Humanoid Olympics 2015 - Team HOly

Laurenz Altenmüller, Simon Bilgeri, Carlo van Driesten

***Abstract*—This report presents the results and achievements of Team HOly in the TUM ICS's *Humanoid Olympics* challenge. Three teams competed in making their provided *Bioloid* robot succeed in walking and climbing stairs. We give details on how we successfully implemented both the mandatory tasks and additional special fight and rgb-d human imitation modes: Utilizing ROS, MoveIt, OpenRave's IKfast and RViz. Furthermore, we share insights on our hardware observations and modifications. On a final note, our soulution's performance is documented in the video linked in the appendix.**

I. Introduction

A. Humanoid Olympics

Humanoid Olympics is a hands-on project where five teams of three students each compete against each other in making a humanoid robot walk and climb stairs. TU München's ICS chair offers this challenge as an elective M.Sc. module. To fulfill this task, each team is equipped with Robotis's educational humanoid robot kit, Bioloid.
Humanoid locomotion is a topic of ongoing research because bipedal walking can cope with difficult terrain and is compatible to human environment and tasks like climbing stairs. As Team HOly (a portmanteau of Humanoid Olympics), we[1] tackled this challenging task. In the following we present our results and share the gained experience.

B. Project Management

It is essential for a sound team-work to divide the tasks into smaller pieces and to efficiently merge afterwards. To do so, we used several tools including Git version control, the Google Drive collaboration platform, the Lucidchart diagram editor, as well as the online Latex editor ShareLatex. To organize at least weekly team meetings, we used Google Calendar and mobile group chat. At the beginning of the project we defined optional and mandatory goals and evaluated them at the end.

C. System Overview

Figure 1 illustrates our final system configuration: Gampad and RGB-D sensor provide the user interface, which is processed by the Main node, using the ROS MoveIt framework. The controllers forward desired joint positions to the Dynamixel motors.

[1]Laurenz Altenmüller, Simon Bilgeri and Carlo van Driesten

II. Hardware

A. Bioloid

Bioloid, a humanoid robot developed by the Korean company *ROBOTIS*, combines 18 Dynamixel AX-12(+ or A) servo motors, which can be addressed uniquely by their ids. Figure 2 illustrates the different body parts as well as the 6 leg and 3 arm joints for each side. In general, these motors can be connected arbitrarily and thus many different configurations can be built. However, the resulting mechanic connections are not very tight and strong. This can be a problem for tasks like walking or climbing stairs, which demand high precision.

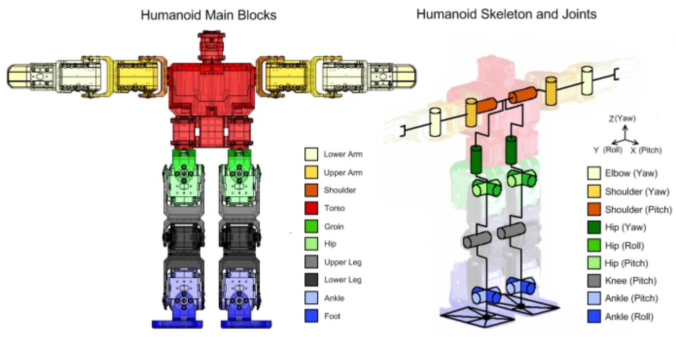


Fig. 2: Bioloid humanoid robot overview[2]

B. Modifications

1) Appearance: Like in many situations in life, a good appearance is important for a competition. Thus, we added a green chest protector, drew eyes on its head and fixed all motor cables to its body to avoid any damage.

2) Hip: In an effort to fix aforementioned non-compliant Bioloid mechanics, we decided to try to modify the robot's construction in such a way that the hip, where the bending was a big problem, would be stabilized through reinforcement. Analysis of the problem showed that the leg is attached to the torso only at the tip of the axis of the motor that rotates on the proximal-distal-axis (see figure 3). We reinforced the structure by enclosing the motor's axis with a spare Bioloid brace part. As the leg now was firmly screwed to the brace and the brace attached to the rotating motor on both of its sides, the modification was able to completely eliminate any bending previously caused by the weak servo horn structure.

[2]http://humanoids.dem.ist.utl.pt/humanoid/overview.html

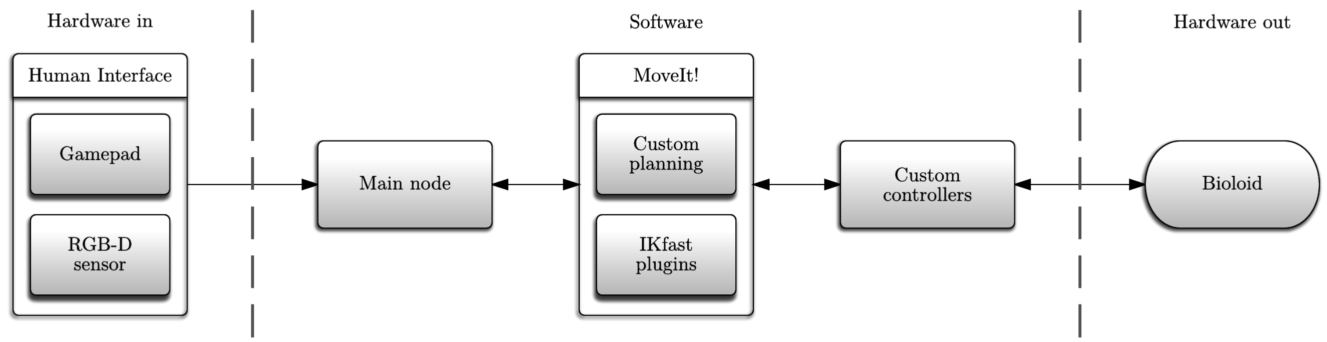


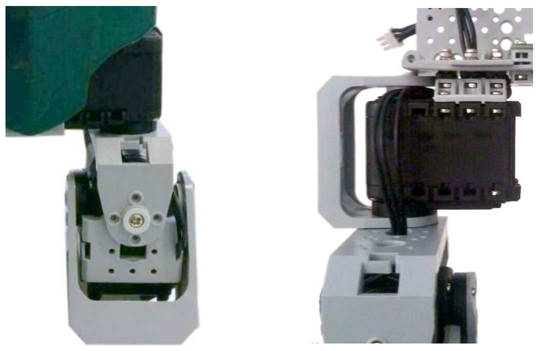
Fig. 1: The HOly system architecture

However, the downside of this modification was that it increased our robots hip width by a considerable amount of about two centimeters, because the reinforcing braces must not rub on the main torso. The fact that the legs now were further apart meant that the robot had to lean over much more to completely shift its weight over to one foot. This made a walking gait much more difficult and also odd looking. In the end we reverted the changes, because the drawback of more complicated walking outweighed the gained stability.

3) Sensors: Sensors are of great importance in robotic systems because they provide feedback to the robot's controller which helps to correctly react and adapt to changes in the robots environment (rough or difficult terrain) or prevent or detect various failures (robot is about to fall). The Bioloid does have some sensors that would be helpful in these regards. For example, the 2-axis gyrometer could be used to determine the robots actual rotation with respect to the floor and hence compensate non-compliant mechanics. Infrared distance measurement sensors could be attached to the feet to determine which feet are touching the ground - this information would be useful when visualizing the robot model in RViz (see section III-B). We were able to read out the sensor data by connecting the Bioloid's CM510 backpack control unit via the RS232 headphone jack to a windows laptop running Robotis's RoboPlus Manager software.

Unfortunately the sensors are only accessable via the CM510 unit. They are not addressable from the Dynamixel bus. However, both the HOly setup (using dynamixel_motors as backend) and the provided bioloid_interface use the Dynamixel bus to control the Bioloid. It would be possible to port the system to use the controller bus, as the dynamixel bus is controllable from there (see also section II-B4), but this involves high development effort. This is why we went with the Humanoid Olympics lab's conventional approach and dropped our hopes of automatic mechanics compensation through sensor feedback.

It should be noted however, that we can still access the Dynamixel motor-internal sensors that report on position error, motor effort, internal temperature and other values.



(a) Normal (b) Modified

Fig. 3: Reinforced Leg Attachment

4) Motors: We're not the first to use the provided Bioloid robot. Past repairs led to the use of differing Dynamixel types, namely the original AX-12+ and its mechanical upgrade AX-12+. We upgraded the remaining, heavily stressed leg motors.

Unfortunately, the Dynamixel AX-12 servo motors are notoriosly bad at reaching or maintaining their goal position when under load. This is because the internal motor controller is a P-only type controller, i.e. the applied motor current (and hence torque) is directly proportional to the positional error. The motors would have enough power to lift the entire robot using a single joint. It's just that when a goal position can only be reached and kept by using some torque, the motor controller never tells the motors to use their entire available force.

The inability of the motors to reach these certain goals is an issue, because it makes tasks like climbing stairs, where the accurate position of each limb is vital to maintaining the robots balance, virtually impossible. We tried to fix this in three ways.

First, we upgraded the motor controller firmware to the latest official release, hoping that this major annoyance might already be resolved. This was not the case. We did, however notice the custom *Morpheus* AX-12 firmware[3], developed by Ricardo Marinheiro of the ActuatedCharacter research project. The Morpheus firmware greatly enhances the Dynamixel motor's capabilities. It offers a very fast 1000Hz internal control loop, 600Hz update frame rate, multi-master communication and, most importantly for us, advanced PID and PIV controllers. After successfully flashing[4] the Morpheus firmware on one of the Dynamixels, we tested it with the included Windows control GUI and could confirm the superior quality of control that manifested in a much more stiff Dynamixel motor joint. There was one big problem, though: The Morpheus firmware achieves the higher update rate and multi-master capability with a completely custom communication protocol, which not only causes bus collisions with unmodified dynamixels, but breaks all compatibility with the conventional Dynamixel control API. The Morpheus firmware is meant to be controlled from the Bioloid's backpack control unit, which the author provides another custom firmware for. However, the Bioloids in the Humanoid Olympics lab use a different version of this controller, that is highly likely incompatible with this firmware. Unfortunately, no other API is provided that could be used to talk to the motors directly via the Dynamixel bus. The only way we could control the upgraded motors from our Linux system was by porting the low level communication protocol from the incompatible backpack control unit's firmware source's C code to the dynamixel_motors Python communication protocol code. This, of course, guaranteed high development effort, which we were, given our determination to finish the project on time, not willing to invest.

So we sticked to the official firmware and searched for other ways to get rid of the residual offset error. While the controller lacks any intergral or differential parts, the proportional term is fairly freely configurable in Dynamixel RAM. Specifically, the user can control the *complicance margin*, *slope* and *punch*. The parameters are illustrated in figure 4.

This can be done by calling ROS services provided by the dynamixel_motors backend, e.g. `rosservice call /L_HAA_controller/set_compliance_slope 12`. By clearing margin and punch to 0 and reducing the slope to 12 the Dynamixel becomes compliant because the motor controller reacts earlier and more intensively to position errors. The movements look less smooth but are more compliant, e.g. legs are lifted higher. Another drawback is that especially when all motors have their parameters changed as described, a nudge can make the robot oscillate, giving it "the shakies´´. We could

[3]https://actuated.wordpress.com/ax12firmware/

[4]According to the Dynamixel documentation, it is not possible to brick a Dynamixel via the Dynamixel bus

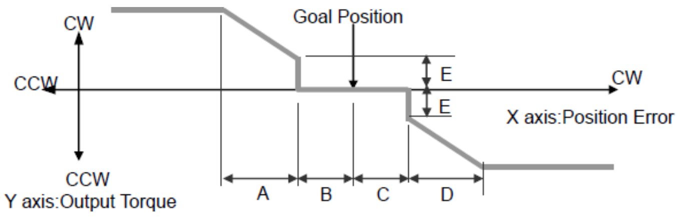


Fig. 4: Compliance Margin (B, C), Slope (A, D) and Punch(E). Source: http://support.robotis.com/en/product/dynamixel/ax_series/dxl_ax_actuator.htm

overcome this by setting the compliance parameters only when and where needed: The arm motor's parameters are never changed, and leg motors are configured only for stair climbing.

While setting the compliance parameters yields a big improvement, the residual motor offset is still not eliminated completely. We compensate for this by adding empirically determined values to our poses (see section **??**).

C. Human Machine Interface

1) Gamepad: To control the robot while it performs tasks, we use a Sony Playstation 3 controller providing 10 buttons and 2 joysticks. Figure 5 indicates our allocation: (R1-L2) Change mode, (X,O) punch, (O) cheer, (left joystick) acceleration, (right joystick) step length and turn.

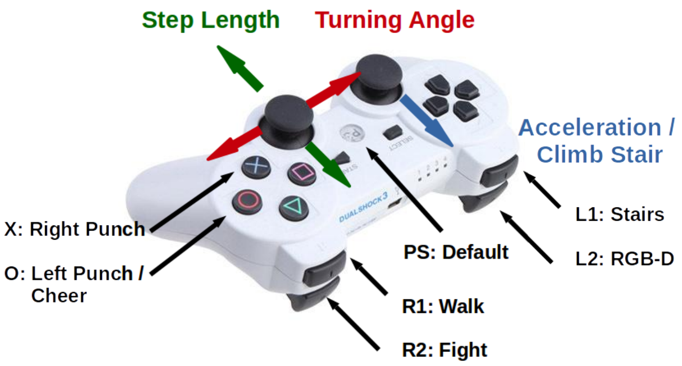
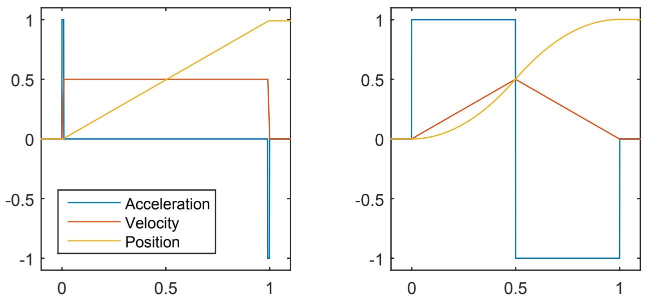


Fig. 5: PS3 controller input

2) RGB-D Sensor: The second source for user input is the RGB-D sensor of the ASUS Xtion Pro[5]. It is a combination of a RGB camera and a depth sensor. The device is used in the *RGB-D* mode where the gestures of the tracked person in front of the camera are imitated by the Bioloid robot.

[5]https://www.asus.com/de/Multimedia/Xtion_PRO/



(a) Constant Velocity (b) Constant Acceleration

Fig. 6: Motion Planning Policies

III. Software

A. Custom Controller

The ROS package bioloid_interface[6] is the Dynamixel controller backend provided to students of the Humanoid Olympics lab. It does a good job at controlling the Dynamixel position but lacks any motor feedback features. Dynamixel_motor[7] on the other hand is a ROS python package that can control the Dynamixels as well, but exposes the complete Dynamixel API through a configurable action server. We followed the tutorials for the Dynamixel stack[8] as well as the CKBot tutorials [9] on how to integrate it with MoveIt.

B. MoveIt

As http://moveit.ros.org/ puts it, MoveIt is state of the art software for mobile manipulation, incorporating the latest advances in motion planning, manipulation, 3D perception, kinematics, control and navigation. It provides an easy-to-use platform for developing advanced robotics applications, evaluating new robot designs and building integrated robotics products for industrial, commercial, R&D and other domains. RViz is a 3D robot visualization that integrates well with Moveit.

1) Planning: By default, MoveIt uses the *OMPL* motion planner. This is generally a good solution, because it allows things like moving the right arm to the left of the left arm by planning a collsision-avoiding path for the right arm around the left arm. The other advantage is that it usually moves end effectors in a straight line in XYZ space, which looks good and reduces unintended collisions with other world objects.
Nevertheless, we were not satisfied with the way the planner moved our limbs, as it seemed to use a constant-velocity policy and only generated about 7 waypoints along the path. Constant velocity causes very high acceleration during both the beginning and end of the movement (see figure 6). According to $F = ma$, high forces act on the robot, causing it to shake and / or even fall. Our idea was to keep these forces to a minimum by using constant acceleration during the entire movement. We simplified the calculation by assuming a straight line in motor joint space between start and end point. Using the motor joint space makes calculations much easier, but results in slightly curved movements in the real-world XYZ space. However, this was not a problem when we shifted the path of an otherwise floor-colliding end effector roughly 5mm upwards. Another benefit of doing the path calculation ourselves was being able to generate any number of waypoints. Using constant acceleration and about 100 waypoints resulted in an extremely smooth motion that greatly improved our robot's overall stability.

2) Inverse Kinematics: The kinematics equations of a robot can be used to obtain the pose of an end-effector, e.g. of the left foot pad, given a joint angle configuration of the corresponding kinematic chain. To do so, it has to be defined how the joints are connected, which is done in the Unified Robot Description Format (URDF). For designing a robotic task in real world (like walking), however, it is very beneficial to compute the joint states of a chain, given a real world pose. For example, it is useful to obtain the joint angles which move the right foot-pad 1cm forward. This is called inverse kinematics.
Inverse solutions of the kinematic equations can be computed either numerically or explicitly. MoveIt implements the numerical KDL library[10] by default. While the solver yields acceptable results for the feet, it performs poorly for chains with less than 6 degrees of freedom (DOF). Furthermore, the computational time of a numerical solver can be an issue for real-time applications (see section III-D4). Hence, we use the explicit inverse kinematics tool IKfast developed by OpenRAVE[11]. IKfast is able to analytically solve various chain setups for explicit inverse equations. The computational time of this explicit method is stated as $\sim 4\mu s$ compared to $\sim 10ms$ for the numerical approach. Furthermore, it provides all possible joint state solutions. Thus, we applied IKfast for each of the 4 kinematic chains, using the URDF file in *.dae* format with rounded values. The ROS package *moveit_ikfast* provides tools for conversion and rounding [12]. Additionally, it is possible to generate custom MoveIt plugins for each solver, which are easily integrated within the framework using the MoveIt setup wizard. Thus, the custom solvers can even be used within RViz. In order to support multiple IKfast plugins, the class names of the generated plugins have to be changed manually, otherwise only one is found. Due to the fact that the arms only feature 3 DOF, the 3D

[6]https://github.com/brennand/bioloid_indigo
[7]http://wiki.ros.org/dynamixel_controllers
[8]http://wiki.ros.org/dynamixel_controllers/Tutorials
[9]http://www.seas.upenn.edu/~juse/tutorial/ckbot_moveit.html
[10]http://www.orocos.org/kdl
[11]http://openrave.org/docs/latest_stable/openravepy/ikfast/
[12]http://moveit.ros.org/wiki/Kinematics/IKFast

direction of their end-effectors is solved for. One important last step is to accurately define all joint limits of the robot within the URDF file. Wrong values can lead to undefined behaviour, as impossible trajectories are found. Furthermore, we constrain the elbow joints to be straight for walking and climbing stairs.

C. RoboPoses

The process of walking can be divided into a sequence of poses, like shifting the weight to one side or lifting a foot. In order to define such poses we implemented two classes, namely *LimbPose* and *RoboPose*. A *LimbPose* contains fields for all six degrees of freedom (roll, pitch, yaw, X, Y, Z) and can be assigned to one of the four limbs of the Bioloid (left/right-hand/foot). Both arms have three degrees of freedom and ignore the values of X, Y and Z. A *RoboPose* can contain an arbitrary number of *LimbPoses* and can be identified by its name. Each class contain a set of convenience functions and the overloading of standard operators, which enables us to combine them easily in our code structure and use a generic programming style. The hierarchical structure and the interaction between these two classes is visualized in figure 7. The example shows that two different poses, which each contain new goal states for two different limbs, can be combined by using the + operator. The coloring of the boxes is indicating the different limbs. The resulting *RoboPose* combines the values of the green colored limb and adds the original values for the other two.
The great advantage of this principle is, that it is possible to store a set of absolute poses and build up a variety of combined relative poses.

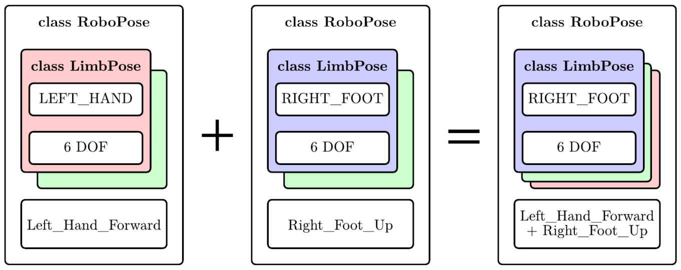


Fig. 7: Class RoboPose and LimbPose

D. Operational Modes

The program flow of our *HOly Main Node* is visualized in Figure 8. It is a three layer hierarchical structure, divided into a top layer for the mode selection (1st Layer), a second layer (2nd Layer) describing the sequential flow of a certain mode and a third layer (3rd Layer) executing a sequence of movements defined by different *RoboPoses*.
After the program starts, the robot will go into its default position and wait for an input by the user for selecting the mode. As an example the user pushes the *R1* button and the program switches to the second layer immediately, initializing the *Walking FSM* and thereby resetting all internal states and values. Using the joystick to control the step length, acceleration and turning angle for walking will trigger an update of the *RoboPoses* using the transmitted input parameters of the *joy_node* from the PS3 controller. The robot is now *IN ACTION*, switches to the next state of the second layer and cannot change its mode till the current action is completed. Each state executes a sequence of movements defined by corresponding *RoboPoses*, which constitutes the third layer of the program flow. Without any user input, the *Walking FSM* will reinitialize and the mode can be changed again.
The *Fighting FSM* and the *Stairs FSM* do function analogously. The *RGB-D Mode* mode on the other hand only consists of a single state repeatedly executing the observed movements by the RGB-D device and can be exited after each cycle.

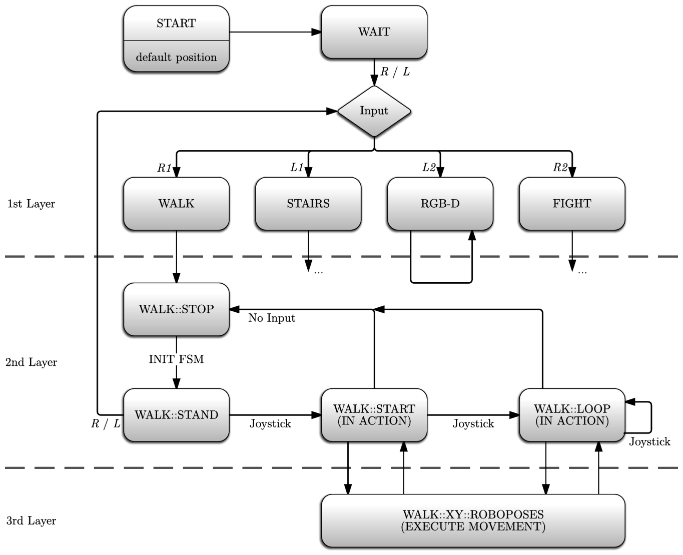


Fig. 8: Hierarchical program flow

1) Walk: One challenging task of the Humanoid Olympics project is to design a walking gait for the *Bioloid* robot. Due to a lack of sensors it is really hard or even impossible to implement stable dynamic walking. Thus, our gait is static, which means that the robot can stop at any time during walking without falling. Nevertheless, we wanted the algorithm to be highly adaptable, intuitive and reusable. The result is visualized in Figure 9. The walking gait is partitioned into 3 phases: Init, Loop and Stop. Each phase incorporates a finite state machine consisting of the different poses. The numbers indicate the absolute RoboPoses, which are constructed using reusable relative poses. In total, there are 9 different absolute poses within the gait. Furthermore, white indicates that there is no weight on the foot, gray stands for dual support and black illustrates single support. Inverse Kinematics, as explained

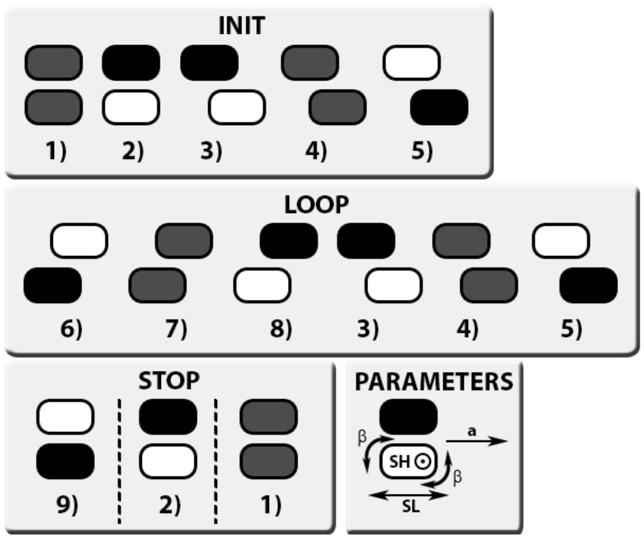


Fig. 9: Walking gait's Init, Loop, Stop sequences and its online parameters β, SH, SL, a

in III-B2, is used to compute and parameterize the poses. The used parameters are: step length SL, step height SH, turning angle β and maximum acceleration a (see **??**). The poses are being recomputed before each action and thus the parameters can be changed online while walking using a game-pad.

In order to transfer from dual to single support, two things showed to be essential. Firstly, the whole robot has to be tilted to the side until the projection of the center of gravity (COG) onto the ground plain lies inside the rectangular area of the foot-pad. This is achieved with adjusting the pitch of the foot-pad. Secondly, the other foot has to be stretched due to the fact that the body is tilted. This can easily be verified considering the Equation

$$Z' = \tan(\alpha)d, \tag{1}$$

where Z' is the needed elongation, α the pitch angle and d the distance between the legs. Hence, the non-supporting leg pushes the other one into single support. For better balancing stability the arms are raised in the leaning direction.

As a first move in Init, the robot shifts its weight to the left (2) as explained before. Because there is no weight on the right foot, it is now possible to shift it forward (3). In (4) the COG is shifted back between the legs, thus there is dual support. In the last step of Init the weight is shifted forward (where it was before the step) and to the right. From this configuration a transition to either Loop (6) or Stop (9) is possible.

Within the loop phase the whole walking sequence is performed. From (5) to (6) the left foot is moved all the way from $-SH$ to $+SH$, which is followed by a dual support phase in (7). In (8) the weight is shifted to the front left and the remaining 3 poses are identical to the Init case.

The Stop phase can be entered either from (5) or (8). After moving the foot back to its default state in single support (9,2), the robot goes back to its initial pose (1).

As shown in the bottom right box, the parameters are updated before each single support pose. Thus, it is possible to take small steps to accurately approach a desired position and also to walk slowly for safe balancing. This can be combined with adding a turning angle β in order to safely go around obstacles.

2) Stairs: Climbing stairs is conceptually similar to walking with the difference of going up. Hence, due to our relative, parameterizable poses, we tried to reuse as much as possible. Our stairs sequence, illustrated in figure 10,

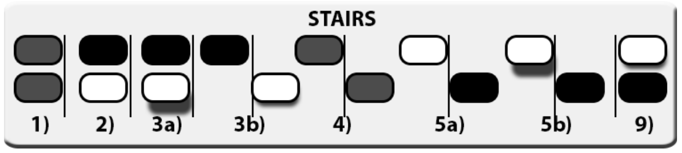


Fig. 10: Stairs sequence, numbers correspond to walking gait

is similar to the Init and Stop phases of walking. Hence, instead of explaining each pose again, the differences to walking will be stressed in the following. In our walking gait, the free foot is always lifted and pushed forward at the same time. In order to prevent a crash, we split the pose into two single poses, one for lifting (3a,5a) and one for pushing forward (3b,5b). Another difference is that the step height and step length have to be very high to climb the stairs accurately. However, the motors and mechanics of the robot are not able to perfectly hold position when one foot leaves the ground completely. Hence, we consider these effects using additive compensate RoboPoses. For example, in pose (3a) the lean angle is increased slightly to keep balance. From pose (4-9) the right foot is not streched like in walking, instead the left foot is brought back before. The most difficult part is the shift to the right foot in pose (5a). In case of walking, we simply did this by stretching the left foot, but it is not long enough to follow this strategy here. Instead of moving the whole body forward, we tilt the whole robot forward and lean to the side to balance. At the same time the left foot pushes its toes to the ground to force the robot into the wanted right balance position. Afterwards the left foot is brought back (5b,9). Finally, the robot is upright again in (1) and ready to take the next step.

3) Fight: The *fighting* mode is not a mandatory goal in our project but has been the past years. We decided to implement the feature anyway because it was not very time consuming with the software base we already had.

After entering the *fighting* mode the robot will go to a stance which lowers the center of gravity. It resembles the basic position a Kung-Fu fighter takes in order to protect himself from loosing balance after fending off an attack. The effectiveness has been proven by trying to push the robot from different sides. An enemy robot facing the Bioloid can be attacked with a left or right punch as simulated in Figure 11. Pressing the X button will execute the punch and the robot will return to its basic stance afterwards. The goal of an attack is to overbalance the enemy robot. When a punch hits the defended the button can be pressed again to execute a follow up attack which pushes the forward-facing arm sideways in order to lever the enemy robot to the side. The same kind of attack can be executed with the left arm as well.

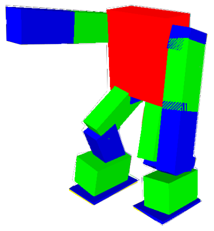


Fig. 11: Fighting mode: Right punch

4) RGB-D: Right at the beginning of the project, when we defined our goals, we had the idea to directly translate human to robot motion using an RGB-D camera. To do so, three steps have to be performed in a loop. Firstly, the human body parts like hand, elbow, shoulder have to be tracked in real world coordinates using the RGB-D camera. The resulting coordinates have to be adapted to the robot's coordinate system and capabilites (e.g. arms only have 3 joints). Finally, inverse kinematics has to be applied continuously in order to move to the desired pose in real-time.
As the Asus Xtion Pro is based on PrimeSense technology, it is compatible with their well-known libraries *OpenNI2* and *NITE2*. Hence, we are able to use the ROS package *skeleton_tracker*[13], which is based on *openni2_tracker* and provides *tf-transforms* of the human body parts. Thus, we calculate the translation vectors between shoulder and elbow, elbow and hand, as well as hip and foot. As the IKFast solution of the arms only considers the direction of the pose, it is necessary to convert the vectors into two angles, which are denoted as α and β in the following. First of all, the coordinate frame of the RGB-D camera has to be aligned with the robot frame. This is done by exchanging the axis subscripts

$$(\,X\,)\,YZ_{rob} = (\,X\,)\;-ZY_{hum}. \tag{2}$$

Figure 12 illustrates the change of coordinate systems and the calculation of arm angles. The angle which moves the arm forward or backward is named α and is defined as

$$\alpha_{l,r} = \arccos\left(\frac{Z_{hum}}{\|t_{eh}\|}\right) - \frac{\pi}{2}, \tag{3}$$

[13]https://github.com/Chaos84/skeleton_tracker

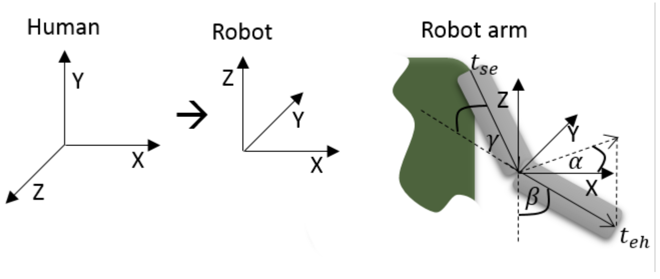


Fig. 12: Coordinate transform from RGB-D to the robot's frame and derivation of α, β, γ

with the translation vector between elbow and hand t_{eh}, its Z-component and the α angle of the left an right hand. β is the angle between body and hand, hence

$$\beta_{l,r} = \arccos\left(\frac{X_{hum}}{\|t_{eh}\|}\right) - \frac{\pi}{2} \tag{4}$$

Furthermore, to account for angles larger than 90° in β an offset is added depending on the hand being above or below the shoulder:

$$\beta_l = \frac{\pi}{2} + (\frac{\pi}{2} - \beta_l), \beta_r = -\frac{\pi}{2} - (\frac{\pi}{2} + \beta_r). \tag{5}$$

Given these two angles the direction of the robot's hand can now be set according to the human pose. In order to transform the elbow as well, we calculate the angle between the vectors shoulder-elbow t_{se} and elbow-hand t_{eh}

$$\gamma_{l,r} = \arccos\left(\frac{t_{se}t_{eh}}{\|t_{se}\|\|t_{eh}\|}\right). \tag{6}$$

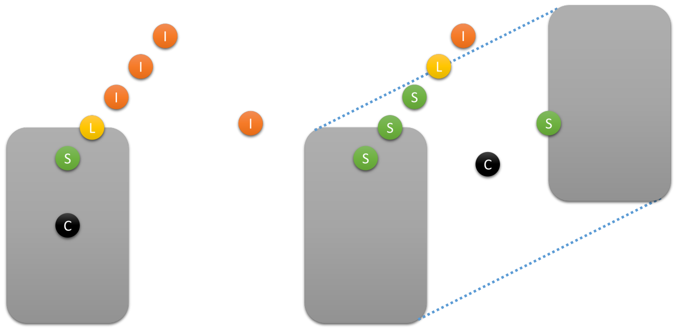
As the explicit solver yields all possible solutions, we can now choose the one with the suiting elbow angle γ.
In order to be able to crouch, the arithmetic mean of the leg vectors is computed and mapped to the robot's max value.

$$Z'_{rob} = \max_{rob}(1 - \frac{(\max_{hum} - Z_{hum})}{\max_{hum} - \min_{hum}}), \tag{7}$$

with the arithmetic mean of the hip-foot translation vectors Z values $Z_{hum} = (Z_l + Z_r)/2$ and the resulting robot Z offset Z'_{rob} which is applied for both feet.
The resulting parameters are saved in a RoboPose and solved using the four IKfast explicit solvers. As there is no need to plan a path, the motors are directly set with the updated joint angles at maximum speed. All the steps are performed in a loop and experiments showed that the robot is able to follow the human motion in near real-time as can be seen in video 1.

E. Discarded Approaches

1) Center of Gravity: One idea that came up in our project-kickoff brainstorming was automated parameter tuning via reinforcement learning. The agent's reward would be based



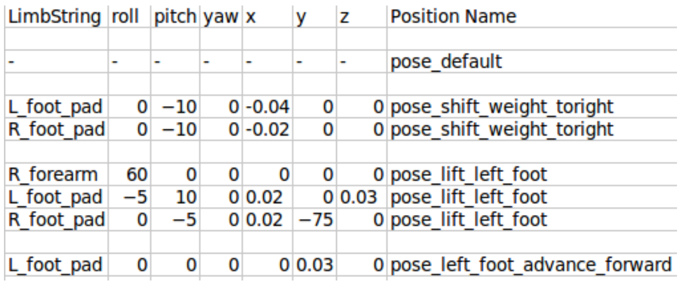
(a) Single support phase (b) Dual support phase

Fig. 13: Projection of the center of gravity on ground plane (S: stable, L: labile, I: instable) and center of convex hull of supporting planes (C).

on how stable the robot's gait is. We never implemented any machine learning techniques, but the stability level was an interesting metric nevertheless, as it helps to assess the quality of static robot poses. The level of stability couldn't be directly measured with sensors (see section II-B3). Instead we look at the distance of the projection of robot's center of gravity on the ground plane to the center of area of the convex hull of the supporting plane(s), as illustrated in figure 13.

We can calculate the center of gravity via the (literally) weighted sum of the transform vectors we receive from our robot_state_publisher. We obtained the weights by actually measuring each part with a scale during the hip modification disassembly (see section II-B2). Since we were only using this for uncritical debugging purposes we simplified the calculation by using the transform vectors directly instead of calculating the vector to the center of gravity of a single part (parts are heavier on one side). Even with the simplification, the results were pretty convincing.
A minor drawback was that at runtime, the orientation of the floor plane is unknown. This could be fixed by adding sensors to the feet (see section II-B3). We could avoid this problem by setting RViz to orthogonal view with the base transform set to a foot pad we knew touched the ground. With the robot visualization's transparency set to about 0.5, we could see how well the robot's center of gravity was over the supporting surface (i.e. the foot pad).

2) Run-Time Adaptable Pose Database: A position is stored in a *RoboPose* object as described in **??**. The basic parameter of a single movement can be derived from analyzing the procedure of e.g. walking and dividing it into sub-sequences. Each pair of position and its six values when executed, will not necessarily lead to the desired result because of the mechanical bending of the robot. Therefore the bending has to be compensated, which proves to be a time consuming task. In order to find the correct compensation values, we have to adapt them while the program is already running and avoid recompiling the code after each change.
To solve this issue we implemented a *CSV* parser. The file shown in figure 14 defines a position by assigning the the six necessary values to each limb and combining them into a single position. One position is separated from the other by an empty line. A *RoboPose* can ether build up on the default position or on the previous position to generate consecutive movements. This approach allowed the fast development of successful movement sequences.
After we found the correct values and transferred them into the code, the tool was not needed anymore, but can be used to develop other movement tasks in the future.



LimbString	roll	pitch	yaw	x	y	z	Position Name
-	-	-	-	-	-	-	pose_default
L_foot_pad	0	−10	0	-0.04	0	0	pose_shift_weight_toright
R_foot_pad	0	−10	0	-0.02	0	0	pose_shift_weight_toright
R_forearm	60	0	0	0	0	0	pose_lift_left_foot
L_foot_pad	−5	10	0	0.02	0	0.03	pose_lift_left_foot
R_foot_pad	0	−5	0	0.02	−75	0	pose_lift_left_foot
L_foot_pad	0	0	0	0	0.03	0	pose_left_foot_advance_forward

Fig. 14: CSV example file

IV. Conclusion

We were not only able to successfully implement all required tasks, namely walking and climbing stairs, but also the additional modes *fighting* and *RGB-D*. By achieving all mandatory goals well in time, we were able to integrate all optional features defined at the beginning. Although very challenging at first, we do use MoveIt as underlying framework, making our implementation highly compatible for future extension. Using MoveIt enables easy visual robot interaction in RViz. Due to the provided *bioloid interface* being incompatible with MoveIt, we were required to develop and deploy a custom controller. Furthermore, we generated explicit inverse kinematics solver plugins using OpenRAVE's IKfast, outperforming any numeric solution. Our custom *RoboPose* arithmetic turned out to be very beneficial for constructing parameterized absolute poses from relative displacements.

In terms of software, the biggest issues we faced were setting up MoveIt, as it is sometimes over complicated and in great parts poorly documented, as well as the provided oversimplified xacro/urdf model. On the hardware side, we struggled against under-powered motors, which are unable to maintain their goal position under load and overheat quickly. The mechanical bending caused by the robot's own weight only made it worse.

We gained a deeper insight in using ROS, advanced programming in C++ and hands-on experience with humanoid robots. Furthermore, it was very interesting to realize the complexity of transferring a human locomotion to a robot.

Appendix

HOly's code is open source. You can find it at https://github.com/jdsika/holy.

This document is also available at https://raw.githubusercontent.com/jdsika/holy/master/Documentation/HOly.pdf

Video 1 demonstrates our solution. Watch it at https://youtube.com/watch?v=PQb3ZKncxoc!

Video 1: "Humanoid Olympics: Team HOly"