Bachelor Thesis

in Computing in Science

Development of a stable robot walking algorithm using center-of-gravity control

Entwicklung eines stabilen Roboter-Laufalgorithmus durch Schwerpunktskontrolle

Technical Aspects of Multimodal Systems
Department Informatics
University of Hamburg

submitted by
Robert Schmidt
27th February 2015

supervised by
Dr. Norman Hendrich
Eugen Richter

Abstract

Stable walking is one of the most important tasks for biped robots. Within this thesis the focus is on robot models and especially on the inverted pendulum model in combination with the capture step framework. It was presented by Marcel Missura and Sven Behnke on the RoboCup world championship 2014 [MB14]. This framework will be the basic approach for the development of a new walking algorithm. The capture step framework is designed to work with more computational power, even on low cost hardware. It's purpose is the additional stabilization of an walking algorithm. Due to missing knowledge about the existing walking algorithm, a new one will be developed. Finally the results show that this is a very complex tasks. The robots are able to walk, but the existing walking algorithm is more stable and even faster.

Kurzfassung

Ein stabiler Gang zählt zu den schwersten Aufgaben zweibeiniger Roboter. In dieser Arbeit wird das Hauptaugenmerk auf das Modell des inversen Pendels und das CaptureStep-Framwork gelegt. Dieses Framework wurde von Marcell Missura und Sven Behnke entwickelt und auf dem Symposium der RoboCup Weltmeisterschaft in Brasilien 2014 vorgestellt [MB14]. Das Framework ist auf geringe Rechenleistung und auf Fehlertoleranz bezüglich der eingehenden Messwerte ausgelegt. Dazu wurde sich am Kontext des RoboCup orientiert. Das Anwendungsgebiet dieses Frameworks ist die weitere Stabilisierung eines einfachen Laufalgorithmus, der als Basis dient. Da uns das Wissen um die genaue Struktur des bisherigen Laufalgorithmus fehlt, wurde mit der Entwicklung eines neuen begonnen. Letztendlich zeigen die Resultate, dass diese Aufgabe sich als schwerer herausstellte als anfangs angenommen. Dennoch können die Roboter laufen.

Contents

Introduction 1

1.1 Motivation

Being a member of the student RoboCup team Hamburg Bit-Bots I learned of the problems of a stable walking algorithm. This aspect becomes more important according to the goal 2050. The goal for the 2050's is to win against the current soccer world champion in a fair match. Until then, there are still many problems to solve, but most of all a stable reliable walking is required. In our first years, we used a simple open loop based walking algorithm, providing sinusoidal and cosinusoidal trajectories. These were used in a simple inverse kinematic model. The resulting walking gait was quite unstable and had no support for additional sensor data. Last year, after the World Championship in Eindhoven, we took the walking algorithm from the world champion Team DARwin. [Tea] Their framework is based on the scripting language lua, with additional bindings to C++. We were able to port their Lua code to C++ and integrate it into our software framework. Due to some modifications on our robots, we still had some stability issues with this new walking algorithm. First we tried to solve these issues by manipulating the final robots pose, calculated by this algorithm, later on, during the GermanOpen in Magdeburg 2014, we adjusted some meaningful configuration values. After that, the stability improved significantly. Currently, this algorithm is able to walk reliably stable, but the resulting walking direction differs from robot to robot. In general the given walking parameters do not fit the resulting walking direction. That's why we want to implement another more reliable and stable walking algorithm. This should be self stabilizing and ZMP based. At the symposium of World Championship in João Pessoa in Brazil 2014, Sven Behnke and Marcel Missura presented an algorithm implemented and tested on their robot platform NimbRo[Nim]. It's an algorithm about balanced walking with captured steps. It is capable of generating an omnidirectional walking gait and recovering from pushes. At the presentation of this algorithm, Marcel Missura pronounced the low computationally cost of the algorithm and the possibility to react very fast on changing stability. In contrast to the current research according to ZMP based walking, they've chosen the very simple model of an linear inverted pendulum, neglected the double support phase and even don't compute the exact ZMP to gather the necessary stability. Their framework is also able to handle relatively high sensor noise. The mentioned simplicity of their algorithm makes it an ideal algorithm for me to implement on our robots and include it into our framework. Current research in this area is coupled with much more computationally cost, more complex models and other techniques e.g. machine leaning or optimized parameters computed by neuronal networks. At least it seems like most of the algorithms were developed and tested on simulators only, so there are almost no results gathered on real robots.

1.2 Related Walking Approaches

For quite a long time the matter of walking is an interesting research area. In the field of the current walking approaches it's clear that the model complexity and the computational cost are increasing. In the case of simulated robots these results lead to meaningful walking approaches, but most of them are of high computationally cost and often untested on real robot devices. On the other hand, a test on a real robot comes with much more problems than a simulation. The robots used in the RoboCup field are mostly low cost robots based on low cost hardware. So the computational power is limited. But more important are the hardware based limits like accuracy of the motors or force, friction and other communication delays between the controller and the motor. All that can be simulated, but even ignoring these additional constrains the model complexity is already high so that these limitations often will be left out from simulation.

1.2.1 First Walking Approaches

In this case the first walking approaches are the easiest too. The robot repeats a simple generated walking pattern based on some velocity parameters. Such a pattern can be based on higher level model assumptions which then are implicitly fulfilled without causing additional computational cost. That was important for the first walking tries due to the those days state of the art computer technology.

1.2.2 Including more Sensor Feedback and Higher Models

The next evolutionary step is including the robots sensor data to gain more stability while walking. A dynamic process like walking on a real robot always has to deal with noise and other disturbances, so it's necessary to include sensor feedback into the applied model. Then the robot has a chance to react on disturbances for self stabilization. The first sensors to be included are the IMU sensors, the accelerometer and the gyroscope providing the applied forces on the robot like acceleration, rotation and gravity but not necessarily separated. In the context of these ideas there are increasing model complexities like balancing the acceleration itself or step related acceleration by moving the robots trunk according to the feet movement. [BMA04]

1.2.3 High Level Walking Models

Most recent walking approaches use ZMP based stability constrained models planning foot trajectories some steps in advance according to a pre-generated, considered as optimal, ZMP trajectory. Such an ZMP trajectory often implies high accuracy for the calculated motion trajectory. The referenced papers prove their approaches using simulators and no real robots. These robot models can consider more robot details than the previously mentioned. The model can be a relatively simple 3D inverted pendulum model [Shu01], or a 3 mass inverted pendulum model [Jag04], for the basic behaviour. Furthermore the ZMP can be calculated using the differences of all the point masses and the resulting vectors instead of approximating it or only considering the centre of mass movement.

1.2.4 Special Situation RoboCup

The walking approaches gained in the field of the RoboCup are a little bit contrasting these just presented approaches. In this area the need of multiple robots and the frequent and extensive use of the robot hardware enforce a low robot price. Otherwise one could not participate. This means many teams cannot afford the use of high precision servo control. The team NimbRo from the University of Bonn published their walking algorithm that is designed for low cost robots and can handle high sensor noise[MB13c]. This walking algorithm doesn't use high level ZMP calculations and doesn't even use direct centre of mass control but applies a simple pattern which were generated once. At generation, these pattern can have been modelled to fulfil the ZMP criterion, but running on a robot and dealing with low servo accuracy still implies instability issues while walking. The standard platform league is the most advanced RoboCup league according to the walking approaches [Fen12].

1.2.5 The Human Reference

The robotic walking algorithms are all more or less human inspired. As for now, they still don't consider running but represent a simple walking with at least one foot on the ground. Looking forward to the goal of the 2050's winning against the soccer world champion, the robots will need to learn to run. Running has some additional challenges for the walking algorithms. The step width needs to increase significantly and there will be need of "zero support phases". On the simulation site, one could start to work in this field. But due to limited servo power it will take some more years, until the robotic will come close to this goal. On the soccer playing site, the technical committee of the RoboCup still wins against the mid size league robots which drive on wheels and play the "best" soccer and show one of the best team behaviours as it was demonstrated on the RoboCup World Championship in João Pessoa in Brazil.

Basics

2

2.1 Walking Related Robot Models

It's necessary to have a model of the robot before developing an walking algorithm. Due to complexity issues it's easier to use one of the following simple models as robot representation than the whole multi joint robot-structure. This would be a joint based model which would be hard to define properly in this algorithmic context. Of course there should be any robot model to calculate for e.g. the centre of mass but not necessarily as the algorithmic part of the walker.

2.1.1 Cart-Table Model

This model can be seen as a basis to all the following models. A cart or point mass is placed on an infinite massless balanced table with exactly one support point. When the cart's centre of mass is not placed over the support point the table would topple. So the cart needs to move to balance the table again.

Figure 2.1 shows an inverted pendulum installed on a cart. This is more or less the combination of this model and the following, the inverted pendulum model.

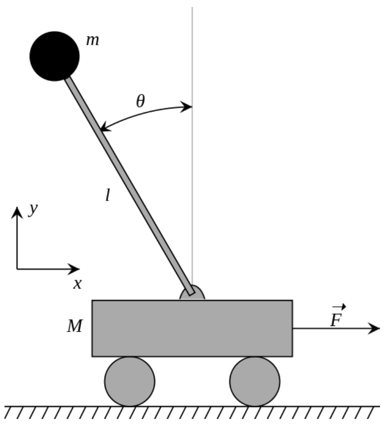


Figure 2.1: An inverted pendulum on a cart [Pen]

2.1.2 Linear Inverted Pendulum

The linear inverted pendulum model (LIPM) is the easiest of the here presented models. It's an easy extension of the cart-table model based on the differential equation $X'' = c^2X$. In this model, the table height and mass of the cart are part of the constant in the differential equation. The differential equation itself is based on the physical constrained that the table remains stable. This leads to an exponential term solving this equation. For this model the dependencies of a point's position, speed and the time to reach a specified position or speed given a state can be calculated easily using linear equations.

$$\begin{aligned} x(\Delta t) &= x_0 \cosh(C\Delta t) + \frac{\dot{x}_0}{C} \sinh(C\Delta t) \qquad (2.1)\\ &= \frac{x_0}{2}\left(e^{C\Delta t} + e^{-C\Delta t}\right) + \frac{\dot{x}_0}{2C}\left(e^{C\Delta t} - e^{-C\Delta t}\right)\\ \dot{x}(\Delta t) &= C x_0 \sinh(C\Delta t) + \dot{x}_0 \cosh(C\Delta t) \qquad (2.2)\\ &= \frac{C x_0}{2}\left(e^{C\Delta t} - e^{-C\Delta t}\right) + \frac{\dot{x}_0}{2}\left(e^{C\Delta t} + e^{-C\Delta t}\right) \end{aligned}$$

In some walking algorithms this model is used to decouple the sagittal and the lateral walking direction. In those walking algorithms the basic assumption is that the two walking directions do not interfere so that decoupling does not influence the model accuracy. Then the two axis of movement are considered separately with their own pendulum as reference.

2.1.3 3D Linear Inverted Pendulum

The 3D linear inverted pendulum model can be derived from the basic two dimensional LIPM extending the known equations with a third dimension. The basic constrains remain the same as for the LIPM with the point mass that needs to be supported by the pendulum. This model makes no assumptions about decoupling the movement into the x and y axis. This separation can be derived from the models equations. Finally, it can be solved by linear equations that were derived from a complex non linear term. [Shu01]

2.1.4 3 Mass Linear Inverted Pendulum Model

The 3 mass linear inverted pendulum model (3MLIPM) is another extension to the models mentioned above. It considers the robot as a structure of 3 linked point masses instead of a single point mass. The 3 points represent the centre of gravity of the two legs and the trunk/torso. In contrast to the other pendulum models this model shows a more complex behaviour for the calculation of the future states. This model can be translated to a single mass case by assuming the leg masses as zero. In a modelled example [Jag04] the two point masses for the legs were classified as support/stance and swing leg. The support leg is considered as relatively static link from the ground to the hip meanwhile the swing leg performs further movements to achieve the next support exchange location.

2.2 Kinematics

The implemented algorithm is based on the control loop proposed by Marcel Missura and Sven Behnke [MB14]. This algorithm provides goal positions for the centre of mass and the swing leg. To calculate the robots joint angles a kinematic framework is used which is capable of providing forward and inverse kinematics. Now I want to introduce the main principles of the forward and inverse kinematics:

2.2.1 Forward Kinematics

The forward kinematic is a transformation of a robot configuration in form of motor positions into 3D space. This transformation requires an accurate robot model. This model is a link-wise description from effector to effector. A link between two effectors can be formulated as transformation matrix. Then the calculation of forward kinematics is a multiplication of linear transformation matrices of the following types. They can be combined using matrix multiplication:

- Translation matrix: $\begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$
- Roll rotation matrix: $\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\phi) & \sin(\phi) & 0 \\ 0 & -\sin(\phi) & \cos(\phi) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
- Pitch rotation matrix: $\begin{pmatrix} \cos(\phi) & 0 & -\sin(\phi) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\phi) & 0 & \cos(\phi) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
- Yaw rotation matrix: $\begin{pmatrix} \cos(\phi) & \sin(\phi) & 0 & 0 \\ -\sin(\phi) & \cos(\phi) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

These matrices represent linear transformations. Transformation matrices are closed under multiplication.

Before we can calculate any position we need to describe the robot model:

Basically the robot model is a description of effectors and their connections which represent the robot's parts e.g. the head, the arms and the legs. These groups can be considered as kinematic chains. Before defining these chains we choose an arbitrary point best positioned between the starting points of the listed chains as root point of all the chains. Then we have to order the members of the chain so that there are no movable components between chain neighbours.

Then we start the effector wise model description: At first we define the root as start with no special transformation matrix, the identity matrix, with an arbitrary defined coordinate system, best equivalent to the global coordinate system. After that we continue with the first member of the chain to build up the basic transformation matrix: This matrix consists of a translation part from the so called parent effector, in this case the root, and a rotation part to align the local coordinate systems. Both parts are expressed in the parent's coordinate system using the transformation matrices mentioned above. Due to the fact that matrix multiplication is associative but not commutative the order of the rotations is important and has to be in a fixed order e.g. roll, pitch and yaw rotation. For each effector in the kinematic chain except the root, we align the coordinate system so, that ideally the x-axis points to the next effector. If the effector is a joint the joints real rotation axis represents the local coordinate systems y-axis. Finally the kinematic chains should have defined end points. For these endpoints we have free choice of the coordinate system, but I would align it with the global coordinate system for the robots "zero" pose.

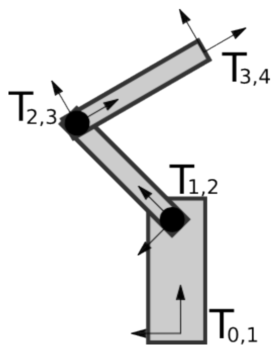


Figure 2.2: An example for an 4 joint kinematic chain.

Now we have our kinematic model and can start with the calculation given the robots joint angles. For the first example we start the calculation at the root joint. Then we proceed with the transformation matrix of the first chain member of our kinematic chain and apply an additional pitch rotation with the joints angle on the right. Then we proceed with the next joint and it's transformation and rotation matrix until we reach the joint of interest. When our start point is not just the root, respectively start and end belong to different chains we have to use inverse transformation matrices and go partly in inverse direction through the kinematic tree. In this case, the joint angles need to switch their sing as long as we use inverse transformation matrices. Alternatively we can calculate both the start and end effector's position using the forward chains and invert the start effector's final position matrix. Then we apply the end effector's position matrix on the right and have the real relative position.

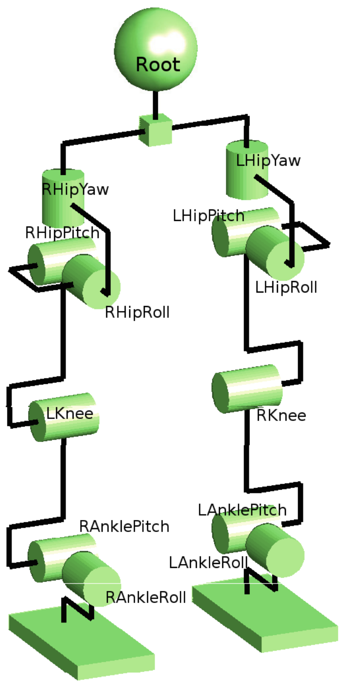


Figure 2.3: The basic chain layout for the DarwinOP with the 12 Leg Components and the Root[PBS14a]

2.2.2 Inverse Kinematics

The inverse kinematic is based on the forward kinematics. Now we have the problem in the opposite definition, given is a target's goal position and we have the robot configuration as input. The result can be either an angle differences or real joint angles which kind is ever needed by the hardware communication model.

The given task can be described as optimisation problem using $\vec{v}$ as target position and $\phi(q)$ as forward kinematic function:

$$\vec{e} = \vec{v} - \phi(q) \tag{2.3}$$

$$L(\Delta q) = ||\Delta q||^2 + ||\phi(q + \Delta q)||_C^2 \tag{2.4}$$

Given a local approximation we can formulate:

$$\phi(q + \Delta q) = \phi(q) + J\ \Delta q \tag{2.5}$$

With the jacobian J representing the local change on the end effector according to an angle manipulation at the considered joint. Reformulating and solving this equation we finally can derive the following equation:

$$\begin{aligned} \Delta q &= J^{\#} e \\ J^{\#} &= J^T (J J^T + C^{-1})^{-1} \\ C &= \lim_{\epsilon \to \infty} \epsilon \mathbb{1} \end{aligned} \tag{2.6}$$

With the pseudo inverse $J^{\#}$. In the ideal case $J^{\#}$ is the inverse of JJ^T. This matrix is a square matrix, but may be not of full rank. To ensure invertibility we add the error term C^{-1}. Now we need to derive our jacobian matrix:

$$J(q) = \frac{\delta \phi(q)}{\delta q} = \begin{pmatrix} \frac{\delta\phi_1(q)}{\delta q_1} & \frac{\delta\phi_1(q)}{\delta q_2} & \cdots & \frac{\delta\phi_1(q)}{\delta q_n} \\ \frac{\delta\phi_2(q)}{\delta q_1} & \frac{\delta\phi_2(q)}{\delta q_2} & \cdots & \frac{\delta\phi_2(q)}{\delta q_n} \\ \vdots & & & \vdots \\ \frac{\delta\phi_m(q)}{\delta q_1} & \frac{\delta\phi_m(q)}{\delta q_2} & \cdots & \frac{\delta\phi_m(q)}{\delta q_n} \end{pmatrix} \tag{2.7}$$

In the described case using the model presented above the jacobian matrix is filled with local derivatives of the joints angle difference effect. This positional manipulation can be computed locally using the vector cross product between the y-axis and the difference vector to the manipulated target effector in the considered joint's coordinate system. To insert this into the jacobian the vector needs to be transformed into the start joints coordinate system. This transformation matrix is the inverse matrix of the rotational part of the joint's position matrix according to the actual traversed kinematic tree. This calculation is that easy due to the kinematic modelling.

For a given task the jacobian is most often not a square matrix, so it's not invertible. This is why the pseudo inverse matrix $J^{\#}$ is used in the formulas above. This matrix can represent a under-determined system of equations, so we add an error term C^{-1} to ensure an almost valid inverse matrix (see equation (2.6)).

The inverse kinematic is not restricted to positional manipulation. It can be applied on every other kind of effector manipulation. In these extra cases the jacobian matrix needs to be created accordingly. E.g. to manipulate an effector's orientation given another effector's orientation we can use other columns of the used transformation and position matrices mentioned in the example above, where the positional difference is the fourth column of the position matrix.[PBS14b]

2.3 Extending the Inverse Kinematics

Basic tasks for the inverse kinematics are position manipulation and with small adjustments even orientation based manipulations. Manipulation of the centre of mass can be calculated in a similar way, there is only need to express the joint's effect of the positional change on the "centre of mass" movement. The next interesting step is combining more than one target into a batch task. The easiest way to do this is extending the jacobian. Thereby one should consider that all effect vectors lay in a comparable range. Otherwise a given task type is preferred by maths. In our case, the angle adjustment tasks are ranged from -1 to 1, so the position manipulation tasks should fit to this too. Therefore the robot's sizes and offsets should be measured in meter. For the Darwin this may be not the optimal measurement because of it's assumed maximum height of 50 cm.

This method is sufficient to solve tasks of equal priority but what if there is a hierarchy for the tasks? Then we need to formulate some preconditions so that the lower valued tasks can be performed without interfering in the results of the higher prioritised tasks.

Lutz Freitag from the FUmanoid RoboCup team presented his kinematic approach from forward and inverse kinematics up to batch tasks and lower prioritized subspace tasks. The part I want to explain is his extension of the kinematic from the batch tasks up to the subspace tasks, a feature our kinematic framework is not capable of.

I want to start this with the basic jacobian matrix J. In the simplest case J has three rows and as many columns as effectors are involved into the task. So J is usually not a square matrix and not invertible. But we can compute an pseudo inverse matrix $J^{\#}$, which can be an inverse in the euclidian sense for case the rank of the Jacobian is three. Otherwise we have to add an error term to guarantee invertibility.

$$J^{\#} = (J^T J + \epsilon \mathbb{1}_n)^{-1} J^T$$

In the ideal case $J^{\#} J = \mathbb{1}_n$ the identity matrix. But in most cases, the Jacobian is not of full rank, so this is not an identity matrix. So, this observation introduces a degree of freedom we can use to formulate for another task. When the Jacobian is of full rank $\mathbb{1}_n - J^{\#} J$ is an zero matrix, otherwise it defines a nullspace we can work with, without interfering in our previous results.

$$N = \mathbb{1}_n - J^{\#} J$$

Now I want to consider two hierarchical tasks having J_1 and J_2 of dimension $m \times n$ and the error vectors e_1 and e_2:

$$\begin{aligned}
J_1^{\#} &= J_1^T (J_1 J_1^T + \epsilon \mathbb{1}_m)^{-1} \\
\Delta q_1 &= (J_1^T J_1 + \epsilon \mathbb{1}_n)^{-1} J_1^T e_1 \\
\Delta q_1 &= J_1^{\#} e_1 \\
N_1 &= \mathbb{1}_n - (J_1^T J_1 + \epsilon \mathbb{1}_n)^{-1} \\
\Delta \hat{q_2} &= N_1 (J_2^T J_2 + \epsilon \mathbb{1}_n)^{-1} J_2^T e_2 \\
\Delta q &= \Delta q_1 + \Delta \hat{q_2}
\end{aligned}$$

Now we want to derive the general formula:

$$\begin{aligned}
J_i^{\#} &= J_i^T (J_i J_i^T + \epsilon \mathbb{1}_m)^{-1} \\
N_i &= N_{i-1} - N_{i-1} J_i^{\#} J_i \\
\Delta q_i &= N_{i-1} J_i^{\#} e_i \\
\Delta q &= \sum_i q_i \\
N_0 &= \mathbb{1}_n
\end{aligned}$$

This way it's possible to order kinematic batch tasks hierarchically and solve them in the given order. The order is guaranteed and no further special cases need to be considered.

2.4 Stability Constraints

Walking is a dynamic process that has to be stabilized. There are many approaches that fulfil this goal. They all are based more or less on the same few constraints.

2.4.1 Centre of Mass

The first and easiest idea to keep a robot stable is assuring that it's centre of mass is always supported by the feet's support polygon. The robot's feet touching the ground define a polygon having at least 4 edge points. Connecting them to an convex structure gives the support polygon. To check this criterion we have to calculate the robot's centre of mass for the x and y axis, but not for the z-axis which is independent for this approach. In the end this is a static stability because we ignore any dynamic components to achieve this kind of robot stabilization. Considering these components too would lead to the second, more general, stability approach, the ZMP approach which will be explained afterwards. The stability gained using this approach is not suited for fast dynamic walking because the restriction of keeping the centre of gravity supported forces the robot to walk with a short step width. Furthermore walking with many fast very short steps is not human like. Usually human increase the step width at higher velocity. Due to limitations of the leg movement abilities and the motors, this kind of walking was sufficient as a start but now should be replaced by less restrictive approach according to the step width or achievable speed. For every component of the robot we have to know it's mass m_c and relative position T_c, then we can calculate an offset vector, or the resulting force vector according to the given root position. $F_c = T_c \cdot m_c$ Summing up all these partial forces leads to the relative mass vector and centre of gravity F_{com}.

$$F_{com} = \frac{\sum F_c}{\sum m_c}$$

We can always change the view of the centre of mass by multiplying a transformation matrix of another robot's component. For e.g. we want to know the position of the centre of gravity relative to the right foot, then we multiply the transformation matrix from the right foot to the given root component of the centre of mass calculation with our given centre of gravity offset vector. Now we have the relative position for another component.

2.4.2 Zero Moment Point

As mentioned above, the zero moment point is based on the ideas of the centre of mass approach but is less restrictive in some points limiting some basic walking gait properties. In a static non moving case, this approach is reduced to the centre of mass approach. The advantages of the zero moment point come into play, when the desired step width is too high to keep the centre of mass supported for all the time during walking. The zero moment point is defined as the point with no inertia when summing up all forces applied in this model. The ZMP model considers the gravitation and acceleration related forces. In the static case, the acceleration forces are zero, but they are essential for the dynamic walking process. The model constraints of the ZMP is that the point calculated as sum of the gravitation and the acceleration forces, always remains inside the support polygon. In mathematical terms:

$$T_{ZMP} = T_{com} + T''_{com}$$

This has the following consequences for the walking pattern: either the trajectory is calculated computationally expensive some steps in future, or the the model complexity needs to be reduced. Theoretically we can always compute a future centre of mass state from the current motion and inertia state. In contrast to the centre of mass approach, it's not always possible to stop at any time, because now the centre of mass can be unsupported during walking, or especially when it's supported stopping would result in falling, because of remaining velocity and inertia. The ability to stop at any time for the centre of mass approach results in the simpler model and lower achievable walking speeds. [BS08]

2.4.3 The Inverted Pendulum

The model of the inverted pendulum implicitly follows the constrains of the ZMP approach. This model combines a designed position and the speed for every time according to a given starting position so that the centre of mass is moving on a line, so that the resulting ZMP stays at the zero position. [hNHI02]

The Capture Step Model

3

3.1 The Implemented Algorithm

This is the description of the capture step framework presented by Marcel Missura and Sven Behnke [MB14].

The capture step framework implemented within this bachelor thesis is based on the following model. The robot is represented as two uncoupled linear inverted pendulums. The pendulum state can be described using the following equations:

$$x(\Delta t) = x_0 \cosh(C\Delta t) + \frac{\dot{x}_0}{C} \sinh(C\Delta t) \tag{3.1}$$

$$\dot{x}(\Delta t) = Cx_0 \sinh(C\Delta t) + \dot{x}_0 \cosh(C\Delta t) \tag{3.2}$$

$$t(x) = \frac{1}{C} \ln\left(\frac{x}{c_1} \pm \sqrt{\frac{x^2}{c_1^2} - \frac{c_2}{c_1}}\right) \tag{3.3}$$

$$t(\dot{x}) = \frac{1}{C} \ln\left(\frac{\dot{x}}{Cc_1} \pm \sqrt{\frac{\dot{x}^2}{c_1^2} + \frac{c_2}{c_1}}\right) \tag{3.4}$$

$$c_1 = x_0 + \frac{\dot{x}_0}{C} \tag{3.5}$$

$$c_2 = x_0 - \frac{\dot{x}_0}{C} \tag{3.6}$$

$$C = \sqrt{\frac{g}{h_{CoM}}} \tag{3.7}$$

Using the equations (3.1) - (3.4) the robot can be modelled as a combination of two independent linear inverted pendulums. The sagittal pendulum represents the forward movement and the lateral pendulum represents the stable stepping and the sidewards movement. Both inverted pendulums are used to describe the robots state and calculate the desired future states. When starting with a new step, the desired support exchange location s is calculated. The calculation of this state s requires four walking inherent parameters α, δ, ω and σ.

3.1.1 The Default Walking Parameters

The trajectory generation of the framework is mainly influenced by the four parameters α, δ, ω and σ. These four parameters represent meaningful restrictions to the generated trajectory. The meaning can be observed regarding the explanations and figure 3.1. Now I want to describe them in the given order:

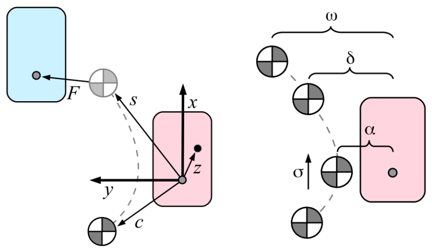


Figure 3.1: A visual description of all important parameters and values used and calculated by the capture step framework. On the left side the calculated values, on the right side the most important parameters. [MB13a]

- α the distance between the centre of the support foot and the position of the centre of mass (CoM) in the step apex. α is a security parameter for the lateral stepping cycle.
- δ represents the nominal desired support exchange location for zero lateral velocity. It's most like the foot offset to the CoM in lateral direction.

The next two parameters require a bit more calculation than these two given parameters. Therefore I want to use a simplified version of the value τ, the half step time. τ, as it will be calculated during the trajectory generation cycle, represents the time, the lateral pendulum needs to reach the desired support exchange location δ starting in the apex with zero velocity. τ is calculated using the positional pendulum equation (3.3).

$$\tau = \frac{1}{C} \ln \left(\frac{\delta}{\alpha} + \sqrt{\frac{\delta^2}{\alpha^2} - 1} \right) \tag{3.8}$$

- ω: ω is the support exchange location used when walking with maximum lateral velocity. The calculation of this value is not the easiest and for now I will use an approximation. In every stepping cycle composed of two single steps, only one step can generate lateral velocity. Each step S_1 and S_2 has its own duration $t_1 = 2\tau_1$ and $t_2 = 2\tau_2$. This leads to the following equation:

$$V_y^{max} = \frac{\omega - \delta}{t_1 + t_2} \tag{3.9}$$

$$\omega = \delta + (t_1 + t_2)V_y^{max} \tag{3.10}$$

$$\tag{3.11}$$

Let the second step generate lateral velocity. So t_2 and τ_2 are unknown, while

τ_1 can be calculated using the default approximation (3.8).

$$\tau_2 = \frac{1}{C} \ln \left(\frac{\omega}{\alpha} + \sqrt{\frac{\omega^2}{\alpha^2} - 1} \right) \tag{3.12}$$

$$\tau_2 = \frac{1}{C} \ln \left(\frac{\delta + (t_1 + t_2) V_y^{max}}{\alpha} + \sqrt{\frac{\left(\delta + (t_1 + t_2) V_y^{max}\right)^2}{\alpha^2} - 1} \right) \tag{3.13}$$

As shown, this leads to a circular dependency. For now, I consider solving this problem approximately with a simple iteration starting with $\tau_2 = \tau_1$ according to the precondition $\tau_2 > \tau_1$, because τ_1 is already representing zero lateral velocity.

- σ: σ is the maximum of sagittal speed reached in the step apex for the robot, while walking. It depends on the given maximum of sagittal velocity the robot can walk with and the nominal half step time τ. The model as an inverted pendulum makes it hard to define the velocity the CoM should have in the step apex to walk with a defined sagittal speed. Therefore σ is the parameter for an easy approximation of the needed saggital velocity in the step apex. This parameter works fine for zero lateral velocity, because any other speed increases the half step time, and also increases the real sagittal velocity with a slower stepping frequency.

$$s_x = \tau V_x^{max} = \frac{\sigma}{C} \sinh(C\tau) \tag{3.14}$$

$$\sigma = \frac{C\tau V_x^{max}}{\sinh(C\tau)} \tag{3.15}$$

3.1.2 Reference Trajectory

$$s_y = \begin{cases} \lambda\xi & if \lambda = sgn(V_y) \\ \lambda\delta & else \end{cases} \tag{3.16}$$

$$\dot{s_y} = \lambda C \sqrt{{s_y}^2 - \alpha^2} \tag{3.17}$$

$$s_x = \frac{\sigma V_x}{C V_x^{max}} \sinh(C\tau) \tag{3.18}$$

$$\dot{s_x} = \frac{\sigma V_x}{V_x^{max}} \cosh(C\tau) \tag{3.19}$$

$$\xi = \delta + \frac{|V_y|}{V_y^{max}} (\omega - \delta) \tag{3.20}$$

$$\tau = \frac{1}{C} \ln \left(\frac{\xi}{\alpha} + \sqrt{\frac{\xi^2}{\alpha^2} - 1} \right) \tag{3.21}$$

The first calculated value s_y (3.16), the desired lateral step position, is set to δ as long as no disturbing incident occurred or lateral velocity is not zero. Otherwise, a middled value between δ and ω is chosen depending on the lateral velocity. The next value $\dot{s_y}$ (3.19) is the result of using the pendulum equations (3.2) and (3.3). The calculated time from (3.3) is used as input for (3.2), starting with zero velocity in the step apex α. The sagittal positions are calculated using the pendulum equations (3.1) and (3.2), starting with at a zero position and taking the half step time τ as input.

The quantities ξ (3.20) and τ (3.21) have the following meaning: ξ is an intermediate distance for the support exchange. It is calculated from a linear interpolation between δ and ω according to the actual speed in relation to the maximum value. Note: $\xi = \delta$ when the lateral velocity is 0. τ, the so called half step time, is the calculated time (3.3) the pendulum needs to reach the position ξ starting in the apex with zero velocity.

The calculated step state is not updated during the step. It's calculated once for a single stepping cycle. To maintain stability, the balance controller updates the step state with current motion data.

3.2 Balance Control

Now I want to calculate some values to improve the robots stability. First the lateral ZMP offset is calculated. Some of the following values depend on this quantity.

$$Z_y = \frac{s_y 2Ce^{C\breve{T}} - c_y C\left(1 + e^{2C\breve{T}}\right) + \dot{c_y}\left(1 - e^{2C\breve{T}}\right)}{C\left(e^{2C\breve{T}} - 2e^{C\breve{T}} + 1\right)} \tag{3.22}$$

$$\breve{T} = 2\tau \tag{3.23}$$

$\breve{T}$ is set on support exchange and decreased afterwards during the following iterations. So $\breve{T}$ can be negative when the assumed step time is exceeded. As described we need the real remaining time until the CoM reaches the desired support exchange location. The remaining time T is calculated using the following formula:

$$T = \frac{1}{C}\ln\left(\frac{s_y - Z_y}{c_y - Z_y + \frac{\dot{c_y}}{C}} + \sqrt{\frac{(s_y - Z_y)^2}{(c_y - Z_y + \frac{\dot{c_y}}{C})^2} - \frac{c_y - Z_y - \frac{\dot{c_y}}{C}}{c_y - Z_y + \frac{\dot{c_y}}{C}}}\right) \tag{3.24}$$

Finally to determine the sagittal ZMP offset I use the capture step formula proposed by Engelsberger [Joh11].

$$Z_x = \frac{s_x + \frac{\dot{s_x}}{C} - e^{CT}(c_x + \frac{\dot{c_x}}{C})}{1 - e^{CT}} \tag{3.25}$$

After that, we are able to calculate the achievable end of step:

$$c'_x = (c_x - Z_x)\cosh(CT) + \frac{\dot{c_x}}{C}\sinh(CT) \tag{3.26}$$
$$\dot{c}'_x = (c_x - Z_x)C\sinh(CT) + \dot{c_x}\cosh(CT) \tag{3.27}$$
$$c'_y = (c_y - Z_y)\cosh(CT) + \frac{\dot{c_y}}{C}\sinh(CT) \tag{3.28}$$
$$\dot{c}'_y = (c_y - Z_y)C\sinh(CT) + \dot{c_y}\cosh(CT) \tag{3.29}$$

This produces the footstep location and the step to step location

$$F_x = \frac{\dot{c}'_x}{C}\tanh(C\tau) \tag{3.30}$$
$$F_y = \lambda\sqrt{\frac{\dot{c}'^{\,2}_y}{C^2} + \alpha^2} \tag{3.31}$$
$$S = F + (c'_x, c'_y)^T \tag{3.32}$$

This a description in my own words of the algorithm which is presented in [MB14]. The meaning of the calculated values s, c and F can also be observed in figure 3.1.

3.2.1 Walking Algorithm

The capture step walking framework as described in [MB14] controls an open loop self-stable walking algorithm. In this case it's the following: [MB13c]. The capture step framework calculates end of step positions for the assumed centre of gravity and the foot. Then these target are transformed into relative amplitudes as input parameters for the open loop compliant joint walking algorithm. This algorithm is described as a self-stable walking algorithm and follows the concepts of minimal computational costs. This walker is as simple as possible just reproducing amplitudes and joint angles according to the input parameters. All the balance control and sensor feedback is evaluated in the capture step balance control layer. The motion algorithm produces its joint angles without special robot knowledge. This needs to be implicitly included in the angle generation model. In the paper presenting this walker [MB13c] there are some parameters listed which have proved stable on the authors' robot the NimbRoOP. These parameters should make the algorithm usable on robots with a similar kinematic layout. This layout is implicitly assumed by the way of calculating the motor position from the input parameters.

Implementation and Integration 4

4.1 Programming Languages

4.1.1 Python

[Pyt] Python is a highly dynamic scripting language. it's easy to learn and can handle most of common programming paradigms like objective and functional programming. Most parts of the BitBots software are written in it. The robot's behaviour is written in Python. Furthermore many of our additional tools we use to work with the robots are written in this language. E.g. it's possible to work with the robot out of an interactive Python shell.

4.1.2 Cython

[Cyt] Within the BitBots project Cython is used in two functions. On the one hand it's the connecting language between Python and C++ and on the other hand it's optimized Python. The Cython syntax is comparable to Python. In Cython it's possible to work with explicit types of the objects, a feature I really can recommend as a C++ developer. To achieve most of the performance optimization the Cython code should be typed. In some cases one can capsule methods so that they are not accessible from Python. Therefore they have to be declared as Cython defined functions. Finally it's possible to include C and C++ libraries and even work with data structures defined in this library. In contrast to Python, Cython is a compiler language. In a first step the Cython file is compiled into a C or C++ file. Then in a second step this file is compiled into a shared object. Now a Python process can import the defined functions from this library. This is possible due to the CPython interpreter. This is an interpreter written in C and executing the Python code. This is why it's easy to add additional libraries written in C or C++.

4.1.3 C++

[C++] C++ is a powerful high performance compiler language. In the BitBots project all computationally expensive calculations are performed using C++. The parts are accessible for the Python programs using a Cython wrapper mentioned above.

4.2 The Hamburg-BitBots Software Framework

The code running on the robots is separated into a few programs. During a RoboCup soccer game there are two programs performing the typically BitBot behaviour. First there is the motion process handling the low level hardware communication. Second there is the behaviour process evaluating the game information and deciding what to do. This second process also handles the image processing. They are both written in Python. They communicate with each other using a file based inter process communication (IPC). The communication relies on continuous updating of all important data from this shared memory file. For e.g. the behaviour evaluates the ball data and decides to run to it. So it tells the motion process to run forward writing new velocity parameters into this shared file. On the other side of the IPC the motion process reads the parameter update and acts accordingly. The calculation of the walking algorithm lies within our motion process to reduce communication time overhead. This loose coupled structure has many advantages. It enables us to parametrize the motion without starting a robot behaviour. We can execute a little script which only needs to write velocity parameters into the motion process. This little script can also be an interactive Python shell.

4.2.1 Software Structure

As described above the BitBots software is Python based and only the performance critical parts are implemented in C++. Furthermore there are some Cython implemented connector classes like the image processing. In this case the Cython class is only a wrapper. This wrapper takes the Python handled camera image and some small parameters and flags. Then the image is unwrapped and passed into the C++ implementation. Afterwards some C++ data structures holding the image processing results are extracted, wrapped and converted into Python objects. The motion and walking algorithm is implemented in a different way. The motion is completely written in Cython and manages the hardware communication, walking algorithm and animation framework. The motion server holds and controls the open loop walking implementation in its own closed loop. This closed loop handles the motor communication and iteratively updates the walking state.

4.2.2 The Motion Process

The motion process is an infinite closed loop. Mainly this loop consists of the hardware communication update part, the internal parameter update and then again the write out of the new motor goal positions. Internally the motion is a state machine deciding whether an external given command can be performed or not. E.g. when the robot lies on the ground then the motion tries to stand up. Meanwhile every other action will be blocked and ignored. Another example is the penalty handling. When a robot is penalized during a game it's not allowed to move. For security reasons the motion has this flag. When it's set there won't be any movement. This flag can be set manually, too. But most time of a game the motion process's most important task is the update of the walking algorithm. The motion process runs the walking implementation in an open loop between the hardware update phase and the writeout phase. This way the walking implementation receives the current robot pose and velocity as update cycle parameters.

4.3 The Capture Step Framework Integration

The implemented Capture Step framework runs within the motion process. At startup the framework is initialized with the robot's kinematic representation. Internally the trajectory generator uses this representation to achieve a basic robot pendulum height. Later on this height is reduced using a configuration value. The pendulum height is the parameter determining the basic step behaviour for any given velocity. Using the required velocity and the pendulum height the trajector generator calculates new step targets. This generator runs exactly once after each support exchange. A step target is valid until the next support exchange. The next module of the framework is the motion observer. This observer is executed every motion cycle and determines the current motion state according to the step targets. Then there is the balance controller evaluating whether the step targets can be fulfilled or not and adjusting them if not. Finally the motion generator produces the new joint target angles using the balanced step targets. On our robots we achieve around 90 motion update cycles per second.

Figure 4.1 describes the implemented framework layout. On the right side you can see the external controller. This can be a BitBots soccer behaviour or a small script. This controller writes new velocity parameters into the shared memory file, the IPC. Then the motion process reads these new parameters from the IPC. This is the entry point for the capture step framework. The motion server handles the hardware communication and walking update. The capture step controller is updated with the current robot configuration in joint space and the required walking speeds. The internal process of the capture step framework is described in 4.3.

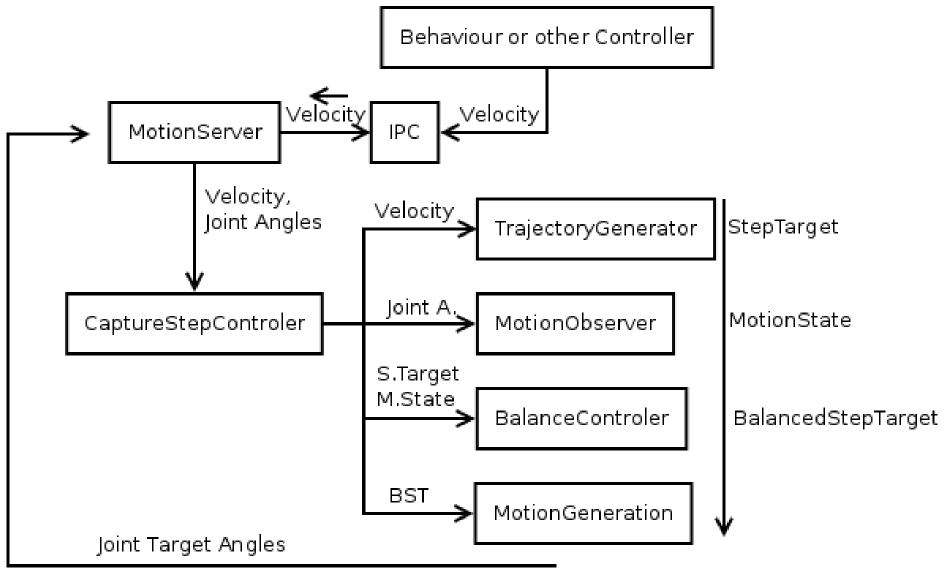


Figure 4.1: The components of the capture step framework and the binding components MotionServer, IPC and the behaviour

4.3.1 Implementation of the Capture Step Walking Framework

The implementation started with the balancing layer 3.1. This was quite easy and took a relatively short period of time due to the good described theoretical background. The main classes were the inverted pendulum, some data connector classes, the balancing controller and a simple observer for the motion state. These classes were implemented using the C++ language. Then the new classes were integrated into the existing Cython based motion implementation. According to the paper description I imagined and implemented some small test cases for the balancing layer. But then there was the main problem: How to use the stepwise generated and periodically updated target values to calculate the next robot gaol pose. On the NimbRo, the reference implementation is a low cost hand-tuned self-stable walking algorithm which is optimized for exactly this robot hardware. So I started to write an inverse kinematic based simple walking algorithm applying trajectory functions for the robot's feet. The most difficult task is the definition of these functions and the timed update.

4.4 Designing the Motion Algorithm

The existing walking implementations are working, but we have the problem, that we do not know how to parametrise them. In the beginning we used the basic standard Darwin walking algorithm, which is not especially stable, and now we use the team Darwin walking algorithm which is reliable stable, but has a constant drift to the left. So, there is no motion algorithm to work on. Due to that, I tried to implement a motion generation algorithm using inverse kinematics.

First of all, the balance controller which is described above generates target positions for the centre of mass and the foot, but these are only end of step targets. So there are many foot states and centre of mass states left to be defined. Now it's the task to find continuous functions describing the foot and the centre of mass trajectory. Therefore I want to use the inverse kinematics which was developed this year. That was inspired by Lutz Freitag from the team FUmanoids. The idea in this chapter is on the one hand describing the implementation and on the other hand showing the parameter space of the goal to develop a walking algorithm.

4.4.1 First Simple Implementation:

As described in the paper [MB14], the trajectories are based on the inverted pendulum model. According to the feet: The robots centre of mass should be moved as described by the inverted pendulum. Then the swing leg needs to reach a calculated position in time. This first version described the foot to foot distance as the twice foot to centre of mass distance. Furthermore the trunk orientation and the foot orientation were aligned with the global coordinate system. A first try on the robot showed the robots feet do not have enough grip to be stable. But finally I found out that this behaviour resulted from my overcautious choice of the walking parameter for α and δ which are described in 3.1.1.

4.4.2 Ideas for a Better Version

Inspired by watching the team NimbRo demo videos [M. a] and [M. b] I assumed the trunk to behave like an inverted pendulum and performing the pendulum swing in a fixed manner. So the second idea to work in the solution space was achieving all described end of step positions by manipulating mainly the orientation from foot to trunk and vice versa. This second version still had the fault of the overcautious choice of parameters, so the approach didn't show. Due to bugs and incomplete features, the kinematic chain design was chosen to represent every chain as a direct chain. This way it's hard to formulate a kinematic task with a trunk position and a desired orientation to the foot, which is not the identity. In this case it's necessary to calculate a combination of position and ankle angle to be applied from the root, to represent the desired position starting in the joint. But finally even this version proved to be unstable and incapable of walking.

4.4.3 Final Idea of the Algorithm

The motion generator accepts the state parameters of the capture step balancing framework. Using this as input this part defines trajectories for the legs, the stance leg and the swing leg. The input state parameters only represent end of step positions, so the motion generator also has the task of interpolating the trajectories over time. Now I want to explain the single parts of the trajectories for the swing leg and the support leg:

support: **x-axis** The inverted pendulum robot model implies a basic reference for the trajectory the support foot should perform. This is similar to the inverted pendulum itself, a combination of exponential functions. The basic behaviour of this function is the assumption of maximum speed in x direction for the support exchange and then an exponential decrease until reaching the zero position with perfect support of the trunk. Afterwards there is an exponential increase until the support exchange with maximum speed.

y-axis For the y-axis there is a similar behaviour but no passing of the apex. The centre of mass is described moving on it's exponential curve towards the zero position while slowing down until the step apex and then increasing speed while heading to the support exchange location.

These two behaviours are implemented using explicit pendulum behaviour. The z-axis is assumed to be constant.

swing: For the swing leg there is no model defined behaviour to find inspiration. At first the foot should reach it's end of step position in time and second we need to lift the swing leg because it's not the support leg. I assume both behaviours to be linear.

x-axis The x-axis is described by the linear trajectory with an additional sinusoidal smoothing as described for the support leg.

y-axis The y-axis is ignored for the interpolation which needs to be treated in a future version.

z-axis Regarding the z-axis we need a lift and a return to the ground. The final lift height is a runtime parameter and the trajectory is a little more complex trigonometric function. I wanted to have a fast lift directly after support exchange and a smooth curve before touching the ground. The lifting part should be as fast as possible and the final ground contact should be smooth too. So this trajectory is a shifted sinus in the range of $[-\frac{\pi}{2}, \frac{3}{2}\pi]$. See figure 4.2.

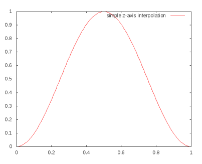


Figure 4.2

4.5 Unmentioned NimbRo Paper Problems

In theory the capture step concept looks plausible and simple, but the reality is different. For example the framework explicitly models no double support phase and neglects the dynamics according to the support exchange. Furthermore the description on basis of the inverted pendulum is hard to find within the pendulum like compliant joint walker. In this reference implementation it seems like none of these simplifications were known. The first results, which really relied on this simplified model constraints weren't satisfying and showed that reality is far more complicated. In the main referenced paper only the balancing layer is described. The real stability providing interface is the compliant walker running in an open loop parametrized by the balancing layer. This compliant joint walker is based on a model as easy as possible just periodically repeating a simple joint angle manipulation. But the description makes clear that in fact it's not that easy to provide stability. There are many parts where the simple periodic behaviour is manually adjusted, which indicates a hand-made approaches only running reliable on one robot model.

4.5.1 Model and Implementation Differences

support phases The capture step model neglects the double support phase and doesn't consider inertia and friction on support exchange. There are no explicit constraints on a trajectory that should fulfil this model. The implementing compliant joint walker introduces a small double support phase and even pushes a little into the ground to compensate energy reducing effect on support exchange which were ignored in the model.

y-axis trajectory the model is based on the inverted pendulum. This implies exponential behaviour for every function which is based on this. Especially the trajectory functions for the x- and y-axis which should match this pattern need exponential components. But the motion generating compliant joint walker is based on sinusoidal functions without any exponential behaviour.

As just described the generated and the modelled trajectory seem to differ significantly but there is no word about this is mentioned describing papers. On the NimbRo robot, the walker provides a reliable stable walking which is omnidirectional and easy parametrizable producing the expected walking directions and velocities. Having this as a basis to work on reduces the complexity of the new implemented balancing layer but doesn't guarantee to be reliable working as flexible on different robots with different walking algorithm bases.

4.5.2 Motion Generator Implementation

The motion generator controls the applied trajectories according to the given target positions. It interpolates the positions so that the trajectories are continuous functions. Basically there are 3 axes for the centre of mass trajectory and another three axes for the swing leg. The centre of mass thereby should remain in the pendulum height on the z-axis. The x- and y-axis have other non-static targets. From modelling they are constrained with the inverted pendulum behaviour. On the robot, this trajectories caused instability and other problems. Now the trajectories on these axes are inspired by a normal pendulum and described by trigonometric functions. This implementation comes close to the idea behind the compliant joints [MB13c]. On the y-axis the centre of mass should move from it's unsupported position at support exchange towards the support point This trajectory is exponential from modelling but regarding the compliant joint implementation it's solved with trigonometric functions. So this part is now implemented with a sinus function with an amplitude which is comparable to the inverted pendulum amplitude.

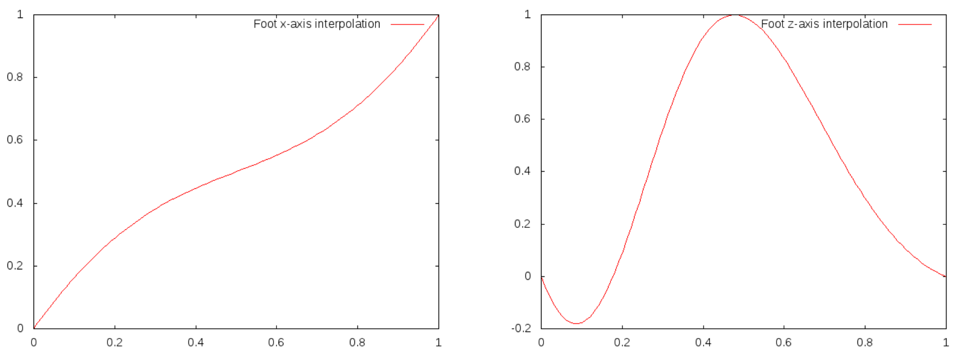
The x-axis comes up with a similar approach. To reach the final end of step position the motion generator interpolates the step distance with two functions. A linear function for some kind of base speed and a trigonometric function inspired by the pendulum behaviour.

The other part is the description of the foot trajectory. In this case the capture model provides no implicit behaviour. There are end of step positions but no model how to apply them. I want to describe the trajectories along the three axes x, y and z. First of all I want to describe the y axis the future step width. There is no requirement but the end of step location, so the trajectory along the y-axis is a linear function. The trajectory along the x-axis is comparable to the centre of mass trajectory along this axis but the end of step targets differ. Finally the foot trajectory along the z-axis, the robot height. The capture step model doesn't require a double support phase but the compliant joints implement a short double support phase. For this behaviour I tried a few interpolation functions with a trigonometric basis. The first implementations were likely to the model assumption with no double support phase but the provided walking was quite unstable. The robot always had problems to remain stable during the pendulum swing. So the idea behind the short double support phase is to support the centre of mass as long as the ZMP is outside the new support foot's support polygon. This results in a small delay of the point when lifting the swing leg from the ground.

4.6 Leaving the Exponential Functions

Many trials and breakdowns convinced me, that the exponential modelling of the trajectories is the correct way in theory, but doesn't really work on a real robot. For now on the trajectory functions are based on trigonometric functions and represent more or less a normal pendulum instead of the inverted.

Even for this decision the team NimbRo from Bonn is still the role model. But in this case not for the theoretic aspect, which is error prone as shown is the chapters above.



(a) The foot trajectory on x-axis with the highest speed at support exchange

(b) The foot trajectory on z-axis with the small push into the ground and implicit double support phase

Figure 4.3: Two foot trajectory interpolation functions

Now I will give an example of the functions describing the step target interpolations:

x-axis The trajectory along the x-axis is the sum of a linear and a cosine function. The cosine is used in a range of $[-\pi, 0]$ and of course additional scaled so that the resulting trajectory doesn't conflict with the end of step position. This function is plotted in figure 4.3a.

y-axis The trajectories according the y-axis are described by a sinus function in the range of $[0, \pi]$ with a comparable amplitude to the inverted pendulum inspired exponential function.

z-axis The foot trajectory on the z-axis is the most complex one as you can see in figure 4.3b. The main idea behind this trajectory function is a sine in range of $[0, 1.7\pi]$. To guarantee that the trajectory function is 0 at its ends, there is a linear part in it too. Finally the internal sine argument is modified so it's not linear at all. This function is plotted in figure 4.3b. This functions provides a "push" into the ground which is described as useful in [MB13c].

The function for the z-axis results from testing on a real robot, there this function behaved best. It was inspired by the previous and the description of the implicit double support phase in the compliant joint walker.

Remark

All the described functions have in common, that they are defined on $[0, 1]$ and at the interval boundaries they have either the value 0 or 1. This makes it easy to apply them as a interpolation function. Then by having a relative timer according to the step and the step target position, the function provides the current interpolation.

In conclusion I implemented every inverted pendulum behaviour with a normal pendulum. The modelled inverted pendulum defines the ranges for the implemented functions. This decision is inspired by the reference implementation of the capture step framework, the compliant joint walking algorithm. In this implementation there is no inverted pendulum behaviour, just implicitly modelled pendulums.

Problems and results

5

5.1 Design of Kinematic Tasks

The kinematic framework is a powerful tool to work with. Unfortunately it's not a direct inversion, but an iterative algorithm working with local derivatives in the joints. In most cases, the kinematic behaves as expected, but there are other examples:

1. The result may look like a random pose meaning the algorithm could not converge to a result.
2. The algorithm converges, but the result doesn't look as expected either.

To avoid these cases, the design of the task becomes more important.

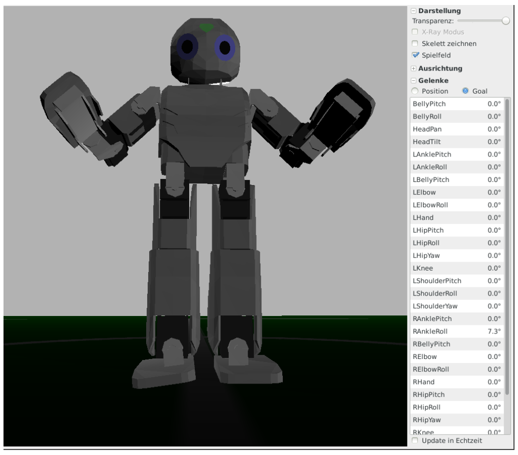
5.1.1 Possibilities to Reduce the Result Space

There are some ways to influence the calculations:

1. The chain to work with, but this one has the smallest influence, otherwise we wouldn't need that chain.
2. We could loose generality of the calculations and use our knowledge about the layout. This way we can influence the kinematics to ignore some joints.
3. But best is the formulation of a batch task, so that the kinematic framework solves multiple subtasks at once.

5.1.2 Task Components

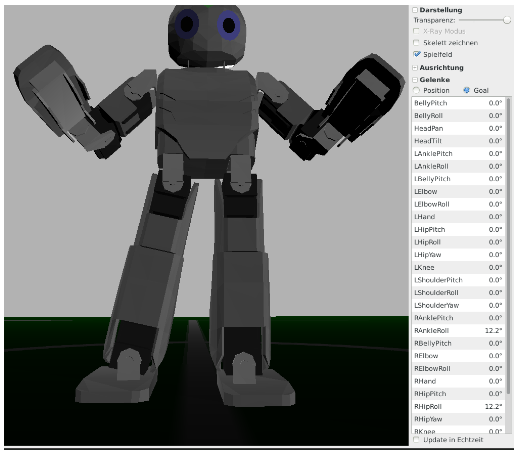
I want to give an example, where the correct kinematic behaviour looks wrong. Given is the chain, starting in the right foot going upwards through the knee and hip to the central root joint. This is the inverse chain to the normal leg chain. Using this chain, the position of the root should be moved slightly to the left, as it is needed for walking. For this task, there are two joints the robot can use: the



Skelett zeichnen [ ]; [✓ Spielfeld]; Position [✓ Goal]

BellyPitch	0.0°
BellyRoll	0.0°
HeadPan	0.0°
HeadTilt	0.0°
LAnklePitch	0.0°
LAnkleRoll	0.0°
LBellyPitch	0.0°
LElbow	0.0°
LElbowRoll	0.0°
LHand	0.0°
LHipPitch	0.0°
LHipRoll	0.0°
LHipYaw	0.0°
LKnee	0.0°
LShoulderPitch	0.0°
LShoulderRoll	0.0°
LShoulderYaw	0.0°
RAnklePitch	0.0°
RAnkleRoll	7.3°
RBellyPitch	0.0°
RElbow	0.0°
RElbowRoll	0.0°
RHand	0.0°
RHipPitch	0.0°
RHipRoll	0.0°
RHipYaw	0.0°
RKnee	0.0°

(a) A simple task definition to move the centre of mass slightly to the left from the foot's point of view. The task is incomplete defined to match the expectation



Skelett zeichnen [ ]; [✓ Spielfeld]; Position [✓ Goal]

BellyPitch	0.0°
BellyRoll	0.0°
HeadPan	0.0°
HeadTilt	0.0°
LAnklePitch	0.0°
LAnkleRoll	0.0°
LBellyPitch	0.0°
LElbow	0.0°
LElbowRoll	0.0°
LHand	0.0°
LHipPitch	0.0°
LHipRoll	0.0°
LHipYaw	0.0°
LKnee	0.0°
LShoulderPitch	0.0°
LShoulderRoll	0.0°
LShoulderYaw	0.0°
RAnklePitch	0.0°
RAnkleRoll	12.2°
RBellyPitch	0.0°
RElbow	0.0°
RElbowRoll	0.0°
RHand	0.0°
RHipPitch	0.0°
RHipRoll	12.2°
RHipYaw	0.0°
RKnee	0.0°

(b) Moving the centre slightly to the left using the properly defined task, respecting the distance and the orientation

Figure 5.1: The Robot performing the same task with a little significant adjustment, so that one looks right, but not the other one, although both are solved correct

hip roll and the ankle roll joint. If a human would try to solve this task, he would automatically use the hip to reach the distance and use the ankle to keep the right foot alignment. This is where the implicit error lies: First I will consider the simple task definition: In this case the target is just the movement. So the kinematic uses the joint with the highest effect, in this case the ankle roll joint as you can see in figure 5.1a. This results is a correct result and in the point of view of the kinematics it's the optimal result. The kinematic framework is a bit "lazy" trying to minimize the angular differences. From the human perspective this result looks wrong due to the implicit additional task of keeping the trunk orientation unchanged.

To achieve this we need to modify our target vector for the kinematic task. This needs to contain information so that the trunk orientation remains. This can be solved as a batch task with the positional target part and a rotational target describing the relative y-axis from foot to trunk as $(0, 1, 0)^T$ or the current relative y-axis. Then the kinematics produce a result matching the human expectation and human way to solve this task.

Furthermore this task could be described in an inverse way. Instead of using the inverse chain, the direct chain could be used. In this case the simple task result is more similar to the expectation. It just looks incomplete with a wrong ankle roll angle.

5.2 Problems with the Motion Generation

During the implementation I had various problems with the motion generation. Most of them belonged to the functions I used for the trajectories or robot hardware. There were less problems with the integration into the BitBots software framework.

In every calculation and assumption I've made before, I never needed to consider any forces and friction at ground contact. This changed with the first version of the walker: The robot's feet had rarely enough grip on the ground to remain stable. The reason why I had this issue is mentioned in 4.4.1. At some times I thought about computation time as another factor for my problems, but for now I didn't really notice it.

5.2.1 The Robot's Representation's Chain Layout

I wanted to use inverse kinematics to apply the calculated walking pattern. On the one hand this has the advantage that there is no need to consider the robot's kinematic layout. On the other hand it's computationally quite expensive and it's necessary to work with metrical distances. The capture step framework provides some positions like the desired end of step centre of mass location or the foot to foot distance. These positions are already in a metric measure.

In the first version I used an inverse chain from the support foot to the centre. Using this chain caused some errors explained in 5.1. The second chain was a long chain from the support foot to the swing foot. Somehow I had some problems with the chain layout, mainly with the inverse chain. Using inverse chains sometimes produces unexpected results as described in 5.1. Due to that I changed the layout to two direct kinematic chains. The problems related to this layout seemed to be easier to handle. Since then I have to invert some positions or apply additional manipulations to the target. This are modifications which were implicitly included in the more complex chain layout.

5.2.2 The First State of the Walking Pattern

In the first version the applied walking pattern kept the robot's orientation fixed and never applied any changes. Belonging to the x-axis and the y-axis the feet's orientation is the same as the robot's torso. As I will mention below, this first version was not able to provide a stable walking pattern.

Adjustments to Solve this Local Problem

As described above, I don't use the natural looking chain layout using the inverse chain, due to some implementation errors. Then I had to emulate the behaviour of

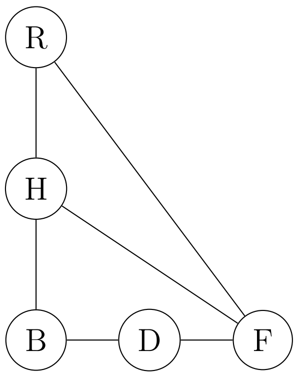


Figure 5.2: Very simple robot model representing the root, the hips, a nominal foot point and the foot position

the inverse chain using the direct one. Watching the NimbRo walking videos made me believe I could work primary using a static pendulum behaviour only using the ankle rolls to reach the target positions.

Figure 5.2 is the solution I applied. I want to manipulate the angle $\angle BFR$ in Figure 5.2, so that the on the ground distance $|\overline{BF}|$ has the desired value for the distance $|\overline{BD}|$.

$$|\overline{BF}| = \cos(\alpha)|\overline{RF}| \tag{5.1}$$
$$|\overline{BD}| = \cos(\alpha')|\overline{RF}| \tag{5.2}$$
$$\alpha' = \arccos\left(\frac{|\overline{BD}|}{|\overline{RF}|}\right) \tag{5.3}$$

5.3 Further Steps with the Robot Angular Manipulation

As expected a simple angular manipulation is not enough to produce a walking pattern. In this case the moments of inertia which were ignored from modelling come into play. The capture step model neglects moments of inertia and assumes a zero time double support phase with no loss of energy at support exchange. Using such a low dynamic implementation is cursed to fail. But reading the paper describing the compliant walker [MB13c] resulted in the idea of pushing into the ground to gain the necessary inertia.

5.3.1 Experiment with the Compliant Joint Inspired Leg Extension

As for the first 2 versions there was the problem that the code was executable, but the robot was unable to walk. The state of implementation looked like a minimal complexity walking solution which I feared to be computationally too expensive.

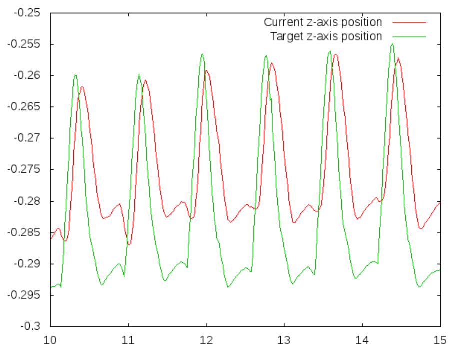
So I thought of further ideas of how to improve the generated walking pattern and maybe use even more robot model based assumptions and direct angle manipulating bypassing the kinematics. Then I tried to copy the idea of modelling a phase dependent leg extension for the support and the swing leg as described in [MB13c]. For the leg extension I manipulated the pose angles directly without using the inverse kinematics because there I already had some problems: The base situation was a quite stretched leg with an additional roll angle. Then the local derivatives to solve the leg extension problem must have been higher for the hip roll than for the combination of the three pitch joints in hip knee and ankle. In fact the robot applied a roll angle difference instead of equal manipulation for hip and ankle angle and the doubled angle manipulation for the knee. To avoid this problem I manipulated the pitch angles directly and finally applied the foot's target position starting from this improved better fitting start point. But neither the idea was good, nor the function I tried to apply.

5.3.2 Motor Delay and Friction

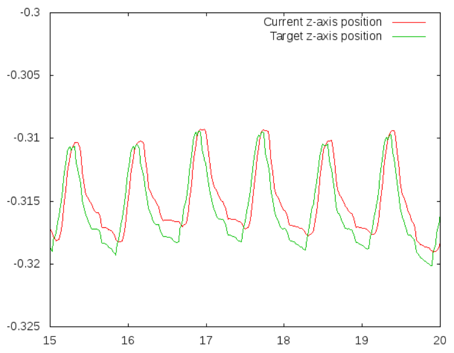
The DarwinOP is a quite cheap robot, so called low-cost hardware. The hardware is designed functionally, but according to high precision movements it's not well suited. The team NimbRo from Bonn published an interesting paper about motor signal manipulation to make the joints reach the desired position in time. There is always friction and the motor bus time delay[SB13]. Their approach is to generate a sample reference trajectory and apply it with the used motors, always comparing required and real position. Then they iteratively apply these trajectories and learn how to parametrize the motors, so that they apply the desired position in time. The motor parameters deviate from the target positions over time, but finally the motor position matches the algorithmic requirement. The older a motor becomes the stronger are the friction effects. Our DarwinOP robots have a mean age of about three years. Some motors are a bit younger, but the main problem remains. So I fear my new algorithm needs to consider this implicitly, because we don't have this low level motion signal manipulation.

Simple Hand-Made Solution

The walking trials also had problems with the applied trajectories. The trajectories on the robot and in the simulator looked different. Now to reduce the error level I implemented a small motor goal position correction. The walking algorithm runs with a quite high frequency so that any new angle for the motors should be relatively small. Using this assumption I increase the target angle and motor speed when the difference is higher than a defined threshold. This simple solution improved the resulting walking pattern in a manner, so that the robot seemed to be stronger in the knees. Before that, the walking pattern looked like the robot would be too



(a) The relative position of the foot on the z-axis in relation to the robot's root. Without using any afterwards correction



(b) The relative position of the foot on the z-axis in relation to the robot's root. Using the afterwards correction

Figure 5.3: The difference of the applied and real position with or without an afterwards pose correction

weak in the knees, unable to lift his feet. The effect of this afterwards applied correction can be seen in the following two plots figure 5.3a and figure 5.3b. The first plot shows the weakness and the second plot the improved angular behaviour on positional basis.

The "Weak" Knees

Now I want to focus the phenomenon of the weak knees while walking. I think this problem is caused by design and the hardware communication delay. The key problem of the weak knees is that the robot doesn't apply the right angles in his knees and leg pitches. The real angles the robot should apply are in a small range around zero. This is an area where it takes quite a long time for the kinematics to converge. Now I want to describe the problems: The kinematics converge faster to the right solution when the initialization is good. So I take the current robot pose as initialization for the kinematics. This is the error part by design, I could use the last result as initialization. The other problem is the hardware, because in the simulator the trajectories looked better. The effect of the weak knees always appeared in situations with almost stretched legs. Due to the performance there are only a few iterations allowed for the kinematics to converge. Now I assume that the calculated angle difference until the next step is so small, so that the motors are too lazy to perform anything under force of the robots weight. So the kinematic framework solved the same task multiple times without any progress. This can be seen as a reason for the different ranges in the figures 5.3a and figure 5.3b.

5.4 First Walking Results

Finally the robot is able to walk in some kind of configuration. The walker has some problems with stability and motion dynamics but provides a reliable movement to the front. On the real robot, there are in some terms huge differences between the needed and the applied joint positions. This fact can be observed regarding figure 5.3a.

Another plot shows the trajectory of the centre of mass according to the foot position: figure 5.4

5.5 Improved Walking Results

Within the last days I was able to improve the walking pattern running on the real robot. Now the robot performs sinus function inspired trajectories to interpolate between the generated step targets. E.g. the foot trajectory for the x- or y-axis. This time our robot Fiona was able to walk relatively stable and reliable forwards and backwards. Additional movements in the sidewards directions were possible too. Although there are still some problems with the DarWinOP robot hardware. The target trajectories still differ in some parts from the real applied motor angles. But for now it's possible to walk without falling to the side. Prior versions of the walking implementation had the problem that the robot almost fell over to the side. I was able to solve this problem by reducing the step-time and adjusting the trajectory in z-axis. Before lifting the foot, the trajectory function describes a little push into the ground. This introduces a short double support phase and helps with the dynamics. This way the centre of mass is almost supported by the support leg when the swing leg leaves the ground.

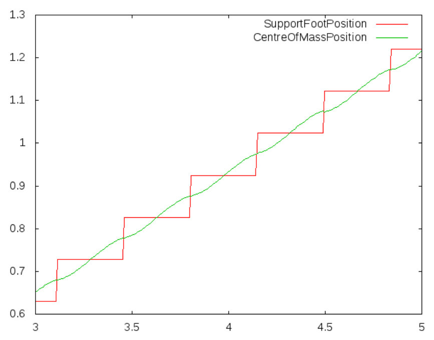
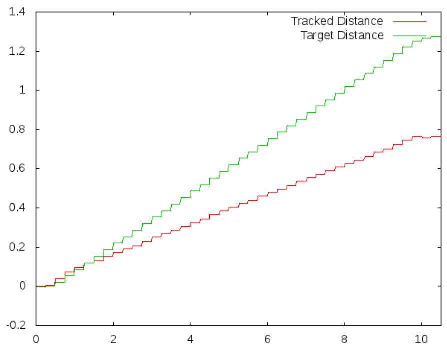


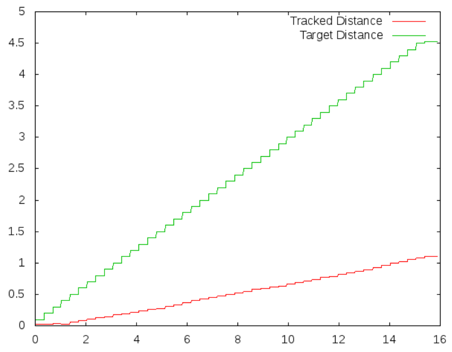
Figure 5.4: The x-axis trajectory of the centre of mass in a global coordinate system plotted in walking direction over time

5.6 Comparison of the walking Algorithm Implementations

The target of the implementation was a more reliable self-stable walking with a natural parametrization. This ambitious goal could not be reached. The existing walking algorithm provides a more stable and faster walking than the new implemented walking algorithm based on the capture step balancing layer. On a 1m simple test-run I compared the two walking implementations on the aspect of speed.



(a) The self tracked and theoretically reached stepwise distance of the 1m sprint of the existing walking algorithm



(b) The self tracked and theoretically reached stepwise distance of the 1m sprint of the new developed walking algorithm

Figure 5.5: The two walking algorithms in comparison of their self tracking while performing the 1m run. The existing algorithm is faster and more accurate according to the requirements of speed

Table 5.1 presents the walking speed performance in comparison. Thereby the existing walking algorithm runs faster, more reliable and more stable. Finally the new developed algorithm performed really unstable and unable to walk at full speed.

Algorithm	Medium Speed	Full Speed
existing	10s	5s
new	16s	unstable

Table 5.1: Times measured for the robot to walk a distance of 1 Meter

Figure 5.5 shows the self tracking results of the performed 1m run. On the left, figure 5.5a is the self tracked run of the existing walking implementation. The robots real velocity is in this case the average of the self-tracked and speed and the required speed defined by the input parameters. On the left side, figure 5.5b, there are the comparable results for the new developed capture step based walking algorithm. In this case the discrepancy between the parameters and the self-tracked result are significantly higher.

5.6.1 The Existing Walking Algorithm

This thesis had the goal to come up with a more reliable walking algorithm, than the existing one. The current walking implementation is an algorithm which is based on the Team DARwin code release of 2013 [Tea]. This code release provided an ZMP based walker. We were able to integrate this implementation into the BitBots software framework. After a long period of time we were able to tune the right parameters so that the robots could walk reliable stable. But we still don't know how to extend this algorithm. It's not documented and we rarely understand the internal behaviour. We are quite happy with the stability of this algorithm but we have problems with the parametrization. The robots often drift away when they try to optimize their position to the ball before kicking. This parameter reliability was the main goal of the new walking implementation. This existing walking implementation is stable and can handle relatively high velocities.

Conclusion 6

6.1 Conclusion

The capture step framework is a promising approach and has a simple theoretical basis. It can be implemented easily. But the core of the stability approach of the framework comes from the stability of the underlying walking implementation. Finally, this walker will be parametrized using the capture step framework results. In this case the framework's mentioned independence of this underlying walker must be analysed critically. To be able to calculate the updated end of step positions, there is some kind of motion observation included within the framework. For instance the underlying walker generates a significantly different kind of trajectory, the balancing options of the capture steps are dramatically reduced.

The reference walker implementation [MB13c] basically follows this model. So this approach can be applied in this case. The results and models show that there is much adjusting necessary to have this framework running on a NimbRo like robot. In conclusion the framework can increase the stability of a running self-stable walking algorithm. But the trajectories generated by this algorithm should be similar to the modelled inverted pendulum. The absence of such an implementation within the BitBots software library made it very difficult for me to reach some kind of approach. Now our robots can walk on this new algorithmic basis. Finally the in game performance will prove the usability.

6.2 Outlook

I could try to solve my problems with the kinematics described in section 5.3.2. To do so I could use less general approaches including assumptions about the robot layout. Furthermore I could try to use another walking algorithm as basic walker. Maybe even the existing but I think that this one is not well suited. This walking algorithm already performs active stabilization and simplified ZMP tracking. As far as we know any parameter adjustment is performed once per step. The balancing effect of the capture step framework is the online parameter adjustment of the open loop basic walking algorithm.

Bibliography

[Beh06] Sven Behnke. Online trajectory generation for omnidirectional biped walking. 2006.

[BMA04] Jacky Baltes, Sara McGrath, and John Anderson. Active balancing using gyroscopes for a small humanoid robot. 2004.

[BS08] Oussama Khatib Bruno Siciliano. *Handbook of Robotics*. Springer, 2008. Chapter 16.4.

[C++] C++ programming language. http://www.cppreference.com.

[Cyt] Cython programming language. http://www.cython.org.

[Dar] Robot: DarwinOP. http://www.robotis.com/xe/darwin_en.

[DGF13] Sebastien Krut David Galdeano, Ahmed Chemori and Philippe Fraisse. Optimal pattern generator for dynamic walking in humanoid robotics. 2013.

[EBN$^+$07] A. Erol, G. Bebis, M. Nicolescu, R. D. Boyle, and X. Twombly. Vision-based hand pose estimation: A review. *Computer Vision and Image Understanding*, 108:52–73, 2007.

[Fen12] Feng Xue, Xiaoping Chen, Jinsu Liu , and Daniele Nardi. Real time biped walking gait pattern generator for a real robot. 2012.

[hNHI02] Tomomichi Sugihara Yoshi hiko Nakamura Hirochika Inoue. Realtime humanoid motion generat ion through zmp manipulation based on inverted pendulum control. 2002.

[HRP] Robot: HRPC-4C. https://en.wikipedia.org/wiki/HRP-4C.

[Jag04] Jagannathan Kanniah, Zar Ni Lwin, D Prasanna Kumar,and Ng Nai Fatt. A'zmp' management scheme for trajectory control ofbiped robots using a three mass model. 2004.

[JHP98] Yong K. Rhee Jong H. Park. Zmp trajectory generation for reduced trunk motions of biped robots. 1998.

[Joh11] Johannes Englsberger, Christian Ott, Maximo A. Roa, Alin Albu-Schäffer, and Gerhard Hirzinger. Bipedal walking control based on capture point dynamics. 2011.

[Kem09] Kemalettin Erbatur, Özer Koca, Evrim Taşkıran, Metin Yilmaz and Utku Seven. Zmp based reference generation for biped walking robots. 2009.

[M. a] M. Missura and S. Behnke. Dynaped demonstrates lateral capture steps. http://www.ais.uni-bonn.de/movies/DynapedLateralCaptureSteps.wmv.

[M. b] M. Missura and S. Behnke. Walking with capture steps. http://www.ais.uni-bonn.de/movies/WalkingWithCaptureSteps.wmv.

[MB13a] Marcell Missura and Sven Behnke. Omnidirectional capture steps for bipedal walking. 2013. Figure 4.

[MB13b] Marcell Missura and Sven Behnke. Omnidirectional capture steps for bipedal walking. 2013.

[MB13c] Marcell Missura and Sven Behnke. Self-stable omnidirectional walking with compliant joints. 2013.

[MB14] Marcell Missura and Sven Behnke. Balanced walking with capture steps. 2014.

[Nao] Robot: Nao. http://www.aldebaran.com/en/humanoid-robot/nao-robot.

[Nim] Robot: NimbRo. http://www.nimbro.net.

[NK06] Koichi Nishiwaki and Satoshi Kagami. High frequency walking pattern generation based on preview control of zmp. 2006.

[OY13] Naoki Oda and Junichi Yoneda. Experimental evaluation of vision-based zmp detection for biped walking robot. 2013.

[PBS14a] Prof. Oussama Khatib Prof. Bruno Siciliano. *Introduction to Humanoid Robotics*. Springer, 2014. Chapter 2, figure 2.19 a.

[PBS14b] Prof. Oussama Khatib Prof. Bruno Siciliano. *Introduction to Humanoid Robotics*. Springer, 2014. Chapter 2.

[Pen] Pendulum cart. https://upload.wikimedia.org/wikipedia/commons/0/00/Cart-pendulum.svg.

[Pyt] Python programming language. http://www.python.org.

[SB13] Max Schwarz and Sven Behnke. Compliant robot behavior using servo actuator models identified by iterative learning control. 2013.

[Shu01] Shuuji Kajita, Fumio Kanehiro, Kenji Kaneko, Kazuhito Yok oiand Hirohisa Hirukawa. The 3d linear inverted pendulum mode: A simple modeling for a biped walking pattern generation. 2001.

[Tea] Team darwin code release 2013. http://www.seas.upenn.edu/~robocup/files/UPenn2013Release.zip.

[vdPvdS13] Thomas Geijtenbeek Michiel van de Panne and A. Frank van der Stappen. Flexible muscle-based locomotion for bipedal creatures. 2013.

[WSY10] Kanako Miura Wael Suleiman, Fumio Kanehiro and Eiichi Yoshida. Enhancing zero moment point-based control model: System identification approach. 2010.

Erklärung

Ich, Robert Schmidt, Matr.-Nr.: 6313640, versichere, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Ich bin mit einer Einstellung in den Bestand der Bibliothek des Departments Informatik einverstanden.

(Ort, Datum) (Unterschrift)