes=K;
        NoCells=M;
    else
        NoNodes=(Nhp*2+1)*Nv;
        NoCells=4*(Nv-1)*Nhp;
270 end

    save(ENVIROMENT_NAME,'g','l','Dv','Nh','Nv','Nhp','M','K','NoNodes',...
                         'NoCells','controller','a','umin','umax');
    fprintf(1,'\n');
```