Strategies for stabilizing a 3D dynamically walking robot

Gijs van Oort

M.Sc. Thesis

Supervisors

dr.ir. S. Stramigioli
ir. V. Duindam
prof.dr.ir. J. van Amerongen

June 2005

Report nr. 023CE2005
Control Engineering
EE-Math-CS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands
Abstract

A dynamic walker is a system that makes use of its natural dynamics in order to walk in an energy-efficient way. In this report two models of dynamic bipedal walkers are described, and strategies are discussed that stabilize the models. Main focus is on sideways stabilization by means of lateral foot placement.

The first model discussed is an 8-degrees-of-freedom bipedal walker, generated with 20-sim’s 3D Mechanics Editor. Each leg has a hip joint (forward/backward rotation), a splay joint (sideways rotation), a knee joint and an ankle joint (that lifts/lowers the foot). The feet are ellipsoids. The dimensions of the walker are inspired by humans, however, the walker has no upper body.

With the help of an ‘aligner’ block that prevents the walker from falling sideways, a simple controller was developed that stabilizes the walker in forward/backward direction. The hip joints are actuated, the splay and ankle joints are fixed and the knee joints are passive (they do have end stops and a locking mechanism). Also a controller was developed that controls the forward velocity of the walker by means of changing the ankle joint angle.

Then the influence of the aligner block was gradually reduced and a controller (based on trajectory prediction) was added that should stabilize the walker in sideways direction. This controller did improve the behaviour with respect to sideways falling, but by manual tuning no set of parameters could be found that stabilizes the walker completely.

The second model discussed is a ‘very simple 3D walker’, consisting of a point mass as the hip, and two massless, stiff legs. Energy injection is done by a ‘toe-off’ mechanism.

Two different controllers were developed to stabilize the walker in three dimensions. One uses a normal discrete state feedback controller based on a linearized approximation of the model around a certain stable cycle (a limit cycle). The other uses a property common to all limit cycles, that results from symmetry: the fact that the sideways velocity exactly halfway the step should be zero. As this controller is not optimized for just one limit cycle, it can be used over a whole range of different limit cycles.

Both controllers stabilize the walker well and have a reasonably large (but different) region of stability. The differences in performance are discussed and explained. It is expected that (a combination of) the controllers (with some adaptations) can also be used for more complex walker models.
Preface

This report describes the work I have done for the last step of my study Electrical Engineering: the Master’s project.

Mr. Stramigioli who was also my supervisor during the internship at DLR, Germany, offered me a great project: research on energy-efficient walking robots. My friend Edwin Dertien, who wanted to start his Master’s project around the same time, liked this subject too, so around September, 2004 we both started our (individual) projects on walking robots. It was great working on the same topic; whenever one of us had a problem, we could discuss it together.

I want to thank my supervisors, Mr. Stefano Stramigioli and Mr. Vincent Duindam, for their help, their interest and enthusiasm. I also want to thank the people from the Control Engineering department, under supervision of Mr. J. van Amerongen and Controllab Products B.V. for their help, Bart, Fayke, Rianne, Dennis, Agnes and Flip for doing the keyboard work I could not do because of RSI and Daphne for supporting me in general. And I especially want to thank Edwin for (again) a wonderful time of working together.

Gijs van Oort, June 2005
## CONTENTS

3.5.1 Analysis ................................................................. 37
3.5.2 Control ................................................................. 39
3.6 Comparison between the controllers ................................. 42

4 Conclusions and recommendations ................................. 43
4.1 Conclusions ............................................................... 43
4.2 Recommendations ....................................................... 44
4.2.1 Add finite mass feet ................................................. 45
4.2.2 Add larger feet ..................................................... 45
4.2.3 More realistic energy addition .................................... 45
4.2.4 More degrees of freedom ......................................... 45
4.2.5 More research on the upper body ............................... 45
4.2.6 Standing still, walking slowly and turning ................... 45
4.2.7 Many more things... .............................................. 45

A 20-sim tips and tricks ..................................................... 47
A.1 Execution order ......................................................... 48
A.2 Conditional assignments ............................................. 48
A.3 “Unable to break algebraic loop for...” .......................... 49
A.4 Instantaneous state changes: resint and event .................... 49
A.5 Double integration and resint ....................................... 51
A.6 Variables used in models of the 3D Mechanics Editor ........... 52

B Floor code ................................................................. 53
B.1 The CalculateContactPoint block .................................. 53
B.2 The CalculateForce block ........................................... 54

C Derivation of the equations for the very simple 3D walker .... 57

D 20-sim code for the very simple walker model ................... 67

E Matlab code for the very simple walker model .................... 71
E.1 stos.m .................................................................. 71
E.2 findlimitcycle.m ..................................................... 75
E.3 findstabilizationrange.m ............................................ 76
E.4 Script: pole placement .............................................. 77
Chapter 1

Introduction

Our world has become such that human beings can easily move around in it. For small distances, such as inside buildings, walking is the easiest and most flexible way of moving, and the environment has been totally adapted to this. In a typical building we find floors at different levels interconnected with stairs. For walking people this is perfect, but for wheeled systems this is a big obstacle. Therefore it is a not-so-strange thought to provide future robots that will work in such an environment with legs instead of wheels.

Unfortunately, the walking robots of today are not yet energy-efficient, stable and robust enough to function properly in a human-oriented environment. At the moment, research on this topic is done at many different places in the world, amongst others at the University of Michigan [7], Cornell [3] and Delft University [13]. Recently the University of Twente also started research on this topic.

Generally the field of walking robots can be divided into two categories: static walkers\(^1\) and dynamic walkers.

Static walkers usually have many joints, all of which are actuated. The robots are usually programmed to make their limbs follow a certain, naturally looking path. Any disturbance from this path is immediately suppressed by active control. This type of control makes such walkers quite energy-inefficient. For example, the Honda P3 consumes about 2 kW, which is more than 20 times the energy a human needs for walking [3].

Dynamic walkers usually have a number of unactuated joints. Therefore, the movement of the limbs attached to those joints is totally determined by the passive dynamics of the limbs. Because of this passive behaviour, no energy is needed for controlling those limbs. This makes a dynamic walker very energy-efficient. There even exist a class of truly passive walkers, that have no actuators at all. These walkers walk down a shallow slope, using only gravity as their source of energy.

Research at the University of Twente is done on dynamic walkers; we try to find ways to make them more energy-efficient, stable and robust.

1.1 Dynamic walkers

Making a dynamic walker actually walk instead of fall is quite a complex task. This is because, as opposed to a static walker, the trajectory of (some of) the limbs can not be controlled directly. Instead, one has to use tricks that make sure a disturbance in the gait is suppressed. Ideally, these tricks only use a tiny amount of actuator energy.

As an example, consider a dynamic walker that falls to the left side. By placing its left foot only a few mm more to the left than it would normally do, it can convert the excess of leftward

\(^1\)Although the term static walker is still frequently used, it is a little outdated. The ‘static walkers’ built today do show some dynamic behaviour, but by far not to the extent of the category of dynamic walkers.
velocity to forward velocity, so the energy of the disturbance can even be used in a useful way.

As it is tricky to make a dynamic walker, researchers started investigating a relatively simple case: walkers that only have two dimensions. These walkers can not fall to the left or right, only forward and backward. In simulation this can be done by only including two dimensions in the model. In practice however, this is not possible. There are two solutions to this problem. Firstly, one can make some kind of guiding system that prevents a two-legged robot from falling sideways. Secondly, one can make a four-legged robot that has all its legs in one line, both inner legs move simultaneously, as do both outer legs, comparable to someone walking with crutches. Figure 1.1 shows the principle of the guiding system. In figure 1.3 a four-legged walker is shown. The four legged walker is the most popular for 2D walkers, probably because it is smaller, more portable and it can walk on any floor without having to set up a guiding system.

Much research is also done on three-dimensional walkers. These look much more human than the 2D walkers: they have two legs and show side-to-side rocking behaviour. 3D walkers give many more problems than 2D walkers. They can fall to more directions (not only forward and backward, but also to the left and right) and they can rotate around many more axes. They usually have very wide feet in order to gain enough friction to suppress yaw (rotation around their vertical axis) and to guide the side-to-side rocking behaviour. Contrary to 2D walkers that show a stable gait for quite a wide range of parameters (even without active control), 3D walkers are usually unstable by themselves and need some kind of feedback control in order to not fall over.

1.2 Previous work

Since 1990 numerous researchers have been busy in the field of dynamic walkers. Many prototypes have been built, 2D as well as 3D types. Most of them are able to walk on a level floor or down a shallow slope. They show a stable gait to a certain extent, but the region of stability is usually small (so a small disturbance, e.g. a few mm drop of the floor height can make them fall). The walkers usually have one (or at most a few) way(s) of active stabilizing, and are primary developed to investigate just that way of stabilizing. Even though research has been done for this topic for a reasonable number of years, the resulting robots still show very poor behaviour. They are by far not robust enough to operate in a normal human environment. So, still much work will have to be done.

A promising state-of-the-art 3D dynamic walker is the biped ‘Denise’ of the Technical University of Delft (figure 1.2). To stabilize it in sideways direction it uses a passive technique called lean-to-yaw-coupling; comparable to a bicycle that steers in the direction it is leaning to [13].
1.3 Previous and current work at the University of Twente

Under supervision of S. Stramigioli research on dynamic walkers has been done for a few years now at the University of Twente. V. Duindam started a PhD studies on the topic, financed by the European project Geoplex. His work mainly focuses on providing and investigating tools that can be used for walker simulation, such as port-hamiltonian techniques and contact models.

In 2004 N. Beekman did a Master’s project on stabilization and control of a 2D walker by means of foot placement. His work resulted in a conceptual design of a 2D walker with four legs [1].

Simultaneously with my Master’s project, E. Dertien finished the design of the robot and constructed it in his Master’s project. The robot walks quite stably. The robot is shown in figures 1.3 and 1.4, the design is described in detail in [4].

1.4 Goal of the project

The goal of this Master’s project is to find new strategies for stabilizing 3D walkers. It focuses on sideways stability. Much research has already been done on this topic; it has even resulted in some experimental dynamic walker prototypes that indeed can walk stably [3, 13]. However,
these robots usually have a very small region of stability; only a minor disturbance is needed to cause a fall. Moreover, they have very wide feet, much wider than humans. This is not wrong in itself, but the question arises how humans can stabilize, even with narrow feet. Apparently there are more strategies than the ones already used in the prototypes.

One of these strategies is called sideways footplacement (or lateral foot placement). This has already been investigated by some researchers (e.g. Kuo [7]), but new strategies for controlling the sideways foot placement might lead to more robustness against falling sideways. Creating and investigating such new strategies are the main topic of this research.

1.5 Report outline

In this report two different walker models are described that were used during the project.

Chapter 2 describes a 3D bipedal walker with 8 degrees of freedom. The first few sections describe the simulation environment (section 2.1), the walker model itself (section 2.2), some modeling considerations (section 2.3) and a few submodels needed (section 2.4). The walker is first restricted to move only in two dimensions, and a controller is developed that keeps it upright in the forward/backward direction (section 2.5). As a sidestep on the research, a controller is implemented that controls the velocity of the walker by means of changing the ankle joint angles (section 2.5.2). After that the 2D motion restriction is removed and a controller is developed and investigated that should stabilize the walker in sideways direction by using sideways foot placement (section 2.6).

Chapter 3 describes a much simpler 3D walker model, consisting of a point mass as the hip and two massless, stiff legs. The dynamic equations for the walker are derived in section 3.2. Two controllers are developed to stabilize the walker in three dimensions, both using sideways foot placement. The first, described in section 3.4, uses pole placement on the linearized equations. The second, section 3.5, uses a property common to all limit cycles for control. Both controllers are compared in section 3.6.

Chapter 4 lists conclusions and recommendations. Some recommendations are just thoughts of the author and do not follow directly from the results described in this report.

Acknowledgement

This project has been done in the context of the European sponsored project GeoPlex with reference code IST-2001-34166. More information is available at www.geoplex.cc.
Chapter 2

An 8-DOF 3D walker in 20-sim

As a part of this project, a model of a 3D walker with eight degrees of freedom was built in the simulation tool 20-sim. The model was produced with 20-sim’s new 3D Mechanics Editor.

2.1 20-sim and the 3D Mechanics Editor

The application 20-sim [2], developed by Controllab Products B.V. is a very powerful simulation tool. One of its strengths is the fact that different modeling techniques, such as block diagrams, mathematical equations and bond graphs can be used together seamlessly. The energy based bond graph representation makes it perfect for describing multiple-domain systems. Many different system analysis tools are provided, as are real-time 3D animations and interaction with Matlab.

One of the new features of 20-sim is the 3D Mechanics Editor. In this editor one can construct multibody systems in a very easy way. The rigid bodies can be interconnected by different types of translational and rotational joints. For each body the mass, center of mass and moment of inertia can be set. Also, for animation purposes, the shape, size and color of each body can be chosen.

The 3D Mechanics Editor produces an equation submodel that can be imported in 20-sim. Interaction of the different bodies with each other and with the environment is internally calculated by using screw theory [12]. Here, forces and torques are represented together in one 6-dimensional variable called a wrench. Similarly, linear and rotational velocities are combined in a 6-dimensional variable called a twist. Note that this matches 20-sim’s energy based approach, as the product of a twist and a wrench is indeed an energy. Position and orientation of each body are kept in a homogeneous matrix, which has the advantage of not having any singularities.

1-DOF Joints in the model are represented as 1-dimensional power ports. This makes it very easy to attach anything to such a joint; a constant force or torque for example can be applied on the joint by simply connecting a Se-element. The flow delivered by the element then represents the (linear or angular) joint velocity. The product of effort and flow is the injected power.

It is also possible to attach any number of hinge points to any body in the 3D Mechanics Editor. This adds an extra 6-dimensional power port to the submodel, that can be used to apply external forces to the body (as a wrench) or, the dual of that, measure the velocity of that body (as a twist). Gravity acting on the system is handled internally by the submodel, so no external forces are needed for that.

The ease with which a complex multibody structure can be made and the flexibility of adding any number of power interaction ports to the model make the 3D Mechanics Editor very well suited for implementing a multi-degree-of-freedom walker.

When working with a model from the 3D Mechanics Editor, it is often convenient or even necessary to know what the internal variables in the submodel represent. Unfortunately, no list existed yet for this, so a non-exhaustive but useful list was made and put in appendix A.
CHAPTER 2. AN 8-DOF 3D WALKER IN 20-SIM

A screenshot of the 20-sim-editor and the 3D Mechanics Editor are shown in figure 2.1.

2.2 Model description

Using the 3D Mechanics Editor, an 8-DOF two-legged walker was constructed. Figure 2.2 shows this construction, with all degrees of freedom indicated and named. Also the orientations of the axes of the reference frame $\Psi_0$ are defined here. These are used throughout the whole report. The dimensions of the walker are inspired by the 2D walker that is currently under construction at the University of Twente and are shown in table 2.1.

2.2.1 Joints

Each leg has four joints. The hip joint moves the leg forward and backward relatively to the hip. The splay joint moves the leg sideways. The knee joint needs no explanation and the ankle joint
2.2. MODEL DESCRIPTION

2.2.1 Size Mom. of inertia

<table>
<thead>
<tr>
<th>Part</th>
<th>Mass (kg)</th>
<th>Size (m)</th>
<th>Mom. of inertia (10^3 kg m^2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>l</td>
<td>w</td>
</tr>
<tr>
<td>Torso</td>
<td>3.0</td>
<td>0.3</td>
<td>0.1</td>
</tr>
<tr>
<td>Hip</td>
<td>0.3</td>
<td>0.1</td>
<td>0.06</td>
</tr>
<tr>
<td>Upper leg</td>
<td>0.7</td>
<td>0.5</td>
<td>0.05</td>
</tr>
<tr>
<td>Lower leg</td>
<td>0.7</td>
<td>0.5</td>
<td>0.05</td>
</tr>
<tr>
<td>Foot</td>
<td>0.2</td>
<td>0.3</td>
<td>0.15</td>
</tr>
</tbody>
</table>

Table 2.1: The dimensions of the 8-DOF walker model.

rotates the foot up and down. Positive directions for the joints are the directions of the arrows shown in figure 2.2.

2.2.2 Upper body

Although it would improve stability to have an extending upper body (it brings the center of mass up) [13], it was chosen not to have one in the model. Instead, the torso was kept small and the mass was varied in some simulations to get a higher overall center of mass of the mechanism. This construction has the advantage over a real upper body that no active control is needed to keep the upper body upright.

2.2.3 Feet

The feet are modeled as ellipsoids that have approximately the size of a human foot. A big difference between human feet and these feet is the fact that these feet have only one contact point with the ground. This makes standing still harder, but ground contact simulations during walking easier.

Most dynamic walkers (both in simulation and in practice) also use curved feet instead of flat feet. Studies have shown that the behaviour of curved feet and that of flat feet with a compliant ankle (which is more or less what humans have) is quite similar: for both the center of pressure travels forward as the center of mass of the walker travels forward [9].

However, unlike many 3D walkers (such as Denise [13] and the unnamed walker of Collins, Wisse and Ruina [3]), the walker described here has ellipsoid feet instead of (nearly) cylindrical ones. Choosing cylindrical feet (with the cylindrical axes parallel to the world’s y-axis) would be obvious: a contact line segment provides friction in the rotational z-axis, which makes it easier to bring the swing leg forward with only a minimal reactive rotation around the z-axis. Wide feet also give more sideways stability, as they can provide a reactive torque if the walker tends to fall [9]. However, humans can walk on stilts, that neither have much rotational friction, nor guide the movement. Depending on the ground contact model, it is also easily possible to provide a large virtual friction coefficient of the foot for rotations around the z-axis, so the friction of cylindrical feet can be mimicked if needed.

2.2.4 Ports

With the 3D Mechanics Editor, hinge points in the torso and in the feet were added to the model. With these it is possible to apply external forces to the bodies. The torso hinge point was used to attach different types of ‘helper blocks’ that helped keeping the walker upright during the first experiments. Such a helper block is described in section 2.4.3. The hinge points in the feet were used to apply reactive ground forces.
CHAPTER 2. AN 8-DOF 3D WALKER IN 20-SIM

2.3 Contact models

The model that comes from the 3D Mechanics Editor represents only a set of interconnected bodies; any interaction with the environment must be added later in 20-sim. This also holds for the interaction with the ground. If there were no interaction, the walker would just fall through it. Generally there are two ways to model contacts between two bodies (e.g. the foot of a walker and the ground): as a rigid contact and as a compliant contact.

2.3.1 The rigid contact model

The rigid contact model assumes the collision between the bodies to be instantaneous and (in this case) totally inelastic. The instantaneous nature is such that that the velocities of the bodies also change instantaneously. The mathematical equivalent is applying an impulse to the system; internally the new state after collision is calculated and the state variables are set to their new value directly.

After the collision has taken place, the two bodies make contact. During contact the relative velocities of the two bodies are restricted to less than 6 dimensions. For example, in the case of a point foot having contact with a frictionless ground, the foot can rotate freely and translate in the x and y-direction. But the movement in the (negative) z-direction is impossible. If the ground has an infinitely large friction, only rotations are allowed, no translations. Internally these restrictions are solved with lagrangian multipliers: a virtual force is constructed that, when applied, makes the accelerations in the ‘forbidden’ directions zero. The end result of this is that some states become totally dependent on the other states. Effectively they are then no real states anymore, because they cannot be chosen freely.

The mathematics involved in the rigid contact model (especially for a multibody system) are quite complex and extensive. However, the solution is known and described in detail in [6]. The authors also provided a tool that generates part of the 20-sim code needed to implement rigid contacts in a multibody model. Because the internal states of the system must be directly set, this method needs editing of the walker’s equation model. This is generally unwanted, because in doing so one destroys the modularity of the model.

Note that some advanced 20-sim programming techniques are required when implementing the rigid contact model. Therefore it is strongly advised to read appendix A, that discusses these techniques.

2.3.2 The compliant contact model

The compliant contact model is mathematically much simpler. At the moment the two bodies make contact, a (possibly non-linear) spring and damper are attached between the two contact points. The spring and damper decelerate the bodies in such a way that a collision is simulated. As the deceleration is not instantaneous, the bodies do penetrate each other a little bit. If the system is critically damped or overdamped, the collision is totally inelastic.

If a linear spring and damper were used, the compliant model would suffer from discontinuities in accelerations and the so-called sticky effect. This can be solved by using the nonlinear Hunt-Crossley model as described in [5]. This model calculates the normal force \( F_N \) between the two bodies as:

\[
F_N(t) = \begin{cases} 
  k x^n(t) + \lambda x^n(t) \dot{x}(t) & x \geq 0 \\
  0 & x < 0
\end{cases}
\]

where \( x \) is the penetration depth, \( k \) and \( \lambda \) the spring and damper parameters and \( n \) a real number, dependent on the shape and material of the object. \( n \) is usually close to unity.

In the case of a foot hitting the ground, this model can at least be used for the z-direction. If the floor is modeled as having an infinitely large friction coefficient, it can also be used for the x- and y-component. However, a better model of the floor would be one with a more realistic
2.3. CONTACT MODELS

friction behaviour, including static, coulomb and viscous friction. Different models can be found in the literature that approach this type of friction. A relatively simple one, also used in the 20-sim block SCVS-friction, is the following model:

\[ F_F = F_N \cdot \left( \mu_c + (\mu_{st} - \mu_c) e^{-\left(\frac{x}{v_{st}}\right)^2} \right) \text{sgn}(\dot{x}) + \mu_v \dot{x} \]

where \( F_F \) is the resulting friction force, \( F_N \) is the normal force as calculated above, and \( \mu_c, \mu_{st}, \mu_v \) and \( v_{st} \) are the model parameters. In order to avoid discontinuities around \( \dot{x} = 0 \), the \text{sgn} function can be replaced by a \text{tanh} function with a very steep slope:

\[ \text{sgn}(\dot{x}) \approx \text{tanh} \left( \frac{\dot{x}}{v_{st}} \right) \]

where \( v_{st} \) is small (e.g. 0.001). As 20-sim provides integration methods with adaptive step sizes, it can cope perfectly with such steep slopes.

2.3.3 Comparison

Both the rigid contact model and the compliant contact model have their advantages and disadvantages.

The rigid contact model does not use stiff springs, so the simulation step length can be larger without making a too large error. However, the mathematics are more complex, so a simulation step takes more time to be calculated. No actual experiments were done, but it is estimated that the rigid contact model outperforms the compliant model with respect to simulation speed.

Another advantage of the rigid contact model is that the two contacting bodies are exactly touching each other instead of penetrating each other (a few mm if the springs are chosen not too stiff). This could make a huge difference in walking, because the clearance between the ground and the swing foot is also only a few mm. If the penetration depth of the stance foot happens to be just larger than the ground clearance of the swing foot, the swing foot will hit the floor, which would not have happened if a different floor model was chosen.

On the contrary, the compliant contact model wins on ease of implementation and flexibility. The latter is very important. In the rigid contact model the contact point is internally seen as a non-movable joint that imposes restrictions to the state. These restrictions prevent sliding of the contact point, so slipping contacts cannot be modeled. But some movies of already produced experimental 3D walkers (e.g. [13]) show that the feet of the walker occasionally do slip. The rigid contact model can not handle this, the compliant contact model can.

In principle a combination of the two can be made. The normal force \( F_N \) could be calculated by the rigid contact model, thereby solving the penetration depth problem. The compliant friction model could then be used to calculate the friction forces, allowing slipping. However, implementing this would again destroy modularity (as part of the contact forces are computed as being internal forces although they actually result from external interaction).

Moreover, the compliant forces only work during contact, not at the instant of collision. The rigid contact model however, does use that instant to apply an impulse that approximates the real world’s very short (but not infinitely short) moment of deceleration. During that short moment friction works too, so that should be implemented in a similar way: a frictional impulse. Unfortunately it is not possible to simply use the (linear) rigid contact equations for solving the (non-linear) friction problem. So, a new set of equations should be found to make the combined model work.

All in all the compliant contact model seems to be the best choice for modeling contacts for a dynamic walker. It is the most flexible, modular and comprehensible. The disadvantages (large simulation time and the penetration depth problem) can be solved by having patience and using a stiffer spring or modifying the contact detection algorithm for the swing leg.
2.4 Submodels

Apart from the model of the walker itself, other submodels are needed as well, such as controller blocks, joint actuators and contact modeling blocks. A number of these blocks were implemented as separate 20-sim submodels so that they could be used in a flexible way. Some of these submodels are described in more detail in this section.

2.4.1 Kneelock, kneefix, knee-end

These blocks were all primarily designed to operate on the knee joints of the walker, but they can be used on any joint. They all limit the motion of the joint in certain circumstances.

The kneelock works much in the same way as a door: if the knee is unlocked (door lever pulled down), the lower leg (door) can swing freely. If it is locked (door lever released), the leg (door) can still swing freely, except for one point (leg stretched or door closed) where it is held in that position until the next unlock. It can be used to keep the leg straight (knee locked) if the leg is stance leg, and bendable (knee unlocked) if the leg is swing leg. The kneelock needs information about the absolute position of the knee. This is information that is not given through the power port. Therefore, it makes use of the global variable $\theta$, which holds all joint angles.

The knee-fix is much simpler: it just holds the lower leg in its starting position (relatively to the upper leg, not to the world). Two versions of this block were made; one with a stiff PD-controller that holds the leg in place, and one with the 20-sim-function constraint. The latter is better; it iteratively finds a force that makes the joint’s angular acceleration exactly zero. However, it can only be used with the MBDF integration method and it cannot be in a conditional statement. So, the PD-controller is much more flexible. The knee-fix blocks only need a power port to connect the joint, so they are very easily connectible.

The knee-end block delimits the movement of the (knee) joint to a certain range. At the boundaries of the range a PD-controller prevents further movement, much in the same way as the compliant contact model described in section 2.3.2.

2.4.2 Floor

As already stated in section 2.3.3, the compliant contact model was used for ground modeling. For handling contacts between the (ellipsoid) foot and the ground, a submodel was made that features:

- realistic friction in translation and in rotation around the z-axis, including stiction and viscous and coulomb friction, based on the equations of section 2.3.2,
- rolling (frictionless),
- Hunt-Crossley equations for the z-component,
- Intelligent on/off mechanism that can be used to solve foot scuffing by temporarily ignoring the ground.

The submodel needs two connections to the walker model: the absolute position and orientation of the foot as an homogeneous-matrix (denoted by $H_{0s}^f$) and the 6-dimensional power port associated with a hinge point in the center of the foot ($T_{s,0}^s/W_s$). Also the foot size is needed as a set of parameters.

The floor model consists of two submodels, as shown in figure 2.3. The complete source code of both subblocks can be found in appendix B.

The calculate-contactpoint block determines which point of the ellipsoid has the lowest z-coordinate; it is this point that will make or have contact with the floor (at $z = 0$). It puts a frame $\Psi_p$ at the contact point, having the same orientation as the world frame, and outputs the position
of this frame expressed in world coordinates \( (H^0_p) \) and the position of the foot \( \Psi_s \) expressed in contact point coordinates \( (H^0_p) \).

The calculate-force block determines the wrench resulting from the contact. For the block the position of the contact point is known; if its z-coordinate is smaller than or equal to zero, there is ground contact. If so, a wrench will be exerted on the foot. This wrench depends on the position of the foot (especially the penetration depth) and on the velocity of the foot (in particular the velocity of the contact point). Therefore, it is convenient to express the velocity of the foot, \( T^{\cdot,0}_s \), in the contact point frame \( \Psi_p \):

\[
T^{\cdot,0}_s = \begin{pmatrix} \omega \\ v \end{pmatrix} = \text{Ad}_{H^0_p} \cdot T^{\cdot,0}_s
\]

By doing this, we separate rotation (rolling) from translation (slipping). Now \( \omega \) indicates a pure rotation around \( p \), \( v \) indicates a pure translation (so a non-zero \( v \) means the object is slipping).

The normal force \( F_N \) can now be calculated with the Hunt-Crossley equation and the use of \( p_z \) and \( \dot{p}_z \). This \( F_N \) is in the direction of the z-axis of \( \Psi_p \).

Now let us call the euclidean projection of \( \omega \) on the xy-plane \( \omega_{xy} \) (so, this is simply \( v \) with its z-component ignored). As the translational friction is non-linearly dependent on \( |v_{xy}| \), we cannot simply treat the x and y direction separately and then add the results; this only works in linear cases. Instead, we have to calculate the friction based on the total velocity and only then decompose it in x and y direction:

\[
|F_{F,xy}| = F(|v_{xy}|) \\
F_{F,x} = \frac{v_x}{|v_{xy}|} \cdot |F_{F,xy}| \\
F_{F,y} = \frac{v_y}{|v_{xy}|} \cdot |F_{F,xy}|
\]

where \( F \) is the friction equation as described in 2.3.2, depending linearly on \( F_N \) and nonlinearly on \( |v| \). The friction parameters were estimated by doing simple imaginary tests such as: ‘Could an 80 kg person stand on one foot if a sideways force of 200 N is pulling the foot? — I guess so, so the friction coefficient (including stiction) must be larger than \( \frac{200}{80} = 0.25 \).’ (See figure 2.4).

As for rotations only rotation around the z-axis of the frame \( \Psi_p \) has friction. For this the friction function is used with \( \omega_z \) as parameter: \( F_{F,\omega_z} = F(\omega_z) \). We now have all forces to build up the resulting wrench (expressed in \( \Psi_p \)):

\[
W^p = \begin{bmatrix} 0 & 0 & F_{F,\omega_z} & F_{F,x} & F_{F,y} & F_N \end{bmatrix}
\]

The power port wants to have the twist and wrench expressed in local coordinates \( \Psi_s \), so finally we have the resulting friction wrench:
CHAPTER 2. AN 8-DOF 3D WALKER IN 20-SIM

Figure 2.4: Illustration of an imaginary test done in order to estimate friction parameters.

\[ W^s = \text{Ad}^T \left( H^p \right) W^p \]

The floor can be turned on and off for each foot separately. If the floor is turned off, the simulation acts as if there were no floor at all; the foot can penetrate it freely, without experiencing a force. This is a useful feature if one wants to do some experiments without too much concern about foot scuffing.

If the floor is turned back on, it is first checked if the foot is above the ground. If not, the floor is not activated until the foot is high enough. This way, the floor can already be turned on while the foot is still scuffing, which simplifies the timing.

Using the floor submodel in 20-sim is easy. Just make sure there is a hinge point in the ellipsoid object, connect the AbsH and power port, set up the size of the ellipsoid in the calculate-contactpoint block and simulate.

2.4.3 Aligner

It is quite hard to make a 3D walker walk when starting from scratch. Everything has to be right, if one thing is not, the walker will fall. Therefore it is convenient to split up the work into smaller steps so that not all problems have to be solved at once. The aligner block makes it possible to let a 3D walker walk as if it were a 2D walker, and then gradually add more of the third dimension to it.

The aligner submodel is a block that applies an external wrench to the body associated with the hinge point it is attached to. The wrench is such that it aligns a certain vector \( \vec{r} \), fixed in the body frame, to a certain vector \( \vec{s} \) fixed in the world frame. If the body is the walker’s torso, \( \vec{r} \) is chosen to be the torso’s local y-axis and \( \vec{s} \) is chosen to be the world’s y-axis, the walker is prevented to rotate around its x-axis (which would cause sideways falling) and its z-axis (which would cause a change of walking direction). Indeed, these restrictions make that only movements are possible that a 2D-walker can do: the 3D problem is reduced to a 2D problem (see figure 2.5).

The wrench is internally determined by a PD-controller. By adjusting the \( K_p \) and \( K_d \) of the controller, one can choose any degree of ‘helping’ by the aligner. Very high values restrict the walker’s movement strictly to 2D, lower values give a limited 3D behaviour. Zeros disable the aligner, leaving the full 3D system as it is. This is very useful for designing controllers; one can gradually increase the demands for them in this way.

Assume the body’s vector \( \vec{r} \) is not aligned with the world’s vector \( \vec{s} \). The PD-controller should then apply a non-zero wrench to the body in order to get the vectors aligned. For that the con-
2.4. SUBMODELS

\[ \vec{r}_y = \vec{s}_x \]

\[ \vec{r}_x = \vec{s}_y \]

\[ \vec{r}_z = \vec{s}_x \]

\[ \vec{r}_x = \vec{s}_y \]

\[ \vec{r}_z = \vec{s}_x \]

Figure 2.5: A perspective (left) and a side (right) view of the 3D walker, showing that if the torso’s \( \vec{r} \)-axis is aligned with the world’s \( \vec{s} \)-axis, the walker indeed looks like (and behaves like) a 2D walker. This figure also makes clear that it is allowed for the body (torso) to rotate around the \( \vec{r} \)-axis without destroying the 2D behaviour.

troller needs an orientation error \( \vec{e} \) and a rotational velocity error \( \dot{\vec{e}} \). The \( \vec{e} \) should represent how misaligned both vectors are. The vector product is a good measure for this:

\[ \vec{e}_b = \vec{r}_b \wedge \vec{s}_b = \vec{r}_b \wedge \left( H_0^b \right)^{-1} \vec{s}_0 \Rightarrow |\vec{e}_b| = |\vec{r}_b| \cdot |\vec{s}_0| \sin \alpha = \sin \alpha \approx \alpha \]

where \( \vec{\bullet}^0 \) represents the vector \( \vec{\bullet} \) expressed in world coordinates, \( \vec{\bullet}^b \) represents the vector \( \vec{\bullet} \) expressed in body coordinates, and it is assumed that \( \vec{r} \) and \( \vec{s} \) are unit vectors and that the angle \( \alpha \) between the vectors is small. The direction of \( \vec{e} \) is the axis along which the body should be rotated to realign \( \vec{r} \) with \( \vec{s} \).

As the orientation of the \( \vec{r} \)-axis does not change when the body rotates around that \( \vec{r} \)-axis, the body may have any rotational velocity around \( \vec{r} \). Rotating around the \( \vec{r} \)-axis is not an error, so the \( \vec{r} \)-component of the body’s rotational velocity \( \vec{\omega} \) should be excluded from the rotational velocity error \( \dot{\vec{e}} \). This can be done by taking the projection of \( \vec{\omega} \) (this \( \vec{\omega} \) is the first 3 elements of the twist \( T^b_{0, b} \)) on the plane perpendicular to \( \vec{r} \):

\[ \dot{\vec{e}}^b = \vec{r}_b \wedge \left( \vec{\omega}^b \wedge \vec{r}_b \right) \]

with the same assumptions as above. The PD-controller is now simply:

\[ W^b = \begin{bmatrix} (K_p \cdot \vec{e}^b - K_d \cdot \dot{\vec{e}}^b)^T & 0 & 0 & 0 \end{bmatrix} \]

where \( K_p \) and \( K_d \) are positive controller parameters. The wrench is already expressed in body coordinates, so it can directly be attached to a power port of the walker model.
2.5 Simulation in 2D

With the help of the aligner block the movement of the walker model was restricted to two dimensions. Also the splay joint angles were fixed by using knee-fix blocks. The work of Beekman [1] was used as a starting point for the simulations in 2D.

The graphical 20-sim model is plotted in figure 2.6. The hip joints and ankle joints are actuated, they all have a PD-controller attached. The knee joints are equipped with end stops and a knee-lock, apart from that the joint movement is unactuated.

The walkerstate block determines in which phase of the step the walker is. A full cycle consists of four states, being:

- State 1: right leg is stance leg, left foot is still behind right foot,
- State 2: right leg is stance leg, left foot has passed right foot,
- State 3: left leg is stance leg, right foot is still behind left foot,
- State 4: left leg is stance leg, right foot has passed left foot.

These four states are used in the controller block to set the hip setpoints, operate the knee locks and to determine the behaviour of the ankle joints. If needed, the controller can also turn off and on the floor for each foot to artificially avoid foot scuffing. A functional overview of the controller block is given in table 2.2.

2.5.1 Simulation results

Much parameter tweaking had to be done before the system could walk stably. Especially foot scuffing was a big problem. Three main causes were found for that:
2.5. SIMULATION IN 2D

Table 2.2: Functional overview of the controller of the 8-dof walker in the 2D simulation. The numbers are all setpoints for the different PD controllers. Note that ‘locked’ means that the lower leg can still rotate freely, until it reaches its extended position; only then the leg will be kept in position. The two setpoints marked with an \(^*\) indicate that these setpoints are controlled when using ankle actuation as described in section 2.5.2.

<table>
<thead>
<tr>
<th>State</th>
<th>Stance leg</th>
<th>Left leg</th>
<th>Right leg</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>right</td>
<td>-0.3</td>
<td>unlocked</td>
</tr>
<tr>
<td>2</td>
<td>right</td>
<td>-0.3</td>
<td>locked</td>
</tr>
<tr>
<td>3</td>
<td>left</td>
<td>+0.3</td>
<td>locked</td>
</tr>
<tr>
<td>4</td>
<td>left</td>
<td>+0.3</td>
<td>locked</td>
</tr>
</tbody>
</table>

- The aligner block is not infinitely stiff, therefore the hip of the swing leg drops a few mm relatively to the hip of the stance leg. Consequently the foot of the swing leg is also a few mm lower than it should be. A solution is to increase the aligner stiffness, which leads to larger simulation times. Also a different aligner block could be made that makes use of 20-sim’s constraint function (see section 2.4.1), but that has the disadvantage that it can only be turned on or off, nothing in between. A last option would be to decrease the torso width, but that involves changing the walker model, which is also unwanted. It was chosen to increase the stiffness of the aligner to \(K_p = 1500, K_d = 100\), which results in a hip drop of about 2 mm.

- The compliant floor model lets the stance foot penetrate the floor a little. The result is that the total walker, including the swing foot, is closer to the ground. With \(K_p = 10000, K_d = 30000\) the penetration depth, and thus the loss in ground clearance for the swing foot is about 8 mm.

- The feet are so big that the toes hit the ground easily. The bending of the swing leg makes the foot rotate so that the front of the foot goes through the ground. Humans solve this problem by pointing their toes up (figure 2.7). The simulated walker has no toe joints, so it has to lift its whole foot up to avoid hitting the ground.

Figure 2.7: In order to get a positive ground clearance, humans lift their toes. The arrow shows where the toes would end up if they were not pointed upwards.
### Table 2.3: Properties of the walker’s gait when walking in 2D

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step time</td>
<td>0.68 s</td>
<td></td>
</tr>
<tr>
<td>Step length</td>
<td>0.45 m</td>
<td></td>
</tr>
<tr>
<td>Forward velocity</td>
<td>0.66 m/s</td>
<td></td>
</tr>
<tr>
<td>Ground clearance</td>
<td>2 mm</td>
<td></td>
</tr>
<tr>
<td>Power consumption</td>
<td>2.3 W</td>
<td></td>
</tr>
<tr>
<td>Peak power consumption</td>
<td>140 W</td>
<td>Once each step, for about 0.02 s</td>
</tr>
<tr>
<td>Specific resistance</td>
<td>0.07 – $P/(m\ g\ \upsilon)$</td>
<td></td>
</tr>
</tbody>
</table>

After manual parameter ‘optimization’ the ground clearance of the foot during normal gait was 2 mm. If a stiffer floor model would be used, in which the stance foot does not lie 8 mm under the floor, the ground clearance would be almost 1 cm, which is good enough.

The PD controller in the hip has parameters $K_p = 300$, $K_d = 100$ that were chosen after some trial-and-error simulations. The actuator consumes 2.3 W, which results in a specific resistance (also called specific cost of transport; the amount of energy required to carry a unit weight a unit distance) of 0.07, which is quite good (humans do 0.38 [11]). The maximum power dissipated is 140 W (only for 0.02 s), which is really a lot. The problem is the choice of the PD controller with a fixed setpoint instead of a smooth path. A different controller could do this much better. The reason that a PD controller with fixed setpoint was chosen anyway is because it is simple and effective [13, 1].

In the resulting simulation the 2D walker walks stably, with a naturally looking gait. The ground clearance is quite large, which is good. In figure 2.8 a series of pictures is shown of the walking system. Some properties of the gait are listed in table 2.3.

#### 2.5.2 Velocity control by means of ankle actuation

A new control feature was implemented in the ankles, which makes it possible to control the forward velocity of the hip. The principle used here is that by rotating the stance foot, the contact point can be shifted forward or backward, which affects the forward/backward acceleration due to gravity (see figure 2.9). A very simple controller was implemented that tries to give the torso a constant forward velocity $v_{ref}$ by adjusting the ankle setpoint $\alpha$ according to the following equation:

$$\alpha = K \cdot (v_{torso} - v_{ref})$$

where $K$ is a controller parameter. The resulting setpoint was limited such that $-0.5 \leq \alpha \leq 0.2$ (in rad) and then low-pass filtered. Even this very simple controller does good work in stabilizing the walker.
2.6 Simulation in 3D

The three-dimensional walker now walks in 2D, with the help of the aligner block. The next step is gradually removing the influence of the aligner and develop algorithms that keep the walker upright in three dimensions. Because the walker with eight degrees of freedom is quite complex (too complex to describe its motion with simple mathematical equations), it was decided to make the walker behave a little simpler. This was done by pointing the feet straight up, so that it walks on its heels. Because the heels are small, the feet now behave almost like point feet instead of (rolling) normal feet.

Also, the default splay angel for both legs was set to $-0.2$ rad, which puts the feet almost under the center of mass (see figure 2.11) instead of precisely under the hips. This might look a bit strange, but humans do this too: the sideways distance between the feet of a normally walking person is only a few cm. As the feet of the walker are almost under its center of mass, the sideways excursions of the torso are not so large, so only a small force is needed to control it. The aligner stiffness was reduced to $K_p = 100, K_d = 13$.

As a first step, a simple controller was made that relates the splay angle for the next step directly to the absolute $y$-position of the torso at the moment of transition from phase 1 (or 3) to phase 2 (or 4): $\phi_{\text{play}} = -0.2 \pm K \cdot y_{\text{torso}}$ for both legs, where either the plus or the minus is used, depending on which leg is the swing leg. So, if the walker is too far to the left, it places the feet more leftward so that the walker falls back to the right.

A simulation run was done in which the $v_{\text{ref}}$ was slowly increased from 0.1 m/s to 0.65 m/s. Indeed, the walker increased its speed accordingly; the step time $t_{\text{step}}$ varied from 1.24 s during the first steps to 0.64 s during the last steps. See figure 2.10. It should be noted that the controller was only active during state 1 and 3. It is during state 2 and 4 that the walker gains kinetic energy from falling forward; as this part of the step is uncontrolled on the ankles, it is no wonder that the forward velocity $v_{\text{torso}}$ increases to much more than $v_{\text{ref}}$.

As the main focus of the project is on sideways stabilization, optimization of this controller was left behind. It is expected that good results can be obtained by having a time-dependent $v_{\text{ref}}(t)$ that matches the velocity path of the desired gait.

Figure 2.9: Velocity (or actually: acceleration) control by ankle rotation $\alpha$: the position of the contact point $p$ influences the acceleration $a$ due to gravity working on the center of mass (c.o.m.). Left: the contact point behind gives a forward acceleration. Right: the contact point in front gives a backward acceleration.
CHAPTER 2. AN 8-DOF 3D WALKER IN 20-SIM

The controller performed not well at all, so it was adapted by including the sideways velocity of the torso. Unfortunately, manual tuning has so far not resulted in a series of parameters that stabilize the gait.

After trying to use the rotational velocity of the torso around the stance foot’s contact point and different methods to keep the torso straight, it was decided that something better was needed.

With the aligner block still at $K_p = 100$, a simple trajectory predictor was built that, at any time within the current step, predicts the $y$-position of the torso at the expected end time of the step. It then calculates a $\phi_{splay}$ for the next step such that the trajectory of the next step ends up in a desired $y$-position for the hip (see figure 2.12). After careful manual tuning of the parameters of the two controllers, the walker could do 10 steps before stumbling and falling forward (when the splay controller was switched off, it did only 2 steps and fell sideways, even with the aligner on). However, the gait was not very smooth and the feet slipped quite often.

Many simulations were done in the search for good parameter values, but they did not result in a series of parameters that made the system walk stably forever. As each simulation took at least 30 seconds, this was not very fast way of finding stable walking behaviour. Moreover, the main interest is not in finding a set of tweaked parameters for some controller with which the system happens to walk, but in finding and understanding a control method that just works (with a whole range of possible parameters) because it is a good method. Unfortunately, the 8-dof walker is just too complex to build and understand such a controller from scratch.

So, it was decided to leave the complex 8-dof walker behind for the moment and first investigate a (much) simpler walker model: ‘the very simple 3D walker’ of chapter 3.

Figure 2.10: Simulation results of the 2D walker accelerating. The bottom graph shows the angle of the left ankle. It can clearly be seen that the behaviour changes when the desired velocity changes.
2.6. SIMULATION IN 3D

Figure 2.11: The 8-dof 3D walker as it is being used for 3D simulations. The legs are bent a little inwards, to get the feet more or less under the center of mass. The walker walks on its heels, which behave more or less like a point contact.

Figure 2.12: Schematic overview of the steps of trajectory prediction. First it is determined where the hip is (1), and where it will end during this step (2). Then the desired end position of the next step is determined (3) and the best position of the foot for the next step is calculated (4).
Chapter 3
A very simple 3D walker

In order to explore the 3D behaviour of a walker in a more mathematical way, a very simple system was designed and simulated in 20-sim and Matlab. The equations of the system were (partly) derived with the help of Maple.

3.1 The walker

The walker that is used in this chapter has the following properties:

- The hip is modeled as a point of mass \( m \),
- The legs are attached to the hip in this point,
- The legs are massless, have no knees and have length \( \ell \),
- The feet are modeled as point feet and have no mass. A consequence of the massless legs and feet is that one cannot move the legs and feet by applying a force, as the acceleration would be infinite. Therefore, instead of using a force as an input, the position of the legs is used as an input directly,
- Each leg has two degrees of freedom: forward/backward position (\( \varphi_{\text{hip}} \)) and sideways position (\( \varphi_{\text{play}} \)).
- Active control of the system is only done by choosing an appropriate position for the swing leg. This type of control is called foot placement.

Figure 3.1 (left) shows how the joints of the walker work. Note that this figure is only a schematic drawing of the kinematics, the joints are assumed infinitely small and all exactly at the same point in the hip. This is shown in figure 3.1 (right).

Although the 3D walker can walk in any direction (and it does that too, if a disturbance is applied), we assume that the walker’s heading is always in the direction of the positive x-axis. This is done for angular reference purposes. If the splay angle \( \varphi_{\text{play}} \) is constant, the foot is moving in the xz-plane only (fore-aft motion); the y-coordinate of the foot remains constant.

The walker has a stance leg and a swing leg, denoted by the subscripts: \( st \) and \( sw \). These subscripts are only used if there is a chance of confusion. During the simulation of the step, the stance leg angles (\( \varphi_{\text{hip,}\text{st}}, \varphi_{\text{play,}\text{st}} \)) are calculated by the integration method, by using the formulas given in section 3.2. The swing leg angles (\( \varphi_{\text{hip,}\text{sw}}, \varphi_{\text{play,}\text{sw}} \)) are direct inputs (and the only inputs) of the system (and therefore they are no states of the system). Because the legs are massless, they do not influence the dynamics of the system (apart from contact constraints of course).

At foot impact the impact energy loss is calculated and new energy is added in order to keep walking. This is described in detail in sections 3.2.2 and 3.2.3. After these calculations, the angles
of the old swing leg can be used as initial angles of the new stance leg. The new swing leg
instantaneously gets a position defined by the input values, so it is actually not swinging. As a
consequence no foot scuffing can occur.

For both the stance leg and the swing leg $\phi_{\text{hip}}$ is positive if the leg is stretched backwards (so
that the foot is behind the hip, or, $x_{\text{foot}} < x_{\text{hip}}$). If the leg is stretched forwards, the angle $\phi_{\text{hip}}$ is
negative. So, for a normal walking gait, where the stance leg travels from the front to the back
(relatively to the hip), $\phi_{\text{hip},\text{st}}$ starts negative and becomes positive as the hip passes its highest
point. This gives a $\dot{\phi}_{\text{hip}}$ that is positive. For a normal gait the $\phi_{\text{hip},\text{sw}}$ will be negative at foot
impact, whereas $\phi_{\text{hip},\text{st}}$ is positive.

For both legs, $\phi_{\text{play}}$ is positive if the leg is pointing ‘outwards’, as seen from the front. For
the left leg, it means that the foot is left of the hip ($y_{\text{foot}} > y_{\text{hip}}$), for the right leg it means that
the foot is right of the hip. Inside the model there is no real awareness of left and right (only
for viewing purposes), the only important thing for the model is that the stance leg’s positive
direction is opposite to the swing leg’s positive direction.

We denote the state of the system at any time by

$$
\mathbf{x} = \begin{pmatrix}
\phi_{\text{hip}} \\
\dot{\phi}_{\text{hip}} \\
\phi_{\text{play}} \\
\dot{\phi}_{\text{play}}
\end{pmatrix}
$$

where all these $\phi$’s refer to the stance leg. The swing leg variables are inputs rather than states,
they are denoted by

$$
\mathbf{u} = \begin{pmatrix}
\phi_{\text{hip}} \\
\phi_{\text{play}}
\end{pmatrix}
$$

where both $\phi$’s refer to the swing leg. The states at the beginning of step $k$ are denoted by $x_k$. The
inputs only influence the system at the very end of each step (they determine when exactly the
3.2 Mathematical derivation of the model

3.2.1 Equations of motion

The direction choice of $\varphi_{\text{play,sl}}$ and $\varphi_{\text{play,sw}}$ has been such that it has no relation at all to left and right. Consequently, the resulting dynamic equations will neither have any relation to left and right. We only bring a choice for left and right for viewing purposes, not for calculating purposes.

However, while deriving the dynamic equations, some use has been made of the cartesian frame. For the mapping from the angles $\varphi$ to the cartesian coordinates $(x, y, z)$ we do need to choose which leg is the left leg and which is the right. So for the derivation of the equations we assume that the left leg is the stance leg and the right leg is the swing leg. As expected, the final dynamic equations are independent of $y$ (the left-right axis) again, so indeed they have no relationship to left and right any more.

If the position of the stance foot is taken as the origin, the position of the hip ($\vec{p}_{\text{hip}}$) can be expressed as follows, at each time $t$:

$$\vec{p}_{\text{hip}} = \begin{pmatrix} x_{\text{hip}} \\ y_{\text{hip}} \\ z_{\text{hip}} \end{pmatrix} = \begin{pmatrix} \ell \cos \varphi_{\text{play}} \sin \varphi_{\text{hip}} \\ -\ell \sin \varphi_{\text{play}} \\ \ell \cos \varphi_{\text{play}} \cos \varphi_{\text{hip}} \end{pmatrix}$$

where $\vec{p}$, $x$, $y$, $z$ and all $\varphi$’s are dependent on $t$. It is obvious that here all $\varphi$’s are related to the stance leg ($\varphi_{\ast,\text{sl}}$), as the position is calculated relatively to the stance foot. The motion of the hip during one step can be calculated with the Euler-Lagrange equations. The lagrangian is the kinetic co-energy minus the potential energy:

$$\mathcal{L} = T - V = \left( \frac{1}{2} ml^2 \left( \dot{\varphi}_{\text{play}}^2 + \dot{\varphi}_{\text{hip}}^2 \cos^2 \varphi_{\text{play}} \right) \right) - \left( ml \ell \cos \varphi_{\text{hip}} \cos \varphi_{\text{play}} \right)$$

Applying the Euler-Lagrange equations and solving for $\ddot{\varphi}$ gives us the angular acceleration...
of the stance leg due to the moving hip mass:

\begin{align*}
\ddot{\varphi}_{\text{hip}} &= \frac{g \sin(\varphi_{\text{hip}}) + 2 \ell \dot{\varphi}_{\text{hip}} \sin(\varphi_{\text{play}}) \dot{\varphi}_{\text{play}}}{\ell \cos(\varphi_{\text{play}})} \\
\dot{\varphi}_{\text{play}} &= -\frac{\sin(\varphi_{\text{play}}) \left( \ell \dot{\varphi}_{\text{hip}}^2 \cos(\varphi_{\text{play}}) - g \cos(\varphi_{\text{hip}}) \right)}{\ell}
\end{align*}

where \( g \) is the gravitational acceleration.

These equations (being the motion equations of an inverted spherical pendulum, expressed in \( \varphi_{\text{hip}} \) and \( \varphi_{\text{play}} \)) can be used by an integration method to calculate the state \( x \) at each time \( t \). The path calculated is the path that the hip follows during one step. As already stated in section 3.1, the swing leg does not influence the system, as it does not make contact with the floor during this phase of the gait. Therefore it is no wonder that \( u \) does not appear in the equations.

At the end of the step foot impact occurs, for which some more equations are needed.

### 3.2.2 Impact equations

At the end of a step, when the swing foot hits the ground, we speak of foot impact. On impact the old swing leg becomes the new stance leg, and the new swing leg’s position is set directly according to the inputs. In this model the contact and support transfer is assumed to be totally inelastic and instantaneous, so some energy is lost. This can be modeled with an instantaneous change of the momentum of the system. The state of the system just after impact (as a function of the state just before impact) can be calculated by using the impact equations described in this subsection. The impact equations combine two purposes:

- Calculate the kinetic energy loss due to the impact of the foot,
- Express the angles and angular velocities of the hip in terms of the angles of the new stance leg.

The notational convention used here for describing the impact equations here is as follows:

- Pre-impact variables are superscripted \(^{-}\), e.g. \( \varphi^{-}_{\text{hip}} \), meaning: the hip-angle, an infinitesimal small time before impact. Similarly, post-impact variables are superscripted \(^{+}\). The values of the input \( u \) are only important just before impact (they act as an input for the impact equations), not after it. So, no confusion can be made, and the \(^{-}\) is omitted.
- Leg naming is simply according to the function of the leg before and after impact: \( \varphi^{+}_{\text{st}} \) refers to the pre-impact stance leg, \( \varphi^{-}_{\text{sw}} \) and \( \varphi^{+}_{\text{sw}} \) refer to the leg that is pre-impact swing leg and becomes stance leg after impact, and \( \varphi^{+}_{\text{sw}} \) refers to the new (post-impact) swing leg.
- Only the stance legs (the old stance leg just before impact and the new one just after impact) are important for the impact equations. The swing leg is not used in the equations, so all variables refer to the stance leg. Because there is no risk of confusion, the subscript \( \text{st} \) will be omitted.

At foot impact the hip instantaneously changes direction and speed. For clarification, this is made visible for the 2D case in figure 3.2. Here, the velocity \( \dot{\varphi}^{-} \) of the hip just before foot impact is decomposed into two components: one along the swing leg \((-\dot{\varphi}^{+}\ell)\), and one perpendicular to that \((\dot{\varphi}^{+})\). The inelastic, instantaneous collision is modeled as the ground applying an impulse to the walker, along the leg. The definition of this type of collision states that the impulse is exactly large enough to stop the movement of the hip in the direction of \(-\ell\) (proof: if the impulse were too small, the foot would penetrate the floor, making the collision non-instantaneous. If
the impulse were too large, the hip would get a positive velocity in the direction of $+\ell$ which would make the foot leave the ground again, making the collision elastic instead of inelastic).

So, directly after impact, the impulse will set the component $-\dot{\ell}$ to zero. The kinetic energy that is dissipated by this impulse is lost. After foot impact the velocity of the hip is $\dot{\varphi}^+$. Mathematically the decomposition can be seen as a simple mapping from one polar coordinate frame ($\Psi^- = (\varphi^-, \ell^-)$) to another ($\Psi^+ = (\varphi^+, \ell^+)$). Because the legs have a fixed length $\ell$, the $\dot{\ell}^-$ is always zero.

In three dimensions the only thing that changes is that we have two $\dot{\varphi}$ variables to cope with: $\dot{\varphi}_{\text{hip}}$ and $\dot{\varphi}_{\text{swing}}$. An easy way of doing the mapping is by using cartesian coordinates as an intermediate step. The two mapping functions $M$ are dependent on the actual spherical coordinates of the hip:

$$
\begin{pmatrix}
\varphi_{\text{hip}}^- \\
\varphi_{\text{swing}}^-
\end{pmatrix}
M_{\varphi_{\text{hip}}, \varphi_{\text{swing}}}
\begin{pmatrix}
x \\
y \\
z
\end{pmatrix}
M^{-1}_{\varphi_{\text{hip}}, \varphi_{\text{swing}}}
\begin{pmatrix}
\varphi_{\text{hip}}^+ \\
\varphi_{\text{swing}}^+
\end{pmatrix}
$$

Although $\ell$ is constant and therefore could be omitted as a variable in the formula above, it is preferable to leave it in, for a couple of reasons. Firstly, it makes the equations bijective and thus invertible. This way, we preserve time-symmetry, which is needed later on. Secondly, having $\ell^+$ makes it easy to calculate the impact energy loss: this is simply $E_{\text{loss}} = \frac{1}{2}m(\ell^+)^2$. A last advantage of having $\ell$ in the formula is that one can use it to inject energy into the system by using a nonzero value for $\dot{\ell}^-$ at the exact moment of impact. This is described in detail in section 3.2.3. Note that the nonzero values for $\dot{\ell}^-$ and $\dot{\ell}^+$ occur only for an infinitesimally small period of time, so the legs don’t become shorter or longer because of this.

By using Maple, formulas were derived to calculate $(\dot{\varphi}_{\text{hip}}^+, \dot{\varphi}_{\text{swing}}^+, \ell^+)$). The two other states, $\varphi_{\text{hip}}^+$ and $\varphi_{\text{swing}}^+$ are the angles of the new stance leg, which was the swing leg before impact. The position of the swing leg was simply equal to the input $u$. During impact the angles of the leg do not change, hence the angles $\varphi_{\text{hip}}^+$ and $\varphi_{\text{swing}}^+$ are simply equal to $u$.

All impact equations can be combined in the (non-linear) function $\text{IE}$:

$$
\begin{pmatrix}
\varphi_{\text{hip}}^+ \\
\dot{\varphi}_{\text{hip}}^+ \\
\varphi_{\text{swing}}^+ \\
\dot{\varphi}_{\text{swing}}^+
\end{pmatrix}
= \text{IE} (\begin{pmatrix}
\varphi^- \\
\dot{\varphi}^- \\
\varphi^- \\
\dot{\varphi}^-
\end{pmatrix}
, u, u)
$$

Figure 3.2: Sketch of a 2D walker at the moment of impact. The hip instantaneously changes direction and speed; from $\varphi^-$ to $\dot{\varphi}^+$. 

\[ \text{Figure 3.2: Sketch of a 2D walker at the moment of impact. The hip instantaneously changes direction and speed; from } \varphi^- \text{ to } \dot{\varphi}^+. \]
3.2.3 Energy injection

At each step the system loses a certain amount of energy $E_{\text{loss}}$ due to foot impact. In order to walk forever, we need to add energy each step to compensate for this. One way of adding this energy is by applying a force (or more correct: an impulse) along to the old stance leg, at the instant of foot impact. The impulse can be interpreted as resulting from the old stance leg being extended (simulating the push-off of the human ankle during normal gait). The kinetic energy added to the system will be denoted by $E_{\text{add}}$. In order to keep the energy level of the system constant, as much energy should be added as is lost during a step, thus: $E_{\text{add}} = E_{\text{loss}}$.

In the previous section the variable $\dot{\ell}^-$ was already introduced, and it was also indicated that this variable could be used for energy injection. The amount of energy added to the system is, analogous to the energy loss described earlier: $E_{\text{add}} = \frac{1}{2} m (\dot{\ell}^-)^2$.

The signs of $\dot{\ell}^-$ and $\dot{\ell}^+$ are easily determined by examining the configuration: the pre-impact stance leg should be extended in order to give the walker some forward energy, so $\dot{\ell}^-$ is positive. If the new stance leg hits the floor, it will experience a compressing force (instantaneously counterreacted by the leg itself), so $\dot{\ell}^+$ is negative. This gives the following equations:

$$E_{\text{add}} = E_{\text{loss}} \iff \frac{1}{2} m (\dot{\ell}^-)^2 = \frac{1}{2} m (\dot{\ell}^+)^2 \iff \dot{\ell}^- = -\dot{\ell}^+$$

The impact equations (described in the previous section and shown in appendix C) reveal that $\dot{\ell}^+$ depends on $\dot{\ell}^-$ (so also $E_{\text{loss}}$ depends on $E_{\text{add}}$), so the above equation gets a little bit more complex. Fortunately, the dependency happens to be not so difficult and can be written as:

$$\dot{\ell}^+ = a \dot{\ell}^- + b$$

where $a$ and $b$ are only dependent on the (fixed) leg length $\ell$, the state just before impact $x^-$ and the input $u$, and are thus known. Substitution of $\dot{\ell}^+$ and rearranging gives us the desired $\dot{\ell}^-$. As an indication for what the resulting equations look like, $a$ and $b$ have been substituted too, giving the totally expanded equation:

$$\dot{\ell}^- = \frac{-b}{1 + a}$$

$$= \left( \ell \cos u_2 \left( \cos u_1 \left( \sin \varphi_{\text{hip}}^- \cos \varphi_{\text{splay}}^- \varphi_{\text{hip}}^- + \cos \varphi_{\text{hip}}^- \sin \varphi_{\text{splay}}^- \varphi_{\text{hip}}^- \right) + \sin u_1 \left( \sin \varphi_{\text{hip}}^- \sin \varphi_{\text{splay}}^- \varphi_{\text{hip}}^- - \cos \varphi_{\text{hip}}^- \varphi_{\text{splay}}^- \varphi_{\text{hip}}^- \right) \right) \right. + \ell \cos \varphi_{\text{splay}}^- \varphi_{\text{splay}}^- \sin u_2)\left/ \left( 1 + \cos u_2 \cos \varphi_{\text{splay}}^- \left( \sin \varphi_{\text{hip}}^- \sin u_1 + \cos \varphi_{\text{hip}}^- \cos u_1 \right) - \sin \varphi_{\text{splay}}^- \sin u_2 \right) \right.$$

This type of energy addition was always used during this research. In fact, it was implemented as a part of the impact equations. The resulting impact equations (thus including energy addition) can be found in appendix C. It is these equations that were used for simulation and mathematical analysis (see also figure 3.3).

3.3 Implementation and simulation of the model

In this section the implementation of the model in 20-sim is explained. Also a set of typical parameter values is chosen that will serve as reference values for further simulations. Also results of a simulation are shown in which the walker was uncontrolled.
3.3. IMPLEMENTATION AND SIMULATION OF THE MODEL

Impact equations including Energy injection

Figure 3.3: A block diagram of the mathematical structure of the walker. The stride function \( S \), used throughout the rest of the chapter, includes the equations of motion and the impact equations, including the energy injection mechanism.

3.3.1 The complete model in 20-sim

Having the equations of motion and impact equations (including energy addition), all ingredients are present to build a simulation block of the walker. In order to be able to do different simulation tests, a block for 20-sim was built as well as a set of Matlab files. Exporting the long formulas was really simple with the use of Maple’s new Matlab command, that directly converts the used formulas to ready-to-run Matlab code. The code obtained is also fully 20-sim-compatible.

The 20-sim model is discussed in more detail here; the code itself can be found in appendix D. For (some of) the Matlab code see appendix E.

Figure 3.4 shows the appearance (left) and a schematic representation of the inner structure of the 20-sim model (right). Although it is perfectly possible in 20-sim to actually build the model in such a graphical representation, it was decided not to do this and keep everything together in one equation submodel. However, the blocks are still clearly recognizable in the code.

The blocks inside the dashed rectangle describe the walker itself. The three blocks outside it are auxiliary blocks for visualization and controller-connection purposes.

The dynamic equations block simply calculates the accelerations \( \ddot{\phi}_{\text{hip}} \) and \( \ddot{\phi}_{\text{play}} \) from the state \( x \), using the equations from section 3.2.1. These accelerations are directly fed into the resettable integrator block that updates the state \( x \).

The impact equations block continuously calculates a new state \( x^+ \) from the current state. The output of this block at each time can be seen as the new state \( x^+ \) if impact would have occurred at that time.

The foot impact detection block fires an ‘impact event’ when it detects the swing foot hitting the ground. It does this by calculating the cartesian coordinates of the hip and the swing foot relatively to the stance foot, and keeping an eye on the \( z \)-coordinate of the swing foot. If it has a negative derivative and it crosses zero, it is seen as a foot impact.

As soon as the resettable integrator receives the fired impact event, it sets its output state \( x \) equal to its ‘new-value’-input, so the state becomes \( x^+ \), just as intended.

An extra advantage of the way the foot impact detection block is split is that the hip and foot positions can also be used as an input for the simulator’s animation view. Some extra animation-related variables are set in the animation variables block.

In the plot-variables block some interesting values (e.g. the energy of the system) are calcu-
3.3.2 Typical parameter values and limit cycles

In order to produce comparable results during the research, a set of typical parameter values is needed that can be used throughout the whole research. It is also convenient to define one limit cycle for further usage. In this section these values are defined.

The walker model has only got three parameters: the hip mass $m$, the leg length $\ell$ and the gravitational acceleration $g$. The hip mass does not really influence the gait of the walker; it only scales the forces and energies. For convenience the mass is chosen to be 1 kg. The leg length is chosen to be 1 m. This is also a convenient value, plus it is close to the length of a human leg and to the leg length of the 2D-walker “Dribbel” [4] that is currently constructed at the University of Twente. So, results can be compared easily between these subjects. The gravitational acceleration is simply chosen to be that of the earth: 9.81 m/s$^2$.

The input $u$ determines the step length and width. For $u_1 = \varphi_{\text{hip,sw}}$ a value of $-0.3$ rad was chosen, which gives an inter-leg angle of 0.6 rad in the limit cycle. This is a value that is used more often. For example, walkers described in [13], [7], [1] and [8] use leg angles around this value. A smaller angle would lead to a more energy-efficient walker, but the step length and hence the walking speed would decrease. Fully passive walkers usually use a smaller inter-leg angle. Humans use a larger inter-leg angle (roughly 0.9—1 rad), as can be seen in figure 3.5. It would seem inefficient to use such a large inter-leg angle, but humans have various techniques to prevent this loss, such as foot rolling and a non-instantaneous double-support phase.

The splay angle $u_2 = \varphi_{\text{splay,sw}}$ was chosen to be 0.05 rad, resulting in a step width of about 10 cm. This value was chosen rather arbitrarily: not too large (as is the case in human gait the fore-aft motion should dominate over sideways motion) but neither too small (we want a real 3D behaviour in which sideways motion is certainly present).

A good limit cycle as starting point for the research was found by using Matlab. First a function was written that calculates the output of the stride function (also called step-to-step function):

\[
\text{function} \ [x_{\text{next}}, \ldots] = \text{stos}(x_{\text{cur}}, u_{\text{cur}})
\]
3.3. IMPLEMENTATION AND SIMULATION OF THE MODEL

Apart from the new state it also returns some extra information, such as the step time and if there exists any $x_{\text{next}}$ at all.

In order to find an $x^*$ that produces a limit cycle for a given input $u^*$, Matlab’s root finding function `fsolve` was used to find the value of $x^*$ that gives:

$$S(x^*, u^*) - x^* = 0$$

The stride function returns the new state at the beginning of the step, just after the impact equations have been applied. Therefore the states $\phi_{\text{hip}}^*$ and $\phi_{\text{spall}}^*$ are simply equal to $u^*$ (as shown in section 3.2.2). So we have only two equations and two unknowns ($\dot{\phi}_{\text{hip}}^*$ and $\phi_{\text{spall}}^*$) left. It appears that there are infinitely many solutions for this system of equations. The underlying reason for this is symmetry, and is described in section 3.5.1. In order to have only one limit cycle for research, one of the solutions was picked: the one that gives a $\dot{\phi}_{\text{hip}} = 0.7$ rad/s at the topmost point ($\phi_{\text{hip}} = 0$ rad). There was no special reason why exactly this one was chosen.

The full set of typical parameters and the chosen limit cycle state are shown in table 3.1. Table 3.2 shows some characteristics of the gait. The limit cycle is unstable, so a means of control is needed to stabilize it.

### 3.3.3 Simulation results

To give a little more insight in what is going on, the results of a typical simulation of the 20-sim model are included here. The simulation was performed using the typical parameters and input values listed in table 3.1. For demonstration purposes, instead of the limit cycle state, a different initial state was used that makes the system fall within a few steps. The initial state used is close to the limit cycle however: $x_{\text{init}} = (-0.3 \text{ rad}, 1.17 \text{ rad/s}, 0.05 \text{ rad}, -0.12 \text{ rad/s})^T$. In this simulation there was no feedback; the inputs $u$ were just kept at their fixed value.

The simulation results are presented in two figures. In figure 3.6 it can clearly be seen that the first two steps look quite similar, which is expected behaviour close to the limit cycle. However, the third step is already quite different: the system is falling. Figure 3.7 shows a top view and a side view of the walking motion. The figure was made by overlaying multiple frames from the 20-sim Animation View.
CHAPTER 3. A VERY SIMPLE 3D WALKER

<table>
<thead>
<tr>
<th>Parameter/ input/state</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m$</td>
<td>1 kg</td>
<td>Hip mass</td>
</tr>
<tr>
<td>$\ell$</td>
<td>1 m</td>
<td>Leg length</td>
</tr>
<tr>
<td>$g$</td>
<td>9.81 m/s$^2$</td>
<td>Gravitational acceleration</td>
</tr>
</tbody>
</table>

$$u = \begin{pmatrix} \phi_{\text{hip,sw}} \\ \phi_{\text{splay,sw}} \end{pmatrix} = \begin{pmatrix} -0.3 \text{ rad} \\ 0.05 \text{ rad} \end{pmatrix}$$

Default input

$$x^* = \begin{pmatrix} \phi_{\text{hip,sl}} \\ \dot{\phi}_{\text{hip,sl}} \\ \phi_{\text{splay,sl}} \\ \dot{\phi}_{\text{splay,sl}} \end{pmatrix} = \begin{pmatrix} -0.3 \text{ rad} \\ 1.17318 \text{ rad/s} \\ 0.05 \text{ rad/s} \\ -0.11559 \text{ rad/s} \end{pmatrix}$$

Chosen limit cycle state

Table 3.1: Typical parameters, input values and a limit cycle state that will be used as a starting point for the research.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_{\text{step}}$</td>
<td>0.702 s</td>
<td>Step time</td>
</tr>
<tr>
<td>$l_{\text{step}}$</td>
<td>0.59 m</td>
<td>Step length</td>
</tr>
<tr>
<td>$v_x$</td>
<td>0.84 m/s</td>
<td>Forward velocity</td>
</tr>
<tr>
<td>$\dot{\phi}<em>{\text{hip}} \mid \phi</em>{\text{hip}} = 0$</td>
<td>0.7 rad</td>
<td>Hip velocity at mid-stance</td>
</tr>
<tr>
<td></td>
<td>19 + 19 mm</td>
<td>side-to-side rocking</td>
</tr>
<tr>
<td>$E_{\text{add}} = E_{\text{loss}}$</td>
<td>0.068 J</td>
<td>Energy consumption per step</td>
</tr>
<tr>
<td>$P$</td>
<td>0.097 W</td>
<td>Power consumption</td>
</tr>
<tr>
<td></td>
<td>0.012 –</td>
<td>Specific cost of transport</td>
</tr>
<tr>
<td></td>
<td>unstable</td>
<td>Stability</td>
</tr>
</tbody>
</table>

Table 3.2: Characteristics of the limit cycle gait as described in section 3.3.2.
3.4 Stabilizing the gait by means of pole placement

The walker’s stride function $S$ can be seen as a discrete system: the states of the next step are calculated from the current step and the inputs. So, a discrete controller can be used to stabilize the system. In this section it is shown that the walker can be stabilized by means of pole placement, much in the same way as described in [7].

Pole placement can only be used on linear systems, so the stride function should be linearized first. Of course, no global stability can be guaranteed when using the controller on the non-linear model, but hopefully the local stability range is large enough to be of interest. In appendix E.4 the solution of the following equations is calculated numerically; here only the most important numbers are listed. Linearization of the uncontrolled system around the equilibrium $(x^*, u^*)$ gives:

$$x^* + \Delta x_{k+1} = S (x^* + \Delta x_k, u^* + \Delta u_k)$$
$$x^* + \Delta x_{k+1} \approx S (x^*, u^*) + J_x \Delta x_k + J_u \Delta u_k$$
$$\Delta x_{k+1} \approx J_x \Delta x_k + J_u \Delta u_k$$

where $J_x = \frac{\partial S}{\partial x}_{x^*,u^*}$ and $J_u = \frac{\partial S}{\partial u}_{x^*,u^*}$.

Figure 3.6: Graph of the states of the system during simulation. After step 2 the system starts falling (this is best visible in phi dot_splay).
Figure 3.7: Top view (top) and side view (bottom) of the steps done by the walker during simulation. See also figure 3.6.
3.4. STABILIZING THE GAIT BY MEANS OF POLE PLACEMENT

<table>
<thead>
<tr>
<th>eigenvalue</th>
<th>1.0000</th>
<th>0.3754</th>
<th>0.0580</th>
</tr>
</thead>
<tbody>
<tr>
<td>eigenvector</td>
<td>(\begin{bmatrix} 0 &amp; 0 &amp; 0.3754 &amp; 0 \ -0.9921 &amp; 0 &amp; -0.0983 &amp; 0.0580 \ 0 &amp; 0 &amp; 0.9952 &amp; 0.3285 \ -0.9952 &amp; -0.1252 &amp; 0 &amp; -0.9427 \end{bmatrix})</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.3: The eigenvalues and eigenvectors of the uncontrolled system.

### 3.4.1 Eigenvalues

Now the eigenvalues of \(Jx\) can be determined. The system is (locally) stable if all eigenvalues have a magnitude smaller than 1. The eigenvalues and eigenvectors are listed in table 3.3. Indeed, the system is unstable (as was already experienced in section 3.3.2), because of the eigenvalue of \(-4.46\). The other eigenvalues seem a bit strange: they are exactly 0 or 1. However, they can be explained quite easily.

As already stated in section 3.3.2, for a given \(u\) there exist infinitely many limit cycles \(\{x^*\}\). They can be described as:

\[
\{x^*\} = \begin{bmatrix} u_1 \\ \phi_{\text{hip}}(n) \\ u_2 \\ \phi_{\text{play}}(n) \end{bmatrix}
\]

where \(\phi_{\text{hip}}(n)\) and \(\phi_{\text{play}}(n)\) are nonlinear functions and \(n\) a real number (in section 3.5.1 it will be shown that all limit cycles indeed lie on a curve and thus can be spanned with only one parameter \(n\)). Locally around \(x^*\) the series of limit cycles can be linearized:

\[
\{x^*\} \approx \begin{bmatrix} x^*_1 \\ x^*_2 + V_2 \cdot m \\ x^*_3 \\ x^*_4 + V_4 \cdot m \end{bmatrix} = x^* + V \cdot m \quad (\text{where } m \text{ is small})
\]

\(V = (0, V_2, 0, V_4)^T\) is exactly the eigenvector associated with the eigenvalue 1. So, a deviation from \(x^*\) in the direction of \(V\) will bring the system in a different limit cycle. According to the definition of a limit cycle, the system will always stay there (assuming no more disturbances), so it will never return to the original limit cycle \(x^*\) anymore. The deviation from \(x^*\) will neither grow nor shrink, this is where the eigenvalue 1 comes from.

Both eigenvalues 0 result from the fact that the stride function \(S\) does not necessarily need the start of the step as an input; it can have any state that occurs during the step. In all cases it just integrates until the start of the next step. So, not \(x^*\) is given as input but the state \(x^* + \frac{dt}{dt} |_{x^*} \delta t\), a very short time after the start of the step, the stride function will still return \(x^*\). So, this deviation in the input does not result in any deviation in the output, hence the eigenvalue 0. The fact that there are two eigenvalues of 0 with different eigenvectors (one would actually expect only one, namely \(\frac{dt}{dt} |_{x^*}\)) is probably due to linearization, which cancels higher order behaviour.

### 3.4.2 Pole placement

It is obvious that the eigenvalues 1, 0 and 0 cannot be changed, they are just a result of the mathematical structure of the system. That leaves us with one changable eigenvalue, which we should place between \(-1\) and 1 for stable walking behaviour.

For this, we apply state feedback to the system: \(\Delta u_k = -K \cdot \Delta x_k\), where \(K\) is a matrix of state feedback gains. This results in a new linear system:
Feedback | $\Delta \phi_{\text{hip}}$ (%) | $\Delta \dot{\phi}_{\text{hip}}$ (%) | $\Delta \phi_{\text{play}}$ (%) | $\Delta \dot{\phi}_{\text{play}}$ (%)  
---|---|---|---|---
$K_{-0.9}$  | min | max | min | max | min | max | min | max  
| -14 | +1 | -1 | +8 | -48 | +16 | -32 | +58  
$K_{-0.4}$  | -9 | +3 | -2 | +5 | -104 | +84 | -208 | +140  
$K_{\text{one}}$  | -10 | +4 | -3 | +6 | -232 | +102 | -300 | +280

**Table 3.4**: Indication of the stability region of the controller with different state feedback gains.

\[
\Delta x_{k+1} = J_x \Delta x_k + J_u \Delta u_k = (J_x - J_u K) \Delta x_k
\]

Using the Matlab command `place(Jx, Ju, P)` we can calculate $K$ such that the eigenvalues of $(J_x - J_u K)$ are equal to the vector $P$. We use $P = (p, 1, 0, 0)^T$ where $p$ is the target value of the freely choosable eigenvalue.

It is expected that eigenvalues close to zero will result in a large $K$, because the controller must influence the system more rigorously. A large $K$ will in its turn lead to a larger input signal $u$, which increases the chance that non-linearities become noticeable. Therefore, a small $K$ might be preferable. In order to investigate this, two eigenvalues were chosen: $p = -0.9$ and $p = -0.4$.

The resulting feedback gains are:

$K_{-0.9} = \begin{bmatrix} -0.726 & -0.300 & -0.637 & -0.240 \\ 0.557 & 0.196 & -3.683 & -1.272 \end{bmatrix}$

$K_{-0.4} = \begin{bmatrix} -0.718 & -0.297 & -0.698 & -0.262 \\ 0.602 & 0.211 & -4.046 & -1.397 \end{bmatrix}$

The $K$’s are not too large, so this gives good hope that the non-linear system can indeed be stabilized with this feedback gain. Also the $K_{-0.4}$ is indeed a bit larger. The numbers on the second row are much larger than the numbers on the first row, which implies that primarily $\phi_{\text{play}}$ is used for control. So another interesting feedback gain would be one where $\phi_{\text{hip}}$ is not used for control at all:

$K_{\text{one}} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0.602 & 0.211 & -4.046 & -1.397 \end{bmatrix}$

So, if this $K_{\text{one}}$ is used, only the splay angle is adjusted by means of feedback; the hip angle always equal to the input $u_1$.

### 3.4.3 Stability region

Using a controller designed for a linear model on a non-linear one, does not guarantee global stability, only local. The question then is how large the stability region is. One way to find out is to just try all combinations of all possible values of the input as initial values, simulate and see if the system ends up in its limit cycle (in other words: if it does not fall after a few hundred steps). However, this is much work (even for a fast computer) and it is hard to visualize a four-dimensional space anyway.
3.5. STABILIZING THE GAIT WITHOUT EXPLICIT KNOWLEDGE OF THE LIMIT CYCLE

Instead, it was chosen to vary each parameter separately, while leaving the other three at the chosen value of $x^*$. For example, for inspection of the first parameter the vector

$$
x_{\text{init}} = \begin{bmatrix}
1 + n & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} x^*
$$

was used, where $n$ was varied. During the simulation of a few hundred steps, the input of the system were: $u = u^* - K \cdot (x - x^*)$. Simulations were done for all three feedback gains. The results are listed in table 3.4. The numbers in the table show that the walker is quite sensitive for variations in $\varphi_{\text{hip}}$ and $\dot{\varphi}_{\text{hip}}$, for all controllers. The reason is that changing either $\varphi_{\text{hip}}$ or $\dot{\varphi}_{\text{hip}}$ has a huge effect on the internal energy of the walker (much more than changing $\varphi_{\text{splay}}$ or $\dot{\varphi}_{\text{splay}}$), which makes the walker go near a different limit cycle. The controller was not optimized for that limit cycle, so it won’t work very well there.

A remarkable thing is that the simple controller $K_{\text{one}}$ performs best on deviations in $\varphi_{\text{splay}}$ and $\dot{\varphi}_{\text{splay}}$. The cause might be the following: even though the deviations are quite much relatively, the forward dynamics still dominate over the sideways. The $K_{\text{one}}$ controller leaves the forward setpoint alone, so the (correct) forward motion is not touched. (Actually this is somewhat oversimplified; of course there is a coupling between forward and sideways motion, but it is assumed that it is not really big near the limit cycle).

We can conclude that it is indeed possible to stabilize the walker by means of pole placing. The region of stability is quite small for deviations in $\varphi_{\text{hip}}$ and $\dot{\varphi}_{\text{hip}}$ and quite large for deviations in $\varphi_{\text{splay}}$ and $\dot{\varphi}_{\text{splay}}$. It is expected that similar results can be achieved for more complex systems (e.g. with finite-mass feet).

3.5 Stabilizing the gait without explicit knowledge of the limit cycle

In this section it is shown that in the case of this walker, for a limited range of parameters, no explicit knowledge about the limit cycle is needed in order to stabilize it. Instead only one simple characteristic property of any limit cycle is used that follows from symmetry.

3.5.1 Analysis

A special case for the impact equations occurs if the position of the stance leg just before impact is equal but mirrored to the (fixed) input values, which is the position of the pre-impact swing leg:

$$
x^* = \begin{bmatrix}
\varphi^-_{\text{hip}} \\
\dot{\varphi}^-_{\text{hip}} \\
\varphi_{\text{splay}} \\
\dot{\varphi}_{\text{splay}}
\end{bmatrix} = \begin{bmatrix}
-u_1 \\
a \\
u_2 \\
b
\end{bmatrix}
$$

with any positive angular velocities $a$ and $b$. In this case, the legs span a plane (let us call this plane the zm-plane) that is exactly vertical (see figure 3.8). The two impact impulses $p_{\text{add}}$ and $p_{\text{loss}}$ are exactly aligned with the legs, so they also lie in the zm-plane. As both impulses are equal in magnitude, their z-components add up, while their m-components cancel each other out. So only the vertical velocity of the hip will change at impact; forward and sideways velocity are unaffected. This makes the post-impact velocities be equal but mirrored (due to the splay sign change) to the pre-impact velocities, resulting in the following post-impact state:
Figure 3.8: A special case for the impact equations: $\varphi_{\text{hip}}^- = -u_1$, $\varphi_{\text{splay}}^- = u_2$. In this case the legs span a plane that is exactly vertical.

The equations of motion are:

$$
\begin{bmatrix}
\varphi_{\text{hip}}^+
\
\dot{\varphi}_{\text{hip}}^+
\
\varphi_{\text{splay}}^+
\
\dot{\varphi}_{\text{splay}}^+
\end{bmatrix} =
\begin{bmatrix}
u_1 \\
\bar{a} \\
u_2 \\
\bar{b}
\end{bmatrix}
$$

which holds for any angular velocities $a$ and $b$. If we can find values for $a$ and $b$ such that the equations of motion, when initialized with post-impact state $x^+$ will return a new pre-impact state $x^-$, we have designed a limit cycle:

$$
\begin{align*}
x^+ &= \begin{pmatrix} u_1 \\ a \\ u_2 \\ -b \end{pmatrix} \\
x^- &= \begin{pmatrix} -u_1 \\ a \\ u_2 \\ b \end{pmatrix} \\
x^+ &= \begin{pmatrix} u_1 \\ a \\ u_2 \\ -b \end{pmatrix}
\end{align*}
$$

In figure 3.9 (left) a sketch of the path is shown that leads to a limit cycle. The question is if indeed such a path exists.

During the motion, the only force acting on the walker is gravity. Because this is a conservative force, the whole trajectory can also be traveled in the opposite direction, by taking the end state $x^-$ as start point, where the velocities are negated. This reversed trajectory is shown in figure 3.9 (right). Note that the reversed path is exactly the mirror of the normal path; in other words, the path is symmetric around the line $\varphi_{\text{hip}} = 0$. This symmetry implies that the sideways velocity $\dot{\varphi}_{\text{splay}}$ at $\varphi_{\text{hip}} = 0$ must be zero. The inverse is also true: any path that has a $\dot{\varphi}_{\text{splay}} = 0$ at $\varphi_{\text{hip}} = 0$ is symmetric, and thus obeys the demands needed for a limit cycle.
3.5. STABILIZING THE GAIT WITHOUT EXPLICIT KNOWLEDGE OF THE LIMIT CYCLE

\[ x^+ = \begin{pmatrix} u_1 \\ a \\ u_2 \\ -b \end{pmatrix} \quad \varphi_{\text{hip}} = 0 \]

\[ x^- = \begin{pmatrix} -u_1 \\ a \\ u_2 \\ b \end{pmatrix} \quad \varphi_{\text{hip}} = 0 \]

\[ \ddot{\varphi}_{\text{play}} = 0 \quad \text{Foot contact point} \]

\[ \ddot{\varphi}_{\text{hip}} = 0 \quad \text{Stance leg} \]

\[ \ddot{\varphi}_{\text{play}} = 0 \quad \text{Hip trajectory} \]

\[ \ddot{\varphi}_{\text{hip}} = 0 \quad \text{Foot contact point} \]

\[ \ddot{\varphi}_{\text{play}} = 0 \quad \text{Stance leg} \]

\[ \ddot{\varphi}_{\text{hip}} = 0 \quad \text{Hip trajectory} \]

\[ \ddot{\varphi}_{\text{play}} = 0 \quad \text{Foot contact point} \]

\[ \ddot{\varphi}_{\text{hip}} = 0 \quad \text{Stance leg} \]

Figure 3.9: Top view of the hip trajectory. The thick arrows show the velocity direction in the points \( x \). The sideways motion is exaggerated for clarity. Left: the normal trajectory; right: the trajectory traveled in opposite direction.

It is easy to prove there are infinitely many limit cycles for a given input \( u \). Let us first assume the movements in forward and sideways direction are uncoupled. For a normal step we can choose any forward velocity \( a \) as long as it is fast enough to push the walker over its highest point. This \( a \) determines the step time \( t_{\text{step}}(a) \), i.e. how long it takes to go from point \( x^+ \) to \( x^- \). In half the step time, the sideways acceleration decreases the sideways velocity \( \dot{\varphi}_{\text{play}} \):

\[ \dot{\varphi}_{\text{play}}(t) = -b + \int_0^{t_{\text{step}}(a)/2} \frac{1}{m} F(\varphi_{\text{play}}) \, dt \]

For each \( a \) there is exactly one \( -b \) for which \( \dot{\varphi}_{\text{play}}(t_{\text{step}}(a)/2) = 0 \). As infinitely many \( a \)’s can be chosen, each with an associated \( -b \), there are infinitely many limit cycles for a given \( u \). All these limit cycles are symmetric around \( \varphi_{\text{hip}} = 0 \). So we can conclude this analysis by stating that a trajectory is part of a limit cycle if (introducing \( Y \) as a shorter notation):

\[ Y := \dot{\varphi}_{\text{play}} \mid \varphi_{\text{hip}} = 0 = 0 \]

### 3.5.2 Control

Imagine the walker is traveling a path that is close to its limit cycle (for a given input \( u_{\text{nom}} \)). Then \( Y \) will be close to zero. We can use this variable to control the input \( u \) such that the walker comes closer to its limit cycle at the next step.

Figure 3.10 shows what happens with an uncontrolled system when \( Y \) is positive in step 1 (remember that the positive direction for \( \dot{\varphi}_{\text{play}} \) is ‘inward’, so each step starts with a negative \( \dot{\varphi}_{\text{play}} \)).

A positive \( Y \) in step 1 makes \( \dot{\varphi}_{\text{play}} \) at the start of step 2 too large (too negative). A controller can compensate for this by placing the foot a little more ‘outward’ at the start of step 2 (it thus increases \( \dot{\varphi}_{\text{play,sw}} = u_2 \)). This increases the sideways acceleration (as that is roughly proportional to the sine of \( \varphi_{\text{play}} \)), so that the sideways velocity returns to its original value again. In order to keep the same step time for step 1, the leg should not only be placed more outward, but also less far forward (so that the vertical distance between the swing foot and the hip remains the same).

A control rule to stabilize the walker can now be set up:

\[ u_2 = \varphi_{\text{play,sw}} = u_{\text{nom,2}} + K \cdot Y \]

\[ u_1 = \varphi_{\text{hip,sw}} = -\arccos \left( \frac{z_0}{\cos \varphi_{\text{play,sw}}} \right) \]

where \( z_0 = \cos u_{\text{nom,1}} \cdot \cos u_{\text{nom,2}} \)

where \( K \) is the control gain. The only variables that should be known to the controller are the input variables \( u_{\text{nom}} \). So no explicit knowledge about a limit cycle is needed in order to stabilize...
CHAPTER 3. A VERY SIMPLE 3D WALKER

Figure 3.10: The hip trajectory of an uncontrolled walker. A positive $Y$ at step 1 results in a too large $\dot{\phi}_{\text{play}}$.

Figure 3.11: Block diagram of the controller described in section 3.5.2

Figure 3.11 shows a block diagram of the constructed system. Note that this is not a standard discrete system because the variable $Y$ is used for feedback, not the states.

In Matlab the dashed block was implemented as one function called $\text{stosc}(x,u,K)$ ('step-to-step-controlled'). This system was linearized around different parameter settings, amongst others around the reference point $(x^*, u^*)$ from table 3.1, with different $K$'s. Some of the eigenvalues are shown in table 3.5. They are also plotted in root locus graphs in figure 3.12.

Unfortunately this controller cannot stabilize the walker over the whole range of forward velocities; e.g. our reference velocity $v = \dot{\phi}_{\text{hip}}|_{\phi_{\text{hip}}=0} = 0.7$ leads to an unstable system, no matter which $K$ is chosen.

According to the numbers in the table, with an increasing velocity the system can be stabilized easier. $v = 0.92$ is the lowest value for which all poles come within the unit circle, so the system can be stabilized. With an increasing velocity $v$, the largest eigenvalue of the uncontrolled system ($K = 0$) comes closer to the unit circle, but it only approaches asymptotically; the system will not get stable by itself (see the column on $v = 100$).

The reason that the system becomes more stable (and easier to stabilize) with an increasing $v$ is simple: as the velocity increases, the step time $t_{\text{step}}$ decreases. So per step the system has less time to fall and the controller can intervene more times per second.
3.5. STABILIZING THE GAIT WITHOUT EXPLICIT KNOWLEDGE OF THE LIMIT CYCLE

Table 3.5: Resulting eigenvalues of the linearized system described in section 3.5.2. First a state $x$ around which the linearization was done was found by using the function findlimitcycle($u, v$) with the parameters listed (where $v = q_{hi p}$). Then the function stoc($x, u, K$) was linearized by using Matlab’s numjac function. Apart from the eigenvalues listed, the eigenvalues of 0 and 1 were present too (see section 3.4.1).

$V = 0.7 \text{ rad/s}$

$V = 0.92 \text{ rad/s}$

$V = 1.5 \text{ rad/s}$

$V = 100 \text{ rad/s}$

Figure 3.12: Root loci for the controller described in section 3.5.2, for different forward velocities $v$. For $v \geq 0.92 \text{ rad/s}$ a feedback gain $K$ exists that stabilizes the system.
As the controlled system is unstable in the reference limit cycle \((x^*, u^*)\), it makes no sense to do stability range analysis around that point. Instead it is chosen to analyze the situation where \(u = (-0.3, 0.05)^T\) and \(v = 1.5\). The \(K\) is chosen that has the smallest non-complex eigenvalues: \(K = 0.2238\). Again each element of the initial state \(x\) is varied separately and a few hundred steps are simulated. If the walker falls within this time, the initial state is said to be outside the stability region; if it still walks, the state is said to be inside the stability region. Results are shown in table 3.6.

### 3.6 Comparison between the controllers

For a proper comparison, also conventional controllers were designed around the same point, just as described in section 3.4. It can be seen that the conventional controllers perform better with this faster walker, which was expected. However, the new controller outperforms the conventional controllers on deviations in \(\phi_{hip}\) and \(\dot{\phi}_{hip}\). This is not so strange, because a deviation in one of these variables hugely affects the internal energy (potential or kinetic). The system always keeps its internal energy at the same level, so the walker can never return to its original limit cycle (that has a different energy level). Instead, it must try to stabilize around a point the controller was not optimized for. As the new controller is not optimized for any limit cycle in particular, it does not have this problem. A drawback of the new controller however is that it cannot stabilize all possible gaits (e.g. the reference gait).

A controller based on symmetry can only be used if indeed there is symmetry in the gait. In general the gait of a (complex) walker is not symmetric, so this type of control would be useless for such walkers. However, the idea behind it, basing the control on a characteristic that holds for a wide range of gaits, can be a powerful tool for controlling walkers that must do more than only walk at one speed. Moreover, it is expected that this controller, with a few modifications (e.g. compensation for asymmetry) can be used in more complex walker models too.
Chapter 4

Conclusions and recommendations

4.1 Conclusions

During this master’s project research was done on how to stabilize 3D walking robots. Two models were investigated: an 8-degrees-of-freedom 3D walker, produced with 20-sim’s 3D Mechanics Editor and a very simple 3D walker consisting of one point mass and to stiff massless legs.

With the help of the produced ‘aligner’ block, the motion of the 8-dof walker was restricted to two dimensions. With some trial and error, parameters were found that lead to a stably looking gait. A new controller was developed that controls the forward velocity of the walker by means of ankle actuation. Even though the controller was very simple, the walker was walking stably and indeed changed its speed.

For simulation of the 8-dof walker in three dimensions the legs were positioned slightly inwards, and the feet were pointed straight up. This was done to give the walker an easier to understand behavior, which should be less complex to control. It appeared however that the system was still quite complex, and it was difficult to build a good controller for it from scratch. The best results were achieved with a simple trajectory prediction system; the walker did ten steps with it before falling (as opposed to two if the controller was disabled). By manual tuning no set of parameters was found that stabilizes the gait completely.

Because the main focus of the master’s project is to create a good controller and understand how it works (instead of doing endless experiments in order to find one set of parameters with which the system happens to walk), it was decided to concentrate on a much simpler walker model.

The ‘very simple 3D walker’ consists of a point mass as a hip and two massless legs. The swing leg’s position is directly set via the system’s inputs. The stance leg position results from the movement of the hip. Foot impact is modeled as being instantaneous. At the moment of foot impact, energy is added to the system by means of an impulse working along the pre-impact stance foot. The amount of energy added is exactly equal to the amount of energy lost during that step, so the total energy of the walker is kept at the same level.

The model was simulated in 20-sim as well as in Matlab. The stride function $x_{k+1} = S(x_k, u_k)$, programmed in Matlab, was used to design different controllers. It was first linearized at a chosen limit cycle and analyzed. Three of the four eigenvalues (their values: 0, 0 and 1) happened to be fixed and could not be changed by means of feedback control. The reason for this was analyzed and described. The fourth eigenvalue could be changed however. In order to put that eigenvalue within the unit circle so that the resulting system is stable, the feedback gain did not need to be so big. Three feedback gains were chosen and their performance on the nonlinear system was analyzed by means of a stability region check. All controllers performed well on deviations in the splay angle, but they were not so robust against deviations in the hip angle.

Next a new type of controller was developed that makes use of symmetry properties of the
walker: it relates the splay angle of the swing leg directly to the sideways velocity of the hip at the moment it passes the highest point of the trajectory of that step. It was found that this controller stabilizes the walker well for velocities a little higher than the chosen reference velocity. Contrary to the ‘normal’ controllers, this controller showed good behaviour against deviations in the hip velocity. The differences between the controllers were analyzed and explained as follows. The conventional controllers were linearized around one limit cycle where the walker has a certain amount of internal energy. As the energy of the walker is kept constant by the energy addition mechanism, a different initial energy level implies that the walker can never return to the original reference level which is the level the conventional controllers were designed for. As forward motion (ϕ_{hip}, ˙ϕ_{hip}) dominates the energy level of the walker, no much deviation is allowed. The new controller was not made for just one limit cycle, so it can cope better with large deviations in the forward states. The controllers performed well on the simple walker, and it is expected that using similar strategies can also help in stabilizing more complex walkers.

4.2 Recommendations

This research on stabilization of walking robots has by no means been finished. The 8-dof walker can only walk stably in 2D, using a simple controller for 3D stability failed. The very simple walker can walk without falling in three dimensions, and it has a reasonably large region of stability. However, the model is much too simple. In order to produce a real 3D walking robot in the future that is stable, more complex models will have to be studied and stabilized first. There are hundreds of possibilities to improve the model, starting from the very simple walker. A few suggestions and considerations for future research are listed below. Not all of these ideas follow directly from the work described in this report; some are just ideas of the author that might be helpful to stabilize and improve walking machines.

4.2.1 Add finite mass feet

When adding mass or inertia to the legs of the very simple walker, they will show dynamic behaviour. This is interesting because then the natural oscillation frequency of the legs can be used as a basis for the step frequency. It is suggested to still use an actuator in the hip to push the leg forward, as it is expected to increase the ground clearance as well as stability [13]. In a real walking robot the actuator will be needed anyway to overcome fiction. By adding masses the symmetry is not destroyed, so the controller should still work (possibly with some minor adaptations).
4.2.2 **Add larger feet**

Point feet are easy from a mathematical point of view, but they are not energy-efficient. Large feet are much better [8]. Much experimentation can be done on feet. It is expected that the shape of feet and the position of the ankle joints within the feet hugely affect the walking behaviour and potentially destroys symmetry of the gait.

4.2.3 **More realistic energy addition**

The current very simple walker model injects exactly as much energy each step as is lost. Humans do not do this (if they are pushed and dissipate the excess of energy at the next foot impact, they don’t perform an extra firm toe-off to keep the excess of energy), so probably also for a walking robot it is better to not do that. A more realistic setting is if the energy injection is independent on the energy loss. For example, a fixed amount of energy could be injected each step. Using the amount of injected energy as an extra input probably makes the walker more flexible and will probably increase robustness.

4.2.4 **More degrees of freedom**

Although it is harder to control a higher-degrees-of-freedom walker, these extra degrees of freedom (e.g. knees, multi-dof ankles) will be needed to give a walking robot enough flexibility to operate well in a human environment. Also, it is expected that, if the joints are controlled correctly, a larger stabilization region can be achieved than with a very simple walker.

4.2.5 **More research on the upper body**

At the University of Delft two walking robots with an upper body and a bisecting hip mechanism [13] were built: the four-legged Max and the two-legged Denise. The authors showed (both in theory and in practice) that walkers with such a mechanism for stabilizing the upper body can walk stably. However, as can be observed in videos of these walkers, the upper body tilts backwards each step, which looks unnatural. It would be interesting to see what causes it and if it can be solved (preferably in a passive way). Another interesting question is if it is possible to adjust the gait in any way by means of tilting the upper body forward or backward on purpose.

4.2.6 **Standing still, walking slowly and turning**

These are things that a walking robot should be capable of before it can operate in an environment surrounded by humans. Dynamic walkers are usually built to walk at one velocity, it would be very interesting to see if it is possible to make a walker that can walk at different velocities with ease.

Making a dynamic walker turn is a topic that has not had much attention yet in the literature. Probably actuators in the ankles will be needed, but it might be possible to also use the passive dynamics of the robot to reinforce the turning.

4.2.7 **Many more things...**

After fifteen years of research on dynamic walkers, the produced machines show nice, natural walking behaviour. However, that is all they do. Things like standing still, climbing stairs, changing direction, stepping over obstacles, coping with rough terrain, withstanding pushes, being able to walk with a load (that influences the center of mass), etc. are all unsolved problems. Hopefully during the coming years solutions for some of these problems will be developed so that the dynamic walkers of the near future are stable enough to walk around by themselves.
CHAPTER 4. CONCLUSIONS AND RECOMMENDATIONS
Appendix A

20-sim tips and tricks

The simulation program 20-sim [2], by Controllab Products B.V. is a very powerful and flexible simulation tool. One of the strengths of the program is its ability to combine different model representations seamlessly, such as block diagrams, IPMs and bondgraphs. Also hierarchical modeling is a powerful feature: a block can consist of multiple interconnected subblocks. The bottom level of the hierarchy is always formed by a set of mathematical equations in a language called SIDOPS+. It should be noted that SIDOPS+ is not a sequential language, hence programming in 20-sim is different from programming in C or Pascal. The help of 20-sim gives some information about the characteristics of the language, but it lacks tips for advanced programming (actually, the information partially is available in the help, but it is hard to find if one does not know the name of the command that has to be used). In this appendix a few tips, strategies and work-arounds are described that are really useful to know, especially if one makes large and complex models.

All code snippets were checked on 20-sim 3.6.0.4 and were found to behave according to what is written here.

A.1 Execution order

20-sim internally rearranges all equations of the whole model. It does this in such a way that for each equation it has only known values at the right-hand side. If necessary it even rewrites the equations themselves. One can see the order of execution of the equations by choosing from the editor’s pull-down menu: model|show equations.

With this in mind, one can easily verify that the following three equation models (left) are completely identical (the sequence of equations that 20-sim generated internally is shown on the right).

```
1 variables
2    real a, b;
3 equations
4    a = sin(time);
5    b = a * 2;
```

```
1 output equations:
2    a = sin(time);
3    b = a * 2.0;
```
However, equation-reordering is not done in code blocks, such as in if-statements. See also the 20-sim-help on this.

### A.2 Conditional assignments

From ‘normal’ programming languages the next conditional code is well-known:

```plaintext
if (time < 3) then
  a = sin(time);
else
  a = cos(time);
end;
```

Although this is correct 20-sim code, it is not a preferable notation. The two statements (line 2 and 4) are inside an if-statement and are thus seen as a code block. Hence, the equations are not included in the equation reordering process.

In the next example (left), 20-sim does not notice that a is always calculated (this is intentional behaviour!), so it does not see the need to rewrite the equation in line 4. Instead, 20-sim generates wrong sequence of equations (right):

```plaintext
variables
  real a, b;
equations
  a = b / 2;
a = sin(time);
```

```plaintext
static equations:
  b = 0.0;
  a = b * 2.0;
```

```plaintext
dynamic equations:
  if (time < 3.0) then
    a = sin(time);
  else
    a = cos(time);
  end;
```

In order to prevent such problems (they can come in easily and unnoticed in large models, and they are hard to find), it is strongly advised to use the if-construction as shown below (left) whenever possible. The assignment of a is now done outside the if-construction, so 20-sim knows that it is always assigned a value (the else case is obligatory). The (correct) equations generated internally by 20-sim are on the right again.
A.3 “UNABLE TO BREAK ALGEBRAIC LOOP FOR…”

Try the following code:

```plaintext
variables
  real a, b;
output equations:
  a = if (time < 3.0) then sin (time) else cos (time) end;
  b = a / 2.0;
```

```
A.3 “Unable to break algebraic loop for…”

Compiling it will give you an error: unable to break algebraic loop for (state==1 and changestate). You might wonder which algebraic loop is meant. Well, none. The problem here is that the variable state has no initial value. So adding initialequations / state=1; to the code solves the problem. It occurs quite often that this error results from an initialization that was forgotten.

A.4 Instantaneous state changes: resint and event

Sometimes one wants to be able to set a state (which is the output of an integrator) to a certain value. For example, this is used in section 3.2.2 to set some velocities instantly to zero. One could just do the following:

```plaintext
variables
  real a, v;
equations
  a = 1;
  v = int(a);
  if (v>0.5) then v = 0; end;
```

When simulating this with the default simulation parameters, the value of v first increases as expected, but it drops to zero after it reached 0.35 and stays there (see figure below).

![Graph](image-url)
Appendix A. 20-Sim Tips and Tricks

We see two things happen:

- After changing the state the integrator does not work anymore.
- The value of v has not become 0.5 before resetting to 0.

The reason for the first issue is that 20-sim does not really notice you change the state yourself; it assumes that only the integrator modifies it. Hence, after you change the state it does not really have an idea anymore about the value of the state. So, please make sure this does not happen:

Never change a state (the output of an integrator) by just assigning a new vale to it (as in: v=0).

The correct way to do it, is to make use of a special version of the integrator: the resettable integrator, or resint. This command has specifically been designed to allow instantaneous state changes. Using the resint results in the following code:

```plaintext
1 variables
2   real a, v;
3   boolean v_is_larger_than_05;
4 equations
5   a = 1;
6   v_is_larger_than_05 = (v > 0.5);
7   v = resint(a, 0, v_is_larger_than_05);
```

A simulation plot of the code is given below.

We see that this works much better, but there is still a problem with finding the point (v > 0.5). This is due to the adaptive step size that 20-sim performs (at least, for some integration methods). The solution is to tell 20-sim something special will happen at v = 0.5 (or: v - 0.5 = 0). It will then search for the exact time that this happens, and will adjust its step size accordingly. This is done with the event command. So, the correct version of the code finally becomes:

```plaintext
1 variables
2   real a, v;
3   boolean v_is_larger_than_05;
4 equations
5   a = 1;
6   v_is_larger_than_05 = event(v-0.5);
7   v = resint(a, 0, v_is_larger_than_05);
```

And indeed the simulation plot shows exactly what we expected to see:
A.5. DOUBLE INTEGRATION AND RESINT

Other versions of the event command exist too: eventup, eventdown and timeevent. See the help of 20-sim for reference.

A.5 Double integration and resint

Unfortunately due to a bug (at least in 20-sim version 3.6.0.4 and earlier), the next code (representing a frictionless bouncing ball) does not work:

```
variables
real a, v, x;
real minusv;
boolean contact;
equations
contact = eventdown(x);
a = -9.81;
minusv = -v;
v = resint(a, minusv, contact);
x = int(v,1);
```

The problem is that the state \( v \) is not reset correctly, the reason is that the variable \( v \) is internally both a state (line 9) and a rate (line 10). This seems to give problems.

The solution is to modify the code such that it is not exactly the same variable that is used for the state and the rate, simply by using a temporary variable \( v_{\text{temp}} \) (the multiplication makes sure the optimizer does not remove the assignment):

```
variables
real a, v, x, v_temp;
real minusv;
boolean contact;
equations
contact = eventdown(x);
a = -9.81;
minusv = -v;
v_temp = resint(a, minusv, contact);
v = 1 * v_temp;
x = int(v,1);
```
The people of Controllabs Products B.V. know about this bug, so it is expected that it will be fixed soon in a newer version of 20-sim.

### A.6 Variables used in models of the 3D Mechanics Editor

The 3D Mechanics Editor makes complex 20-sim code. In order to edit it, one has to understand which variable exactly means what, how it is represented etc. A non-exhaustive list of variable names used by the 3D Mechanics editor is presented here. $\Psi_0$ represents the world’s coordinate frame.

<table>
<thead>
<tr>
<th>Var. name</th>
<th>Represents</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>General</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$t_{3_00}$</td>
<td>$T_{3_0}^0$</td>
<td>The twist of body 3 relatively to the world frame ('body 0'), represented in the coordinate frame $\Psi_0$. This is a 6-element column vector.</td>
</tr>
<tr>
<td>$tT_{3_00}$</td>
<td>$(T_{3_0}^0)^T$</td>
<td>Transpose of $t_{3_00}$.</td>
</tr>
<tr>
<td><strong>Floating body (e.g. Body1)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Body1\AbsHInit</td>
<td>$H_{1\AbsH}^0$</td>
<td>Initial position of Body 1, represented in $\Psi_0$.</td>
</tr>
<tr>
<td>Body1\P0,Initial</td>
<td>$P_{1\P0}^0$</td>
<td>Initial impulse of the Body 1. Compare this to a wrench (not to a twist): The velocity of the body is equal to the velocity the body would get if a ‘dirac-pulse wrench’ would be applied.</td>
</tr>
<tr>
<td><strong>Any body (e.g. Body2)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Body2\AbsH</td>
<td>$H_{2\AbsH}^0$</td>
<td>Homogeneous matrix (containing position and orientation information) of Body 2, expressed in $\Psi_0$.</td>
</tr>
<tr>
<td><strong>Any hinge point on a body (e.g. hingepoint H4 on Body1)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Body1H4\AbsH</td>
<td>$H_{H4}^0$</td>
<td>Homogeneous matrix, containing position and orientation information about the hingepoint, expressed in $\Psi_0$.</td>
</tr>
<tr>
<td>Body1H4.e</td>
<td>$T_{H4}^{H4_0}$</td>
<td>Flow: The twist of this point (and thus of any point of Body 1), relative to $\Psi_0$, expressed in local coordinates $\Psi_{H4}$ (so not expressed in $\Psi_0$). Calculate $T_{H4}^{H4_0}$ with: $t_{4_00} = \text{Adjoint}(\text{Body1H4}\AbsH) \ast \text{Body1H4.e}$</td>
</tr>
<tr>
<td>Body1H4.e</td>
<td>$(W_{H4}^H)^T$</td>
<td>Effort: The transpose of the wrench exerted on Body 1, represented in local coordinates $\Psi_{H4}$ (so not expressed in $\Psi_0$). Note that this variable is stored as a column vector (though officially a wrench is a row vector).</td>
</tr>
</tbody>
</table>
Appendix B

Floor code

This appendix deals with the 20-sim model for the floor. As shown in the picture below, the compliant floor model consists of two subblocks. The source of both subblocks is listed next.

B.1 The CalculateContactPoint block

/* Equation Submodel
Calculates the contact point as the point of the floating object that is closest to the floor (of actually: has the lowest Z-component). For a sphere this is really easy: just take the H matrix of the sphere and subtract the radius.

s = ellipse
p = contact point

AbsH = H_s^0, so the position of the reference point of s, expressed in world coords. Hp_0 is actually H_p^0, so the position of the contact point expressed in world coordinates.
Hs_p is actually H_p^1, so the position of the contact point expressed in object coordinates.
*/

variables
    real H_gndEll[4,4];
    real Hsphere[4,4];
    real xAxis[4], yAxis[4], Znew3[3], Znew4[4];
    real p_foot[4];
    real p_closest[3];
parameters
    real rx = 0.15;
real ry = 0.01; // simulate a point contact; was: 0.075;
real rz = 0.01; // simulate a point contact; was: 0.05;

// Calculate the point of the ellipse that is the closest to the
// ground (assuming the ellipse is above the ground, not under it)
// thus the point that has a minimum Z-coordinate.

// Needed: H_ellGnd = H matrix that represents the position of the
// ellipse expressed in world coordinates (H_ellipse^ground)
H_gndEll = inverseH(AbsH);

// transformation matrix to reshape the world such that the
// ellipse becomes a sphere:
Hsphere = [1/ry, 0, 0, 0; 0, 1/rx, 0, 0; 0,0,1/rz,0;0,0,0,1];

// For the next two lines: the 0 as the last element of the axis
// is intentional, because we are only interested in rotations, not
// in translations. We also could have made the position part of
// the H matrix zero, but this is faster.

xAxis = Hsphere * H_gndEll * [1; 0; 0; 0];
yAxis = Hsphere * H_gndEll * [0; 1; 0; 0];

Znew3 = cross(xAxis[1:3], yAxis[1:3]);
Znew4 = [-Znew3[1]; -Znew3[2]; -Znew3[3]; norm(Znew3)/norm(Znew3)];
p_foot = inverse(Hsphere) * (Znew4);

// now p_foot is the point on the ellipse that is the closest to the
// ground, expressed in foot frame coordinates.

p_closest = AbsH[1:3,1:4]*p_foot;

B.2 The CalculateForce block

variables
  boolean contact;
  boolean contactevent;
  real err[6];
  real Tp[6]; // Twist of the floating body expressed in the contact >>
  // point coordinate frame
  real Wp[6]; // Wrench on the floating body expressed in the contact >>
  // point coordinate frame
  real Fn; // Normal force = Wp[6]. Is separate because we need it more >>
  // often.
  real omega_z, v_tr; // v_tr = v-translation the speed of point p over >>
  // the plane
  real Fr; // rotational friction force;
  real Ft; // translational friction force;
  real ContactSize;
  real FloorIsOn;
parameters
  real zeros[6]=[0;0;0;0;0;0];
real $K_p = 100000.0$, $K_d = 600000.0$;
boolean $true_6 = [true; true; true; true; true; true]$;

// Below: all parameters for the rotational friction
real rotational_friction$\mu_{st} = 0.04$;
real rotational_friction$\mu_c = 0.02$;
real rotational_friction$\mu_v = 0.002$;
real rotational_friction$slope = 1000.0$;
real rotational_friction$\omega_{z_{st}} = 0.02$;
real translational_friction$\mu_{st} = 0.5$;
real translational_friction$\mu_c = 0.25$;
real translational_friction$\mu_v = 0.025$;
real translational_friction$slope = 1000.0$;
real translational_friction$v_{st} = 0.02$;

intialequations
contact=false;
FloorIsOn = On;
equations

// P is the twist/wrench expressed in coordinates of the object.
// We can calculate the P in coordinates of the contact
// point by using some adjoint matrices:
Tp = Adjoint$(H_{s_p}) * P.f$;

contactevent=eventdown$(H_p_0[3,4])$; // Note: plotting contactevent does not work.
err = resint$(Tp, zeros, true_6$ and contactevent$,$ zeros$)$;

// err is now the displacement of the contact point, where only
// slipping is seen as a displacement, not rolling.
if (contactevent) then contact = true; end;

if (eventup$(H_p_0[3,4])$) then contact=false; end;

// We can turn on/off the floor, but it would be wrong to turn it on
// when the foot is under the floor. So we need a check for that. If
// we should turn the floor on (indicated by the input signal On
// that is turned true), we should wait for the foot to become above
// the floor (measured by the contact variable being false). Only
// then we are allowed to turn on the floor. The variable FloorIsOn
// contains the current status of the floor.
if (not On) then FloorIsOn = false; end;
if (On and (not FloorIsOn) and (not contact)) then FloorIsOn = true; end;

// So now we have an error in which the three last components
// indicate what the position error due to slip is. It is this error
// for which we must compensate. As the power P is already in
// coordinates of the contact point, we can easily assign forces to
// this. We use the hunt-crossley model for the compliant contact
// $F = -kp * z^m + -kd * z^m * zdot$, where we have $n=1$:

// This model has a friction component for rotation around the local
// Z-axis. The formula was taken from the 20-sm Block SCVS-friction.
// Coefficients were very roughly estimated for a football and a
// (flat) human foot. Remarkably, the estimation of the flat human
// foot gave lower coefficients, which is contra-intuitive (the
// contact area is much larger. Therefore for the moment we will use
// the football coefficients:
APPENDIX B. FLOOR CODE

// \( \mu_{st} = 0.04; \mu_c = 0.02; \mu_v = 0.002. \)
//
// Now the model also has translational friction. Parameters: a rough guess based on human feet.
// \( \mu_{st} = 0.5; \mu_c = 0.25; \mu_v = 0.025. \)

\[
Fn = -Kp \times \text{err}[6] - Kd \times \text{err}[6] \times Tp[6]; \quad \text{// Calculate this on beforehand, because we need it for the friction calculations}
\]
\[
\omega_z = Tp[3];
\]
\[
Fr = -Fn \times \left( (\text{rotational_friction}\mu_c + (\text{rotational_friction}\mu_{st} \times \text{abs}(\tanh(\text{rotational_friction}\text{slope}\times \omega_z))) - \text{rotational_friction}\mu_c) \times \exp(-((\omega_z / \text{rotational_friction}\text{omgaz_st})^2)) \times \text{sign}(\omega_z) + \text{rotational_friction}\mu_v \times \omega_z \right);
\]
\[
v_{tr} = \text{norm}(Tp[4:5]);
\]
\[
Ft = -Fn \times \left( (\text{translational_friction}\mu_c + (\text{translational_friction}\mu_{st} \times \text{abs}(\tanh(\text{translational_friction}\text{slope}\times v_{tr}))) - \text{translational_friction}\mu_c) \times \exp(-((v_{tr} / \text{translational_friction}\text{v_st})^2)) \times \text{sign}(v_{tr}) + \text{translational_friction}\mu_v \times v_{tr} \right);
\]
\[
Wp = \text{if} (\text{FloorIsOn and contact}) \text{then} [0;0;Fr;Ft * (Tp[4]/v_{tr});Ft * (Tp[5]/v_{tr});Fn] \text{else} \text{zeros} \text{end};
\]
\[
P.e = \text{transpose}(\text{Adjoint}(Hs_p)) \times Wp;
\]

// For the size of the reference frame arrows that indicate contact:
ContactSize = \text{if} (\text{not On}) \text{then} 0 \text{else}
\text{if} (\text{not FloorIsOn}) \text{then} 0.1 \text{else}
\text{if} (\text{contact}) \text{then} 0.5 \text{else} 0.3 \text{end}
\text{end};

Contact = FloorIsOn and contact;
Appendix C

Derivation of the equations for the very simple 3D walker

In this appendix the whole Maple sheet is shown in which the equations were derived for the very simple 3D walker. First a table is given that shows which variable names in Maple correspond to which variables in this report.

<table>
<thead>
<tr>
<th>Maple name</th>
<th>Report name</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>p, p_en</td>
<td>( \dot{p}<em>{\text{hip}} ) or ( \ddot{p}</em>{\text{hip}} )</td>
<td></td>
</tr>
<tr>
<td>v, v_en</td>
<td>( \ddot{p}_{\text{hip}} )</td>
<td></td>
</tr>
<tr>
<td>phi_hip</td>
<td>( \phi_{\text{hip, st}} ) or ( \phi_{\text{hip, st}}^- )</td>
<td></td>
</tr>
<tr>
<td>phidot_hip</td>
<td>( \phi_{\text{hip, st}}^+ ) or ( \phi_{\text{hip, st}}^- )</td>
<td></td>
</tr>
<tr>
<td>phidotdot_hip</td>
<td>( \phi_{\text{hip, st}}^+ )</td>
<td></td>
</tr>
<tr>
<td>l</td>
<td>( \ell ) or ( \ell^- )</td>
<td></td>
</tr>
<tr>
<td>ldot</td>
<td>( \ell ) or ( \ell^- )</td>
<td></td>
</tr>
<tr>
<td>pnew</td>
<td>( \ddot{p}_{\text{hip}}^+ )</td>
<td></td>
</tr>
<tr>
<td>newphi_hip</td>
<td>( \phi_{\text{hip, st}}^+ )</td>
<td></td>
</tr>
<tr>
<td>newphidot_hip</td>
<td>( \phi_{\text{hip, st}}^+ )</td>
<td></td>
</tr>
<tr>
<td>newl</td>
<td>( \ell^+ )</td>
<td></td>
</tr>
<tr>
<td>newl_dot</td>
<td>( \ell^+ )</td>
<td></td>
</tr>
</tbody>
</table>

The table shows the correspondence between Maple variable names and their corresponding variables in the report. The remarks column provides additional information about the variables, such as their position relative to the stance foot.
Calculations for a simple walker model

Name of this document: stab06_final.mw
This document does some calculations on a very simple walker model: a point mass as hip and two massless legs. The leg angles are defined as hip and splay angles.
These formulas were copied directly into 20-sim (stab06.em)
GvO, 050317
Last update: 050511

> restart;
> with(VariationalCalculus):
> with(CodeGeneration):

Find equations for hip/splay angles

Dynamic Equations (Lagrange)

> svv:= (diff(x(t),t)=xdot,diff(y(t),t)=ydot,
  diff(z(t),t)=zdot):
> svv2:= (phi_hip(t)=phi_hip,
  phi_splay(t)=phi_splay,diff(phi_hip(t),t)=phidot_hip,
  diff(phi_splay(t),t)=phidot_splay,
  diff(diff(phi_hip(t),t),t)=phidotdot_hip,
  diff(diff(phi_splay(t),t),t)=phidotdot_splay):
> p:= {
  x(t) = l * sin(phi_hip(t))*cos(phi_splay(t)),
  y(t) = -l * sin(phi_splay(t)),
  z(t) = l * cos(phi_hip(t))*cos(phi_splay(t)) }

p := { x(t) = l * sin(phi_hip(t))*cos(phi_splay(t)),
y(t) = -l * sin(phi_splay(t)),
  z(t) = l * cos(phi_hip(t))*cos(phi_splay(t)) }

> v := diff(p,t);

v := { -l sin(phi_hip(t)) sin(phi_splay(t)) (d/dt phi_hip(t))
  cos(phi_splay(t))
  - l cos(phi_hip(t)) sin(phi_splay(t)) (d/dt phi_splay(t))
  (d/dt) phi_splay(t)
  (d/dt) phi_hip(t)
  cos(phi_splay(t))
  - l cos(phi_hip(t)) sin(phi_splay(t)) (d/dt phi_splay(t))
  (d/dt) phi_splay(t)
  (d/dt) phi_hip(t)
  cos(phi_splay(t))

> T := simplify(subs(svv,v),1/2*m* (xdot^2 + ydot^2 +
  zdot^2)));
> V := simplify(subs(p,m * g * z(t)))
> L := T - V;
\[
L = \frac{1}{2} m l^2 \left(\left(\frac{d}{dt}\phi_{\text{splay}}(t)\right)^2 + \left(\frac{d}{dt}\phi_{\text{hip}}(t)\right)^2 \cos(\phi_{\text{splay}}(t))^2\right) - m g l \cos(\phi_{\text{hip}}(t)) \cos(\phi_{\text{splay}}(t))
\]

\[
> \ \text{el} := \text{EulerLagrange}(L, t, [\phi_{\text{hip}}(t), \phi_{\text{splay}}(t)]);
\]

\[
el := \left\{-m l^2 \left(\frac{d}{dt}\phi_{\text{hip}}(t)\right)^2 \cos(\phi_{\text{splay}}(t)) \sin(\phi_{\text{splay}}(t))
+ m g l \cos(\phi_{\text{hip}}(t)) \sin(\phi_{\text{splay}}(t)) - m l^2 \left(\frac{d^2}{dt^2}\phi_{\text{splay}}(t)\right) \cos(\phi_{\text{splay}}(t))^2
+ 2 m l^2 \left(\frac{d}{dt}\phi_{\text{hip}}(t)\right) \cos(\phi_{\text{splay}}(t)) \sin(\phi_{\text{splay}}(t)) \left(\frac{d}{dt}\phi_{\text{splay}}(t)\right)
+ \frac{1}{2} m l^2 \left(\frac{d}{dt}\phi_{\text{splay}}(t)\right)^2 + \left(\frac{d}{dt}\phi_{\text{hip}}(t)\right)^2 \cos(\phi_{\text{splay}}(t))^2
- m g l \cos(\phi_{\text{hip}}(t)) \cos(\phi_{\text{splay}}(t))
- \left(\frac{d}{dt}\phi_{\text{hip}}(t)\right)^2 m l^2 \cos(\phi_{\text{splay}}(t))^2 - \left(\frac{d}{dt}\phi_{\text{splay}}(t)\right)^2 m l^2 = K_1\right\}
\]

\[
> \ \text{mm} := \text{map}(x->`\text{if}`(\text{nops}(x)<>2, x=0, 0=0), \text{el}); \ # \ the \ if 
\]

\[
> \ \text{subs}(\text{svv2}, \text{solve}(\text{mm}, \{\text{diff(diff(\phi_{\text{hip}}(t),t),t)}, \text{diff(diff(\phi_{\text{splay}}(t),t),t)}\}));
\]

\[
dynamics := \text{solve}(\frac{\text{d}^2}{\text{d}t^2}(\phi_{\text{hip}}(t)), \frac{\text{d}^2}{\text{d}t^2}(\phi_{\text{splay}}(t)))\)
\]

\[
\text{Impact equations}
\]
In this part equations are derived for calculating new \(\phi_{\text{hip}}(t)\) and \(\phi_{\text{splay}}(t)\) after impact

\[
> \ \text{svv3} := \{ \text{op(svv2)}, \text{new}\phi_{\text{splay}}(t) = \text{new}\phi_{\text{splay}},
\text{new}\phi_{\text{hip}}(t) = \text{new}\phi_{\text{hip}}, \text{new}l(t) = l,
\text{diff(new}\phi_{\text{splay}}(t), t) = \text{new}\phi_{\text{dot}}_{\text{splay}},
\text{diff(new}\phi_{\text{hip}}(t), t) = \text{new}\phi_{\text{dot}}_{\text{hip}},
\}
APPENDIX C. DERIVATION OF THE EQUATIONS FOR THE VERY SIMPLE 3D WALKER

\[ \text{diff(newl(t),t)=newl\_dot} : \]

We already have the speeds in x,y,z; this is v:

\[ v; \]

\[ \begin{align*}
\left( \frac{d}{dt} x(t) \right) &= l \cos(\phi_{\text{hip}}(t)) \left( \frac{d}{dt} \phi_{\text{hip}}(t) \right) \cos(\phi_{\text{splay}}(t)) \\
&\quad - l \sin(\phi_{\text{hip}}(t)) \sin(\phi_{\text{splay}}(t)) \left( \frac{d}{dt} \phi_{\text{splay}}(t) \right), \\
\left( \frac{d}{dt} y(t) \right) &= -l \cos(\phi_{\text{splay}}(t)) \left( \frac{d}{dt} \phi_{\text{splay}}(t) \right) \frac{d}{dt} z(t) = \\
&\quad -l \sin(\phi_{\text{hip}}(t)) \left( \frac{d}{dt} \phi_{\text{hip}}(t) \right) \cos(\phi_{\text{splay}}(t)) \\
&\quad - l \cos(\phi_{\text{hip}}(t)) \sin(\phi_{\text{splay}}(t)) \left( \frac{d}{dt} \phi_{\text{splay}}(t) \right) \\
\end{align*} \]

We need the length-extension to make the whole thing bijective:

\[ > \text{pnew} := \{ \]
\[ \begin{align*}
x(t) &= \text{newl}(t) * \\
&\quad \sin(\text{newphi\_hip}(t)) * \cos(\text{newphi\_splay}(t)), \\
y(t) &= \text{newl}(t) * \sin(\text{newphi\_splay}(t)), \\
z(t) &= \text{newl}(t) * \\
&\quad \cos(\text{newphi\_hip}(t)) * \cos(\text{newphi\_splay}(t)) \}; \\
pnew := \{ x(t) = \text{newl}(t) \sin(\text{newphi\_hip}(t)) \cos(\text{newphi\_splay}(t)), \\
y(t) = \text{newl}(t) \sin(\text{newphi\_splay}(t)), \\
z(t) = \text{newl}(t) \cos(\text{newphi\_hip}(t)) \cos(\text{newphi\_splay}(t)) \} \\
\end{align*} \]

\[ > \text{vnew} := \text{diff(pnew,t)}; \]

\[ \begin{align*}
\left( \frac{d}{dt} x(t) \right) &= \left( \frac{d}{dt} \text{newl}(t) \right) \sin(\text{newphi\_hip}(t)) \cos(\text{newphi\_splay}(t)) \\
&\quad + \text{newl}(t) \cos(\text{newphi\_hip}(t)) \left( \frac{d}{dt} \text{newl}(t) \right) \cos(\text{newphi\_splay}(t)) \\
&\quad - \text{newl}(t) \sin(\text{newphi\_hip}(t)) \sin(\text{newphi\_splay}(t)) \left( \frac{d}{dt} \text{newl}(t) \right) \frac{d}{dt} y(t) = \\
&\quad \left( \frac{d}{dt} \text{newl}(t) \right) \sin(\text{newphi\_splay}(t)) \\
&\quad + \text{newl}(t) \cos(\text{newphi\_splay}(t)) \left( \frac{d}{dt} \text{newl}(t) \right) \frac{d}{dt} z(t) = \\
&\quad \left( \frac{d}{dt} \text{newl}(t) \right) \cos(\text{newphi\_hip}(t)) \cos(\text{newphi\_splay}(t)) \\
&\quad - \text{newl}(t) \sin(\text{newphi\_hip}(t)) \left( \frac{d}{dt} \text{newl}(t) \right) \cos(\text{newphi\_splay}(t)) \\
&\quad - \text{newl}(t) \cos(\text{newphi\_hip}(t)) \sin(\text{newphi\_splay}(t)) \left( \frac{d}{dt} \text{newl}(t) \right) \]
Now equate these two:

\[
\text{sol}:=\text{simplify} \left( \text{solve} \left( \text{subs} \left( v, v_{\text{new}} \right), \left\{ \text{diff} \left( \text{newphi}_\text{hip} \left( t \right), t \right), \text{diff} \left( \text{newphi}_\text{splay} \left( t \right), t \right), \text{diff} \left( \text{newl} \left( t \right), t \right) \right\} \right) \right):
\]

\[
\text{ie}_{\text{no energy}}:=\text{simplify} \left( \text{subs} \left( \{ \text{op} \left( svv3 \right) \}, \text{sol} \right) \right);
\]

\[
\text{ie}_{\text{no energy}} := \{ \text{newphidot}_\text{splay} = \\
\quad -\cos(\phi_\text{hip}) \text{phidot}_\text{hip} \cos(\phi_\text{splay}) \sin(\text{newphi}_\text{splay}) \sin(\text{newphi}_\text{hip}) \\
\quad + \sin(\phi_\text{hip}) \sin(\phi_\text{splay}) \text{phidot}_\text{splay} \sin(\text{newphi}_\text{splay}) \sin(\text{newphi}_\text{hip}) \\
\quad + \cos(\text{newphi}_\text{hip}) \sin(\text{newphi}_\text{splay}) \sin(\phi_\text{hip}) \text{phidot}_\text{hip} \cos(\phi_\text{splay}) \\
\quad + \cos(\text{newphi}_\text{hip}) \sin(\text{newphi}_\text{splay}) \cos(\phi_\text{hip}) \sin(\phi_\text{splay}) \text{phidot}_\text{splay} \\
\quad - \cos(\text{newphi}_\text{splay}) \cos(\phi_\text{splay}) \text{phidot}_\text{splay}, \text{newphidot}_\text{hip} = ( \\
\quad \cos(\text{newphi}_\text{hip}) \cos(\phi_\text{hip}) \text{phidot}_\text{hip} \cos(\phi_\text{splay}) \\
\quad - \cos(\text{newphi}_\text{hip}) \sin(\phi_\text{hip}) \sin(\phi_\text{splay}) \text{phidot}_\text{splay} \\
\quad + \sin(\phi_\text{hip}) \text{phidot}_\text{hip} \cos(\phi_\text{splay}) \sin(\text{newphi}_\text{hip}) \\
\quad + \cos(\phi_\text{hip}) \sin(\phi_\text{splay}) \text{phidot}_\text{splay} \sin(\text{newphi}_\text{hip}) \} / \cos(\text{newphi}_\text{splay}), \\
\text{newl}_\text{dot} = -l ( \\
\quad -\cos(\text{newphi}_\text{splay}) \cos(\phi_\text{hip}) \text{phidot}_\text{hip} \cos(\phi_\text{splay}) \sin(\text{newphi}_\text{hip}) \\
\quad + \cos(\text{newphi}_\text{splay}) \sin(\phi_\text{hip}) \sin(\phi_\text{splay}) \text{phidot}_\text{splay} \sin(\text{newphi}_\text{hip}) \\
\quad + \cos(\text{newphi}_\text{splay}) \cos(\text{newphi}_\text{hip}) \sin(\phi_\text{hip}) \text{phidot}_\text{hip} \cos(\phi_\text{splay}) \\
\quad + \cos(\text{newphi}_\text{splay}) \cos(\text{newphi}_\text{hip}) \cos(\phi_\text{hip}) \sin(\phi_\text{splay}) \text{phidot}_\text{splay} \\
\quad + \cos(\phi_\text{splay}) \text{phidot}_\text{splay} \sin(\text{newphi}_\text{splay}) \}) \}
\]

Energy

The total energy of the system is:

\[
> \text{etot} = \text{subs} \left( svv2, T+V \right) ;
\]

\[
\text{etot} = \frac{1}{2} m \ l^2 \ (\text{phidot}_\text{splay}^2 + \text{phidot}_\text{hip}^2 \cos(\phi_\text{splay})^2) \\
\quad + m \ g \ l \ \cos(\phi_\text{hip}) \ \cos(\phi_\text{splay})
\]

Energy addition during impact (and calculating newphi_\text{hip} and newphi_\text{splay})

In order to walk forever, we need to add energy each step. We want to do this by applying a force (or momentum/impuls') exactly at foot impact time. This force is a force that can be interpreted as resulting from the new swing leg being extended (simulating the extension of the human ankle during normal gait). One way to implement this, is directly calculating what the resulting angular velocities will become if such a force is applied that the total energy of the system remains the same. This is what we are going to do here.

Imagine that the (former stance/new swing)leg indeed extends at foot impact. This gives a whole new set of newphi's. We should now find an extension rate (ldot) for which holds that the total kinetic energy of the system remains the same, or in other words: the amount of energy added by this mechanism is identical to the amount of energy lost due to foot impact:

\[
\text{Eadd} = \text{Eloss} \\
\frac{1}{2} \ m \ l \text{dot}^2 = \frac{1}{2} \ m \ \text{newldot}^2
\]
\( \dot{l} = -\text{newldot} \) (indeed, the signs should be opposite)

This extra equation indeed needs one extra independent variable: \( \dot{l} \). It can be found that newldot is dependent on \( \dot{l} \), and can be written as: \( \text{newldot} = a \cdot \dot{l} + b \). This makes the solution quite simple.

\[
\begin{align*}
\text{svv4} & := \{ l(t) = l, \quad \text{diff}(l(t),t) = \dot{l}, \quad \text{op(svv3)} \} ; \\
\text{p_en} & := \{ \\
\quad x(t) &= l(t) \cdot \sin(\phi_{\text{hip}}(t)) \cdot \cos(\phi_{\text{splay}}(t)), \\
\quad y(t) &= -l(t) \cdot \sin(\phi_{\text{splay}}(t)), \quad \# \text{The swing leg is the right leg, hence no minus sign here} \\
\quad z(t) &= l(t) \cdot \cos(\phi_{\text{hip}}(t)) \cdot \cos(\phi_{\text{splay}}(t)) \} ; \\
\text{v_en} & := \text{diff}(\text{p_en},t) ; \\
\text{vnew} & := \{ \\
\quad \frac{d}{dt}x(t) &= \left( \frac{d}{dt}l(t) \right) \sin(\phi_{\text{hip}}(t)) \cos(\phi_{\text{splay}}(t)) \\
\quad &+ l(t) \cos(\phi_{\text{hip}}(t)) \left( \frac{d}{dt}\phi_{\text{hip}}(t) \right) \cos(\phi_{\text{splay}}(t)) \\
\quad &- l(t) \sin(\phi_{\text{hip}}(t)) \sin(\phi_{\text{splay}}(t)) \left( \frac{d}{dt}\phi_{\text{splay}}(t) \right), \\
\quad \frac{d}{dt}y(t) &= -l(t) \sin(\phi_{\text{splay}}(t)) - l(t) \cos(\phi_{\text{splay}}(t)) \left( \frac{d}{dt}\phi_{\text{splay}}(t) \right), \\
\quad \frac{d}{dt}z(t) &= l(t) \cos(\phi_{\text{hip}}(t)) \cos(\phi_{\text{splay}}(t)) \\
\quad &- l(t) \sin(\phi_{\text{hip}}(t)) \left( \frac{d}{dt}\phi_{\text{hip}}(t) \right) \cos(\phi_{\text{splay}}(t)) \\
\quad &- l(t) \cos(\phi_{\text{hip}}(t)) \sin(\phi_{\text{splay}}(t)) \left( \frac{d}{dt}\phi_{\text{splay}}(t) \right) \} ; \\
\end{align*}
\]

\( \text{vnew} \); \# we already have this one.

\[
\begin{align*}
\frac{d}{dt}x(t) &= \left( \frac{d}{dt}\text{newl}(t) \right) \sin(\text{newphi}_{\text{hip}}(t)) \cos(\text{newphi}_{\text{splay}}(t)) \\
&+ \text{newl}(t) \cos(\text{newphi}_{\text{hip}}(t)) \left( \frac{d}{dt}\text{newphi}_{\text{hip}}(t) \right) \cos(\text{newphi}_{\text{splay}}(t)) \\
&- \text{newl}(t) \sin(\text{newphi}_{\text{hip}}(t)) \sin(\text{newphi}_{\text{splay}}(t)) \left( \frac{d}{dt}\text{newphi}_{\text{splay}}(t) \right), \\
\frac{d}{dt}y(t) &= \left( \frac{d}{dt}\text{newl}(t) \right) \sin(\text{newphi}_{\text{splay}}(t)) \\
&+ \text{newl}(t) \cos(\text{newphi}_{\text{splay}}(t)) \left( \frac{d}{dt}\text{newphi}_{\text{splay}}(t) \right), \\
\frac{d}{dt}z(t) &= \left( \frac{d}{dt}\text{newl}(t) \right) \cos(\text{newphi}_{\text{hip}}(t)) \cos(\text{newphi}_{\text{splay}}(t))
\end{align*}
\]
\[
- \text{newl}(t) \sin(\text{newphi}_\text{hip}(t)) \left( \frac{d}{dt} \text{newphi}_\text{hip}(t) \right) \cos(\text{newphi}_\text{splay}(t))
\]
\[
- \text{newl}(t) \cos(\text{newphi}_\text{hip}(t)) \sin(\text{newphi}_\text{splay}(t)) \left( \frac{d}{dt} \text{newphi}_\text{splay}(t) \right)
\]

\[
> \text{newdot} := \text{collect} \left( \text{simplify} \left( \text{solve} \left( \text{subs} \left( \text{svv4}, \text{subs} \left( \text{v_en}, \text{vnew} \right) \right), \\{\text{newl}\_\text{dot}, \text{newphidot}_\text{hip}, \text{newphidot}_\text{splay} \} \right) \right), \text{ldot} \right);
\]

\[
\text{newdot} := \{ \text{newl}\_\text{dot} = \left( \cos(\text{newphi}_\text{splay}) \cos(\text{newphi}_\text{hip}) \cos(\phi_\text{hip}) \cos(\phi_\text{splay}) + \cos(\text{newphi}_\text{splay}) \sin(\phi_\text{hip}) \cos(\phi_\text{splay}) \sin(\text{newphi}_\text{hip}) \right. \]
\[
- \sin(\phi_\text{hip}) \sin(\text{newphi}_\text{hip}) \ldot \right) \left. - \cos(\text{newphi}_\text{splay}) \cos(\text{newphi}_\text{hip}) \sin(\phi_\text{hip}) \sin(\phi_\text{splay}) \right. \]
\[
+ \cos(\text{newphi}_\text{splay}) \cos(\text{newphi}_\text{hip}) \cos(\phi_\text{hip}) \cos(\phi_\text{splay}) \right) \}
\]

Now we will have to find the factors \(a, b, c, d, e, f\) such that:

\[
\text{newl}\_\text{dot} = a \ast \ldot + b
\]
\[
\text{newphidot}_\text{hip} = c \ast \ldot + d
\]
\[
\text{newphidot}_\text{splay} = e \ast \ldot + f.
\]

\[
> \text{fs} := \left[ \begin{array}{l}
a = \text{coeff} \left( \text{subs} \left( \text{newdot}, \text{newl}\_\text{dot} \right), \ldot, 1 \right), \\
b = \text{coeff} \left( \text{subs} \left( \text{newdot}, \text{newl}\_\text{dot} \right), \ldot, 0 \right), \\
c = \text{coeff} \left( \text{subs} \left( \text{newdot}, \text{newphidot}_\text{hip} \right), \ldot, 1 \right), \\
d = \text{coeff} \left( \text{subs} \left( \text{newdot}, \text{newphidot}_\text{hip} \right), \ldot, 0 \right), \\
e = \text{coeff} \left( \text{subs} \left( \text{newdot}, \text{newphidot}_\text{splay} \right), \ldot, 1 \right), \\
f = \text{coeff} \left( \text{subs} \left( \text{newdot}, \text{newphidot}_\text{splay} \right), \ldot, 0 \right), \\
\end{array} \right]
\]
\[ f = \text{coeff}(\text{subs}(\text{newdot}, \text{newphidot}\_\text{splay}), \text{ldot}, 0)\];

\[ f_{\text{si}} := \{\text{op}(\text{map}(x->\text{rhs}(x)=\text{lhs}(x), f_{\text{s}}))\} ; \]  
# inverted form:

\[ \{ \cos(...)=a \} \]

\[ \cos(\text{newphi}\_\text{splay}) \cos(\text{newphi}\_\text{hip}) \cos(\phi\_\text{hip}) \cos(\phi\_\text{splay}) - \sin(\phi\_\text{splay}) \sin(\text{newphi}\_\text{splay}), b = -\cos(\text{newphi}\_\text{splay}) l \sin(\phi\_\text{hip}) \sin(\text{newphi}\_\text{splay}) \text{phidot}\_\text{splay} \sin(\text{newphi}\_\text{hip}) - \cos(\text{newphi}\_\text{splay}) \cos(\text{newphi}\_\text{hip}) l \cos(\phi\_\text{hip}) \sin(\text{newphi}\_\text{splay}) \text{phidot}\_\text{splay} + \cos(\text{newphi}\_\text{splay}) \cos(\text{newphi}\_\text{hip}) \text{phidot}\_\text{hip} \cos(\phi\_\text{hip}) \sin(\text{newphi}\_\text{splay}) \sin(\text{newphi}\_\text{hip}) - \cos(\text{newphi}\_\text{hip}) \cos(\phi\_\text{hip}) l \sin(\phi\_\text{hip}) \text{phidot}\_\text{hip} \cos(\phi\_\text{splay}) - l \cos(\phi\_\text{hip}) \text{phidot}\_\text{splay} \sin(\phi\_\text{hip}) \sin(\text{newphi}\_\text{hip}), c = (\cos(\text{newphi}\_\text{hip}) \sin(\phi\_\text{hip}) \cos(\phi\_\text{splay}) - \cos(\phi\_\text{hip}) \cos(\phi\_\text{splay}) \sin(\phi\_\text{hip}) \sin(\text{newphi}\_\text{hip})) / (\cos(\text{newphi}\_\text{hip}) l), d = (\cos(\text{newphi}\_\text{hip}) l \cos(\phi\_\text{hip}) \text{phidot}\_\text{hip} \cos(\phi\_\text{splay}) - \cos(\phi\_\text{hip}) \sin(\phi\_\text{hip}) \sin(\text{newphi}\_\text{hip}) \text{phidot}\_\text{splay} + l \cos(\phi\_\text{hip}) \text{phidot}\_\text{hip} \cos(\phi\_\text{splay}) \sin(\phi\_\text{hip}) + l \cos(\phi\_\text{hip}) \sin(\phi\_\text{hip}) \text{phidot}\_\text{splay} \sin(\phi\_\text{hip}))) / (\cos(\text{newphi}\_\text{hip}) l), e = (-\cos(\text{newphi}\_\text{hip}) \sin(\text{newphi}\_\text{hip}) \cos(\phi\_\text{hip}) \cos(\phi\_\text{splay}) - \sin(\phi\_\text{hip}) \cos(\phi\_\text{splay}) \sin(\text{newphi}\_\text{hip}) - \cos(\text{newphi}\_\text{hip}) \sin(\phi\_\text{splay}) \sin(\text{newphi}\_\text{hip}) / l, f = (l \sin(\phi\_\text{hip}) \sin(\phi\_\text{hip}) \text{phidot}\_\text{splay} \sin(\phi\_\text{hip}) \sin(\text{newphi}\_\text{hip}) + \cos(\text{newphi}\_\text{hip}) \sin(\text{newphi}\_\text{hip}) \cos(\phi\_\text{hip}) \sin(\phi\_\text{hip}) \text{phidot}\_\text{splay} - l \cos(\phi\_\text{hip}) \text{phidot}\_\text{hip} \cos(\phi\_\text{hip}) \sin(\phi\_\text{hip}) \sin(\text{newphi}\_\text{hip}) + \cos(\text{newphi}\_\text{hip}) \sin(\text{newphi}\_\text{hip}) \cos(\phi\_\text{hip}) \sin(\text{newphi}\_\text{hip}) \text{phidot}\_\text{splay} - \cos(\text{newphi}\_\text{hip}) l \cos(\phi\_\text{hip}) \text{phidot}\_\text{splay} / l)\]  

\[ \text{newdotabcd} : \text{formulas new velocities newdot, newphidot}\_\text{hip} \text{and newphidot}\_\text{splay}, \text{expressed in a, b, c, and d (that were defined above)} \]

\[ \text{newdotabcd} := [\text{newdot} = a * \text{ldot} + b, \text{newphidot}\_\text{hip} = c * \text{ldot} + d, \text{newphidot}\_\text{splay} = e * \text{ldot} + f] ; \]

\[ \text{newdotabcd} := [\text{newdot} = a \text{ldot} + b, \text{newphidot}\_\text{hip} = c \text{ldot} + d, \text{newphidot}\_\text{splay} = e \text{ldot} + f] \]

Check if all went okay (this is: if newdot and newdot2 are indeed the same functions). Indeed, newdotabcd is equal to newdot.

\[ \text{testvalues} : = \{\text{newphi}\_\text{hip}=0.1, \text{newphi}\_\text{splay}=0.2, \phi\_\text{hip}=0.3, \phi\_\text{hip}=0.4, \text{phidot}\_\text{hip}=0.11, \text{phidot}\_\text{splay}=0.22, \text{ldot}=0.44, l =1\} ; \]

\[ \text{evalf} (\text{subs} (\text{testvalues}, \text{newdot})); \]

\[ \text{evalf} (\text{subs} (\text{testvalues}, \text{subs} (f_{\text{si}}, \text{newdotabcd}))); \]

\[ \text{subs} (\%, \%); \]

\[ \{\text{newdot} = 0.2129554040, \text{newphidot}\_\text{splay} = -0.4247519569, \]
newphidot_hip = 0.1661017830
[ newl_dot = 0.2129554040, newphidot_hip = 0.1661017830, 
  newphidot_splay = -0.4247519569
] [ 0.2129554040 = 0.2129554040, -0.4247519569 = -0.4247519569, 
  0.1661017830 = 0.1661017830 ]

Now calculate ldot that provides the right amount of energy:

```maple
> ldot_en := ldot = subs(solve({op(1,newdotabcd),
  ldot=-newl_dot},{ldot,newl_dot}),ldot);
```

```maple
ldot_en := ldot = \( \frac{b}{1+a} \)
```

```maple
> subs(fs,ldot_en);
```

```maple
ldot := \(-\cos(\text{newphi}_splay) \cdot \sin(\text{phi}_splay) \cdot \text{phidot}_splay \cdot \sin(\text{newphi}_hip) - \cos(\text{newphi}_splay) \cdot \cos(\text{newphi}_hip) \cdot \cos(\text{phi}_splay) \cdot \sin(\text{newphi}_hip) + \cos(\text{newphi}_splay) \cdot \cos(\text{newphi}_hip) \cdot \sin(\text{phi}_splay) \cdot \sin(\text{newphi}_hip) - \cos(\text{newphi}_splay) \cdot \sin(\text{phi}_splay) \cdot \sin(\text{newphi}_splay) - \text{sin(\phi}_splay) \cdot \sin(\text{newphi}_splay)\) /
```

```maple
> testvalues2 :=
  {\text{newphi}_hip=-0.3,\text{newphi}_splay=0.1,\text{phi}_hip=0.3,\text{phi}_splay=0.
  1,\text{phidot}_hip=0.11,\text{phidot}_splay=0.22,\text{l}=1};
```

```maple
testvalues2 := { \text{phi}_hip = 0.3, \text{phidot}_hip = 0.11, \text{phidot}_splay = 0.22,
  \text{newphi}_hip = -0.3, \text{newphi}_splay = 0.1, \text{phi}_splay = 0.1, \text{l} = 1 } 
```

```maple
> evalf(subs(fs,testvalues2,ldot_en));
```

```maple
evalf(subs(%testvalues2,newdot));
```

```
ldot = 0.05610061532 
[ newl_dot = -0.05610061532, newhipdot_splay = -0.2200000000, 
  newphidot_hip = 0.1100000000 ]
```

**Code generation**

Now we collect all equations needed for the 20-sim model. As C has no standard notation
for power \((x^2)\), it uses pow(). This should be replaced manually. For simple variables that
are sqrt()ed, it uses \(e^c\) instead of \(e^2\). This is no problem for 20-Sim, it only looks weird.
Hey, the new Maple version also supports Matlab code, which is completely compatible
with 20-sim. So rather use that.

```maple
> C([op(dynamics)],coercetypes=false);
```

```maple
phidotdot_hip = (g * sin(phi_hip) + 2 * phi_hip *
  sin(phi_splay) * phidot_splay) / l / cos(phi_splay);
```

```maple
phidotdot_splay = -sin(phi_splay) * (l * phidot_hip *
  phidot_hip * cos(phi_splay) - g * cos(phi_hip)) / l;
```

```maple
> #Matlab([op(dynamics)],coercetypes=false);
APPENDIX C. DERIVATION OF THE EQUATIONS FOR THE VERY SIMPLE 3D WALKER

```plaintext
[ > impactformulas:= [op(fs), ldot_en, op(newdotabcd)]:
[ > C(impactformulas,coercetypes=false);

a = cos(newphi_splay) * cos(newphi_hip) * cos(phi_hip) * cos(phi_splay) + cos(newphi_splay) * sin(phi_hip) * cos(phi_splay) * sin(newphi_hip) - sin(phi_splay) * sin(newphi_splay);
b = -cos(newphi_splay) * l * sin(phi_hip) * sin(phi_splay) * phidot_splay * sin(newphi_hip) * cos(phi_splay) * cos(newphi_hip) * l * cos(phi_hip) * sin(phi_splay) * phidot_hip * cos(phi_splay) * sin(newphi_hip) - cos(newphi_splay) * cos(newphi_hip) * l * sin(phi_hip) * phidot_hip * cos(phi_splay) * sin(newphi_hip) - l * cos(phi_splay) * phidot_splay * sin(newphi_hip); c = (cos(newphi_hip) * sin(phi_hip) * cos(phi_splay) - cos(phi_hip) * cos(phi_splay) * sin(newphi_hip)) / cos(newphi_splay) / l;
d = (cos(newphi_hip) * l * cos(phi_hip) * phidot_hip * cos(phi_splay) - cos(newphi_hip) * l * sin(phi_hip) * sin(phi_splay) * phidot_splay + l * sin(phi_hip) * phidot_hip * cos(phi_splay) * sin(newphi_hip) + l * cos(phi_hip) * sin(phi_splay) * phidot_splay * sin(newphi_hip)) / cos(newphi_splay) / l;
e = (-cos(newphi_hip) * sin(newphi_hip) * cos(phi_hip) * cos(phi_splay) - sin(phi_hip) * cos(phi_splay) * sin(newphi_hip) * sin(newphi_splay) - cos(newphi_hip) * sin(newphi_splay) * l * cos(phi_hip) * sin(phi_splay) + l * cos(phi_splay) * phidot_splay * sin(newphi_hip) - cos(phi_splay) * phidot_splay) / l;
ldot = -0.10e1 / (1 + a) * b;
newl_dot = a * ldot + b;
newphidot_hip = c * ldot + d;
newphidot_splay = e * ldot + f;
[ > #Matlab(impactformulas,coercetypes=false);
```

– 66 –
Appendix D

20-sim code for the very simple walker model

This appendix lists the 20-sim submodel equations that makeup the very simple walker. First a table is given that gives the input and output port names of the submodel. The outputs represent the state $x_i^{k}$: how the state was at the start of this step.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>newphi_hip</td>
<td>phihip_begin</td>
</tr>
<tr>
<td>newphi_splay</td>
<td>phidothip_begin</td>
</tr>
<tr>
<td></td>
<td>phisplay_begin</td>
</tr>
<tr>
<td></td>
<td>phidotsplay_begin</td>
</tr>
</tbody>
</table>

variables

real phidotdot_hip , phidotdot_splay ;
real phidot_hip , phidot_splay ;
real phi_hip , phi_splay ;
real phidot_hip_temp , phidot_splay_temp ;

real newl_dot , newphidot_hip , newphidot_splay ;
boolean ev ;
real etot , eloss , ee ;
real a , b , c , d , e , f , ldot ;
real campos [3] ;

parameters

real g = 9.81 ;
real l = 1.0 ; // {m}
real m = 1.0 ; // {kg}

initial equations

Pst = [0;0;0] ;
leftlegisstanceleg = true ;
eloss = 0 ;
phihp_begin = phi_hip ;
phidothip_begin = phidot_hip_temp ;
phisplay_begin = phi_splay ;
phidotsplay_begin = phidot_splay_temp ;

equations
APPENDIX D. 20-SIM CODE FOR THE VERY SIMPLE WALKER MODEL

// Dynamic equations (directly copied from Maple)
phidotdot_hip = (g* sin ( phi_hip ) + 2*l* phidot_hip * sin ( phi_splay ))/(l* cos ( phi_splay ));
phidot_splay = -sin ( phi_splay )*(l* phidot_hip^2* cos ( phi_splay ) - g*cos (phi_hip))/l;

// Impact equations: (directly copied from Maple)
a = cos ( phi_splay )* cos ( newphi_splay )* sin ( newphi_hip )* sin ( phi_hip )* sin ( newphi_splay ) * sin ( phi_splay ) + cos ( newphi_hip )* cos ( phi_splay ) * cos ( newphi_hip )* cos ( phi_splay ) + c = (cos ( newphi_hip )* sin ( phi_hip )* cos ( phi_splay ) - cos ( phi_hip )* cos ( phi_splay ))/cos ( newphi_splay );
d = (l* cos ( phi_hip )* sin ( phi_splay )* phidot_splay* sin ( newphi_hip ) + l* sin ( phi_hip )* phidot_hip* cos ( phi_splay )* sin ( newphi_hip ) - cos ( newphi_hip )* l* sin ( phi_splay )* phidot_hip* cos ( phi_splay ))/cos ( newphi_hip )* l; 
e = (-cos ( newphi_splay )* sin ( phi_splay ) - cos ( phi_splay )* sin ( newphi_hip )* sin ( newphi_splay )* sin ( phi_hip ) - cos ( phi_hip )* cos ( newphi_hip )* cos ( phi_splay )* sin ( newphi_splay ));
f = (-cos ( phi_splay )* cos ( newphi_splay )* l* phidot_splay + sin ( newphi_hip )* sin ( newphi_splay )* l* sin ( phi_hip )* sin ( phi_splay ));

ldot = -0.1 * e1 / (1 + a) * b;
newl_dot = a * ldot + b;
newphidot_hip = c * ldot + d;
newphidot_splay = e * ldot + f;

// Hip and swing foot position
Phip = Pst + l*
       [sin (phi_hip)*cos (phi_splay)];
       (if (leftlegisstanceleg) then -1 else 1 end) * sin (phi_splay);
       cos (phi_hip)*cos (phi_splay)];

Psw = Phip - l*
       [sin (newphi_hip)*cos (newphi_splay)];
       (if (leftlegisstanceleg) then 1 else -1 end) * sin (newphi_splay);
       cos (newphi_hip)*cos (newphi_splay)];

// Impact detection:
ev = eventdown ( Psw [3] ) ;
Pst_temp = 1 * Pst; // Needed for animation

if ( ev ) then
       // Animation variables
       leftlegisstanceleg = not leftlegisstanceleg;
Pst=Psw;

// x+ - hold (stores the state just after impact (x+) for
// use by a controller)
phiihip_begin = -newphi_hip;
phidot_hip_begin = newphidot_hip;
phisplay_begin = newphi_splay;
phidotsplay_begin = newphidot_splay;

// Plottable variables
eloss = /*eloss + */0.5 * m * newl_dot^2;

end;

// Resettable integrator
phidot_hip_temp = resint(phidotdot_hip,newphidot_hip,ev);
phidot_splay_temp = resint(phidotdot_splay,newphidot_splay,ev);

phidot_hip = 1*phidot_hip_temp; // Workaround for a 'bug' in 20-Sim
phidot_splay = 1*phidot_splay_temp; // Workaround for a 'bug' in 20-Sim

phi_hip = resint(phidot_hip,newphi_hip,ev);
phi_splay = resint(phidot_splay,newphi_splay,ev);

// Animation variables
Psw_show = resint( 40*(Psw-Psw_show), Pst_temp, [ev;ev;ev] );
campos = int( 2 * (Phip-campos));

// Plottable variables: potential plus kinetic energy
etot = 0.5*m*l^2*(phidot_splay^2+phidot_hip^2*cos(phi_splay)^2)+m*g*l*cos>(
  phi_hip)*cos(phi_splay);
ene = etot+eloss;
Appendix E
Matlab code for the very simple walker model

These are Matlab files belonging to the very simple 3D walker model.

E.1 stos.m

function [x_next,t, fallen] = stos(x_current,u,noerroronfall)
  \% function [x_next, t, fallen] = stos(x_current,u,[noerroronfall])
  \%
  \% stos: step-to-step formula.
  \% This function gives the new state at the beginning of a step, given the
  \% initial conditions x_current and the inputs u. The integration is
  \% stopped when the swing foot touches the ground. The swing foot has
  \% angles phi_next.
  \%
  \% x_current, an x_next are:
  \%
  \%  / phi_hip_stance \   
  \% / phidot_hip_stance /   
  \% x_k = / phi_splay_stance /   
  \% \ phidot_splay_stance /
  \%
  \%  / phi_hip_swing \   
  \% u = \ phi_splay_swing /   
  \%
  \%
  \% Returns:
  \% x_next: The state at the beginning of the next step
  \% t: the time this step took.
  \% fallen: 0 when the robot has not fallen this step, 1 when it did fall
  \%
  \% Note that:
  \% - phi_hip_stance is negative at the beginning of the step and positive
  \% at the end of the step (and is negated again after the step was done
  \% so that the x represents the new values for the beginning of the next
  \% step
  \% - phi HIP swing is always negative (as the leg is always to the front).
  \% - phi_splay_x are both always positive; it represents the angle of the
  \% leg, where 'outward' is positive.
  \%
  \% Don't forget to initialize the walker parameters, e.g.:
% global walker_l walker_g walker_m;
% walker_l = 1; % [m]
% walker_g = 9.81; % [m/s2]
% walker_m = 1; % [kg]
%
% If as a third parameter a non-zero value is given, the integration will
% not give a terminating error when the robot falls, but it will set
% output variable fallen to 1.

global walker_l walker_g walker_m;
global stos_noerroronfall;
t_max = 5; % [s] Each step lasts only this long; if this is exceeded,
% the robot is assumed to fall

if (exist('noerroronfall') & noerroronfall ~=0)
stos_noerroronfall = 1;
else
% so: if nothing was given, do give an error if the robot falls
stos_noerroronfall = 0;
end;

x_current = colvec(x_current);
u = colvec(u);

if (length(x_current)~=4)
clear stos_noerroronfall;
error ('wrong x_current given (see help stos)');
end;

if (length(u)~=2)
clear stos_noerroronfall;
error ('wrong u given (see help stos)');
end;

% check if the walker has a pose that exists and that is okay according
% to our walking criteria:
if (x_current(1)<-pi/2 | x_current(1)>0 | x_current(3)<-pi/2 | x_current>(3)>pi/2 |x_current(2)<0)
if (stos_noerroronfall)
disp ('stos: wrong x_init:'); disp (x_current);
x_next = [0;0;0;0];
t = t_max;
fallen = 1;
return;
else
x_current',
error ('stos: wrong x_init');
end;
end;

l = walker_l;

options = odeset ('Events',@ev,'RelTol',1e-9,'AbsTol',1e-9);

% Do the integration
[tt,yy] = ode45(@intg, [0,t_max], [x_current; u], options);

if (stos_noerroronfall==2)
fallen = 1;
else
    fallen = 0;
end;

if (tt(end)==t_max)
    if (stos_noerroronfall)
        disp ('t_max reached');
        x_next = [0;0;0;0];
        t = t_max;
        fallen = 1;
        return;
    else
        yy(end,:) % show this
        error ('t_max reached');
    end;
else
    yy(end,:) % show this
    error ('not over top');
end;

if (yy(end,1)<0) % then we are not over the top
    if (stos_noerroronfall)
        disp ('not over top');
        x_next = [0;0;0;0];
        t = t_max;
        fallen = 1;
        return;
    else
        yy(end,:) % show this
        error ('not over top');
    end;
else
    clear stos_noerroronfall;
    yy = yy(end,:);
    % if we’re here, the step was okay, so do the impact equations
    % (copied directly from 20Sim).
    phi_hip = yy(1);
    phidot_hip = yy(2);
    phi_splay = yy(3);
    phidot_splay = yy(4);
    newphi_hip = yy(5);
    newphi_splay = yy(6);

    a = cos(newphi_splay) * sin(phi_hip) * cos(phi_splay) * sin(newphi_hip) +>
        cos(newphi_splay) * cos(newphi_hip) * cos(phi_hip) * cos(phi_splay) >>
        - sin(phi_splay) * sin(newphi_splay);
    b = -cos(newphi_splay) * l * sin(phi_hip) * sin(phi_splay) * phidot_splay>>
        * sin(newphi_splay) - cos(newphi_splay) * cos(newphi_hip) * l * sin(>>
        phi_hip) * phidot_hip * cos(phi_splay) - cos(newphi_splay) * cos(>>
        newphi_hip) * l * cos(phi_hip) * sin(phi_splay) * phidot_splay + cos(>>
        newphi_splay) * l * cos(phi_hip) * phidot_hip * cos(phi_splay) * sin(>>
        newphi_hip) - l * cos(phi_splay) * phidot_splay + sin(newphi_splay);
    c = (cos(newphi_hip) * sin(phi_hip) * cos(phi_splay) - cos(phi_hip) * cos(>>
        phi_splay) * sin(newphi_hip)) / cos(newphi_splay) / l;
    d = (cos(newphi_hip) * l * cos(phi_hip) * phidot_hip * cos(phi_splay) - >>
        cos(newphi_hip) * l * sin(phi_hip) * sin(phi_splay) * phidot_splay * >>
        l * sin(phi_hip) * phidot_hip * cos(phi_splay) * sin(newphi_hip) + l >>
        * cos(phi_hip) * sin(phi_splay) * phidot_splay * sin(newphi_hip)) / >>
APPENDIX E. MATLAB CODE FOR THE VERY SIMPLE WALKER MODEL

```matlab
\[
\cos(\text{newphi}_splay) / l;
\]
\[
e = -(\sin(\phi_hip) * \cos(\phi_splay) * \sin(\text{newphi}_splay) * \sin(\text{newphi}_hip) + \cos(\text{newphi}_hip) * \sin(\text{newphi}_splay) * \cos(\phi_hip) * \cos(\phi_splay)) / l;
\]
\[
f = -(l * \sin(\phi_hip) * \sin(\phi_splay) * \phi_dot_splay * \sin(\text{newphi}_splay) * \sin(\phi_hip) * \sin(\phi_splay) + \cos(\text{newphi}_hip) * \sin(\text{newphi}_splay) * \cos(\phi_hip) * \sin(\text{newphi}_splay) + \cos(\text{newphi}_splay) * \sin(\phi_splay)) / l;
\]
\[
\text{ldot} = -b / (1 + a);
\]
\[
\text{newl}_dot = a * \text{ldot} + b;
\]
\[
\text{newphidot}_hip = c * \text{ldot} + d;
\]
\[
\text{newphidot}_splay = e * \text{ldot} + f;
\]

% Now we have enough info to build the new vector:
%(we have to do some negating for leg swap)
\[
x_{next} = [\text{newphi}_hip, \text{newphidot}_hip, \text{newphi}_splay, \text{newphidot}_splay]';
\]
\[
t = \text{tt}(\text{end});
\]

% calculation of integral
function dy = intg(t,y)
\[
g = \text{walker}_g;
\]
\[
l = \text{walker}_l;
\]
\[
\phi_hip = y(1);
\]
\[
\phi_dotted_hip = y(2);
\]
\[
\phi_splay = y(3);
\]
\[
\phi_dotted_splay = y(4);
\]
\[
\phi_dotteddot_hip = (g * \sin(\phi_hip) + 2 * l * \phi_dotted_hip * \sin(\phi_splay)) / l / \cos(\phi_splay);
\]
\[
\phi_dotteddot_splay = -\sin(\phi_splay) * (l * \phi_dotted_hip * \phi_dotteddot_hip) / \cos(\phi_hip)); / l;
\]
\[
dy = [\phi_dotted_hip; \phi_dotteddot_hip; \phi_dotted_splay; \phi_dotteddot_splay; 0; 0];
\]

% O's: our setpoints for the swing leg don't change
endfunction

%---------------------------------
% Events function; we should terminate if the next foot hits the ground.
% For this we have the test if the hip is above the leg height of the new
% foot.
function [value,isterminal,direction] = ev(t,y)
\[
g = \text{walker}_g;
\]
\[
l = \text{walker}_l;
\]
\[
\phi_hip = y(1);
\]
\[
\phi_dotted_hip = y(2);
\]
\[
\phi_splay = y(3);
\]
\[
\phi_dotted_splay = y(4);
\]
\[
\phi_dotteddot_hip = (g * \sin(\phi_hip) + 2 * l * \phi_dotted_hip * \sin(\phi_splay)) / l / \cos(\phi_splay);
\]
\[
\phi_dotteddot_splay = -\sin(\phi_splay) * (l * \phi_dotted_hip * \phi_dotteddot_hip) / \cos(\phi_hip)); / l;
\]
\[
value = \text{swingfoot}_height * \phi_hip + \phi_dotted_splay * y(2);
\]
\[
\text{isterminal} = 1;
\]
\[
direction = -1; \% downward
\]
```
% we also want to test if the robot is still walking forward (instead of falling backward). As soon as the robot starts to fall backward, we terminate the integration.

if (y(2) <= 0)
    if (stos_noerroronfall ~= 0)
        stos_noerroronfall = 2; % indicate that the robot fell
        disp('stos: Robot falling backward');
    else
        t, y, % show these
        clear stos_noerroronfall;
        error ('stos: Error: robot is falling backward.');
    end;
end;

if (hipheight <= 0)
    if (stos_noerroronfall ~= 0)
        stos_noerroronfall = 2; % indicate that the robot fell
        disp('stos: Hip under ground');
    else
        t, y, % show these
        clear stos_noerroronfall;
        error ('stos: Error: hip under ground.');
    end;
end;

% endfunction

E.2 findlimitcycle.m

function [x_limitcycle, u_limitcycle, Jx, Ju] = ...
    findlimitcycle (phi_0, phidot_hip_top)
% function [x_limitcycle, u_limitcycle, Jx, Ju] = 
%    findlimitcycle (phi_0, phidot_hip_top)
%
% This function finds a limit cycle for the given parameters. It also returns the jacobian Jx = dSTOS/dx, Ju = dSTOS/du around this limit cycle.
%
% phi_0 = [phi_hip_0, phi_splay_0]'
% phi_hip_0 should be negative; phi_splay_0 should be positive.
% phidot_hip_top should be positive; it is the desired velocity
% phidot_hip at phi_hip=0.
% Before usage, set the globals walker_l, walker_g and walker_m to the desired values.

x0 = [3, 0]'; % very rough guess; phidot_hip large, so that we will do a step anyway (we won’t fall back)
o = optimset('tolfun',1e-9, 'display','off');

x = fsolve(@optimizefunction, x0, o, phi_0, phidot_hip_top);

x_limitcycle = [phi_0(1), x(1), phi_0(2), x(2)]';
u_limitcycle = [phi_0(1), phi_0(2)];
APPENDIX E. MATLAB CODE FOR THE VERY SIMPLE WALKER MODEL

J = numjac(@numjac_stos,[],[x_limitcycle; u_limitcycle],x_limitcycle>>
,1000*[1:1:1:1:1:1],[],0);
Jx = J(:,1:4); Ju = J(:,5:6);

function y = optimizefunction(x, phi_0, phidot_hip_desired_top)
% function y = optimizefunction(x, phi_0, phidot_hip_desired_top)
% % Use this function with fsolve, to find an
% % x = [phidot_hip_nul, phidot_splay_nul],
% % that gives a limit cycle.
% % Conditions are:
% % - phidot_splay_end = phidot_splay_nul
% % - phidot_hip(phi_hip=0) is approximately equal to
% % phidot_hip_desired_top (only a fraction of a percent off)
% % Usage:
% % x = fsolve(@optimfunc,x0, optimset, phi_0, phidot_hip_desired_top);
global walker_l walker_g walker_m;
calculate what the energy of the system would be if the phidot_hip is
% equal to the desired phidot_hip. We do an approximation here:
% cos(phi_splay)=1 at phi_hip=0
E_top = 1/2 * walker_m * walker_l^2 * phidot_hip_desired_top^2 + walker_m>
* walker_g * walker_l;
xx = [phi_0(1), x(1), phi_0(2), x(2)]';
uu = [phi_0(1), phi_0(2)];
yy = stos(xx, uu);
% Calculate the E at the end of the step. As the total energy of the
% system does not change, this energy will be equal to the E_top, if the
% phidot_hip at phi_hip=0 were equal to phidot_hip_desired_top.
E = 1/2 * walker_m * walker_l^2 * (yy(2)^2 + yy(4)^2) + 
walker_m * walker_g * walker_l * (cos(yy(1)) * cos(yy(3)));
% functions that should be zero:
y = [yy(4) - xx(4); E - E_top ];

% function for numjac, because it needs a different syntax than stos
% accepts:
function x_next = numjac_stos(t,x)
x_next = stos(x(1:4),x(5:6));

E.3 findstabilizationrange.m

function [d, s, steps] = findstabilizationrange (dspace, el, xlc, ulc, K)
% % function [d, s, steps] =
% % findstabilizationrange ([d_min, d_step, d_max], el, xlc, ulc, K)
% % does a search for stability. It deviates the el’th (1..4) parameter
% % with d, and returns if the robot still walks after 300 steps (s=1) or
% % not.
% Returns two column variables:
% - d is the set of deviations
% - s = the associated stabilization
% - steps = number of steps needed to find out if it is stable (either%
%       fallen, converged or max_steps)
% Usage:
% (varies second element of zlc between zlc(2)*(1-0.5) and zlc(2)*(1+2.0)
% and uses the feedback gain K:)
% [d,s,steps] = findstabilizationrange([-0.5,0.01,2],2,zlc,ulc,K);
% d_min = dspace(1); d_step = dspace(2); d_max=dspace(3);
s = [ ];
d = [d_min:d_step:d_max];

for i = 1:length(d)
    dd = d(i);
    x = xlc; x(1) = (1+dd) * x(1);
    if (x(1)<pi/2 | x(1)>pi/2 | x(3)<-pi/2 | x(4)>pi/2)
        s(i)=1; % indicate: invalid starting position
        steps(i)=0;
        fallen=2;
        step = 0;
    else
        for step = 1:100
            u = ulc-K*(x-xlc);
            [xnew, t, fallen] = stos(x,u,1);
            if (fallen) break; end;
            if (mynorm(xnew-x)<1e-4) break; end; % then we converged
            x = xnew;
        end;
        s(i) = 1 - fallen;
        steps(i)=step;
        end;
        disp ([dd, 1-fallen, step]);
    end;
d = colvec(d);
s = colvec(s);
steps=colvec(steps);

E.4 Script: pole placement

Below a series of commands is listed with Matlab’s output. These commands were used to obtain
numerical values of eigenvalues and controller gains.

>> clear all
% In this script we do pole placing for the simple 3D walker.
% Initialize
>> global walker_l walker_g walker_m;
>> walker_l = 1; walker_g = 9.81; walker_m = 1;
% Parameters to search for a limit cycle:
>> phi_0 = [-0.3, 0.05];
>> phidot_hip_top = 1.5; % desired phidot_hip at phi_hip=0

% the findlimitcycle function looks for a limit cycle and returns
% x* and u*. It also returns the jacobians Jx and Ju
>> [xlc, ulc, Jx, Ju] = findlimitcycle(phi_0, phidot_hip_top)

xlc =
APPENDIX E. MATLAB CODE FOR THE VERY SIMPLE WALKER MODEL

-0.3000
1.7737
0.0500
-0.0617

ulc =

-0.3000
0.0500

Jx =

0 0 0 0 0
1.6529 1.0063 -0.4911 -0.0967 0
0 0 0 0 0
0.4711 0.1807 -3.9451 -1.7739 0

Ju =

1.0000 0
-1.6297 0.3716
0 1.0000
0.1935 0.5174

% eigenvectors and eigenvalues of this system (uncontrolled) are:
>> [Vx,Dx] = eig(Jx)

Vx =

0 0 0.5215 0
-0.0348 -0.9979 -0.8517 0.1141
0 0 0 0.4118
-0.9994 -0.0650 0.0517 -0.9041

Dx =

-1.7676 0 0 0
0 1.0000 0 0
0 0 0 0
0 0 0 0

% Now do pole placement. Try to get the -4.46 pole to -0.4.
% Find a feedback gain K such that dx(k+1) = (Jx - Ju * K) dx(k)
% is stable:
>> P9 = [-0.9 1 0 0]';
>> K9 = place(Jx, Ju, P9)
place: ndigits= 15
K9 =

-0.8747 -0.5408 -0.0362 -0.0847
E.4. SCRIPT: POLE PLACEMENT

\[-0.3050 \quad -0.2055 \quad -0.6216 \quad -0.3090\]

\[P4 = [-0.4 \ 1 \ 0 \ 0]';\]
\[K4 = \text{place}(Jx, Ju, P4)\]
\[\text{place: ndigits= 15}\]
\[K4 =\]
\[
-0.8533 \quad -0.5294 \quad -0.1004 \quad -0.1125
-0.1621 \quad -0.1294 \quad -1.0511 \quad -0.4950
\]

\[K0 = [0,0,0;K4(2,:)]\]
\[K0 =\]
\[
0 \quad 0 \quad 0 \quad 0
-0.1621 \quad -0.1294 \quad -1.0511 \quad -0.4950
\]

% The K9, K4 and K0 can be used for control; the stability range can be
% determined too (this is not done in this script).
% Now for the other type of control: without knowledge of the limit
% cycle. As the control is already done halfway the step, the control
% has to be implemented as part of the system dynamics.

\[\]

\[\]

\[\]

\[\]

-79-
Bibliography


