Developing a Reactive and Dynamic Kicking Engine for Humanoid Robots

Frederico Bormann, Timon Engelke,
and Finn-Thorben Sell

Universität Hamburg
{7bormann,7engelke,7sell}@informatik.uni-hamburg.de

Abstract. [FTS] One of the most essential yet not satisfactorily solved tasks in humanoid robot soccer is to kicking a potentially moving ball in varying environments. We present a new approach to dynamically and reactively plan and execute such a kicking movement. Free and Open Source frameworks and libraries are used in compliance with agreed upon standards to achieve maximum interoperability. The software is also configurable to suit many environments.

Keywords: Robotics · Humanoid · Control · Motion · Open Source

1 Introduction[TE]

The RoboCup[1] is an international competition for robots founded in 1997[12]. It consists of multiple leagues that compete in different domains. One of these domains is the Humanoid Soccer League where two teams of up to four humanoid robots play soccer against each other. The rules are similar to the FIFA rules but are often simplified[2].

Therefore, as in usual soccer games, kicking the ball to score goals or pass to team mates is very important. Other tasks, for example team behavior or penalty shoots, also strongly rely on well directed kicks. Since positioning a robot to face a certain direction or have a certain distance to the ball takes time and is still a challenge, the kicking engine should be tolerant of the ball position and be able to kick in different directions.

Additionally, the robot has to be able to stand on a single foot during the kick. Due to different surfaces and environmental conditions, the robot must keep its balance by using sensor feedback.

The approach that is presented in this paper covers both challenges. After a presentation of different existing solutions to the kick problem in Section 2, our approach is presented in Section 3. First, the decisions required to determine a kick motion dynamically based on the ball position, kick direction and kick speed are presented in Sections 3.1 to 3.4. Then, our stabilizing method is described in

[1] https://www.robocup.org, Accessed August 29, 2019
[2] http://www.robocuphumanoid.org/wp-content/uploads/RCHL-2019-Rules-final.pdf, Accessed August 29, 2019

Section 3.5, consisting of two different approaches leading to a reactive control of the robot's pose.

In Section 4, our implementation is described. We are using ROS[11], the *Robot Operating System*, an Open Source framework for developing distributed and parallel robot applications. Its large community and high interoperability standards ease development of new applications since libraries are readily available and usable [1].

Our results are summarized in Section 5 and future work is discussed in Section 6.

2 Related work^{FTS}

Planning and executing robot movement is a well established but still active field of research. In this Section three main types of movement planning/execution from different robocup teams are shown. These approaches were chosen because the first two are the most widely used ones and the third is very similar to our own.

2.1 Static Animation^{FB}

One of the most obvious approaches is to determine a reasonable kick motion manually, so that all motor positions are determined in advance by trial and error or a talented individual. When a kick shall be executed, the robot only needs to interpolate between these saved key frames and play back the resulting sequence. This is called animation and is the approach currently used by the Hamburg Bit-Bots [2].

The main advantage lays in its technical simplicity, i.e. it is easy to implement and there are only very simple calculations and no extra data needed at runtime. But this also marks its largest weaknesses: First, the robot is not able to react to the position of the ball dynamically. That is why we need to make sure that the ball is in a viable position relative to the robot's foot before the kick, which could even be impossible if the ball is moving. Second, the robot is very vulnerable to falls if the conditions are not quite the same as of the time of recording, e.g. different quality of underground material.

2.2 Walk Kick^{FB}

Another approach not requiring much new development is to make use of the already implemented walking algorithm[10]. With this method kicking a ball is achieved by taking a step with a large overshoot in the desired direction. The quality of this method significantly depends on the stability of the walking. In the case of the Hamburg Bit-Bots "a good walking on the artificial grass" was achieved in the past [2].

Apart from that, it implies one problem already seen with static animation in Section 2.1, because it is not able to react to the position of the ball dynamically, let alone desired kicking directions. In addition to that, it results in the robot only being able to kick while walking.

2.3 UT Austin Villa's Approach[FTS]

UT Austin Villa is another team participating in RoboCup that has developed an approach which successfully learns an appropriate kicking motion via machine learning techniques and then executes that in a non-static manner. In theory this is a great approach because it allows for dynamic adaption to the balls movement and can also deal with a change in the environment. The only caveat is that it relies heavily on a good simulator to train the algorithm and has only been shown to properly work in the same simulated environment. [8]

3 Approach[FB]

In the following section, we are going to present the kick procedure. For this purpose, we divided this process into several phases. Their properties will be discussed in Section 3.1.

When the specific foot poses for these phases are determined, we have to face several challenges. These challenges as well as possible approaches to take the necessary decisions for solving them are also described in the following section.

In addition to that, we have to consider the problem that the robot is vulnerable to falls if the boundary conditions change. We will consider stabilizing methods to compensate that.

3.1 Kick Execution Phases[FB]

First of all, we have to take a more detailed look at the task the robot should fulfill. In our case we will examine the following aspects of the task: The robot is supposed to kick a ball placed at a known position in a given direction with a certain speed. This marks the three input values needed for the planning process:

1. ball position relative to the robot
2. kick direction relative to the robot
3. kick speed

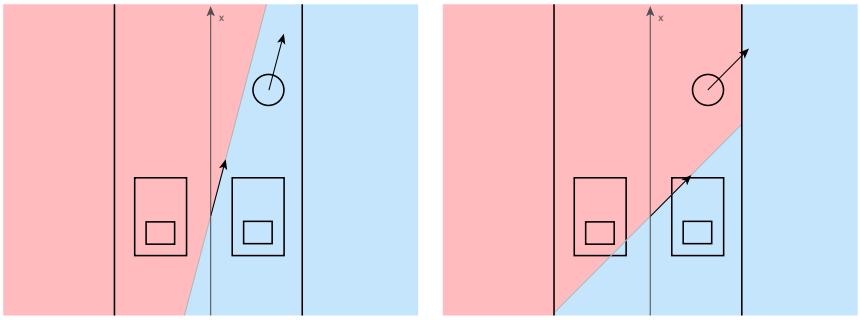
Now the kick planning process has to turn that information into a proper movement trajectory of the kicking foot. To simplify the consideration of the different challenges related to that, we have decided to split the movement into five phases:

1. **Raise kicking foot**
   The robot has to raise the foot which will kick the ball up to a proper height. The value of that height does not have to be changed according to the input values named above, but it should be adjusted for example to fit the ball size.
2. **Kick ready position**
   The risen foot now has to be placed in a position near the ball suitable for executing the kick. This is necessary because moving the foot directly to the ball position does usually not result in the desired kick direction. This kick ready position is explained further in Section 3.3.

3. **Kick**
   The kick is a quick movement along the specified kick direction with the wanted kick speed. After this phase the most important part is completed and our remaining goal is to reset the robot into a default position.
4. **Move back**
   As a second to last step we return the kicking foot to the position it had after phase 1.
5. **Lower kicking foot**
   Finally, the kicking foot is lowered until it touches the ground. At the end of this phase the robot is expected to stand stable in its default position.

3.2 Kicking foot selection[FB]

The first decision that has to be made is the choice of the kicking foot. This selection should optimize the ability of the robot to reach a proper kick ready position that is close enough to the chosen foot and avoid self collision with the other leg.



(a) The desired kick direction is almost parallel, so it is appropriate to use the right foot.

(b) Kicking with the right foot would require to reach back from an improper position.

Fig. 1: The figure shows the feet of the robot, the ball position and the kick direction. Balls in the red area should be kicked with the left foot and vice versa.

If the desired kick direction is (almost) parallel to the x-axis, it seems reasonable to simply divide the area in half along the x-axis[3] and choose the foot closer to the ball. But it fails for example if a ball on the right has to be kicked to the right, since this could require the right foot to reach left and collide with the left foot.

We address that problem with a criterion taking into account both the ball position and the kick direction. To accomplish that, we rotate the dividing line

[3] forward facing axis, as defined in ROS Enhancement Proposal 103 (Standard Units of Measure and Coordinate Conventions)

separating the areas determining the kicking foot from each other. Thereby, we change the chosen foot for the kick if the kick direction is rather sideways (Figure 1a). But it also maintains the foot selection for simple cases as described above (Figure 1b).

To formalize that idea, let α be the angle between the x-axis and the position vector of the ball and β the angle of the kick direction. Then the function f calculating the kicking foot is defined as follows:

$$f(\alpha, \beta) = \begin{cases} \text{left} & \text{if} \quad \beta - \alpha < 0 \\ \text{right} & \text{else} \end{cases} \tag{1}$$

In addition to that, we define that this criterion should only be applied within a corridor with a certain width as shown in Figure 1. Balls outside of this corridor should always be kicked with the left or right foot respectively, since kicking these balls with the other foot would require it to move too far away from its original position, which could even be impossible.

3.3 Kick ready position[FB]

In the description of the second phase of the kick process we mentioned a kick ready position ensuring that it is possible to kick the ball into the wanted direction. That position has to be somewhere in the direction opposed to the kick direction in a certain distance from the ball. We call that the windup distance. An example can be seen in Figure 2.

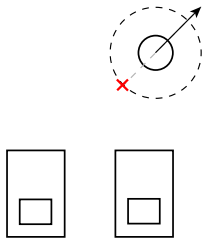


Fig. 2: Determination of the kick ready position (red cross). Windup distance is marked as a dashed line.

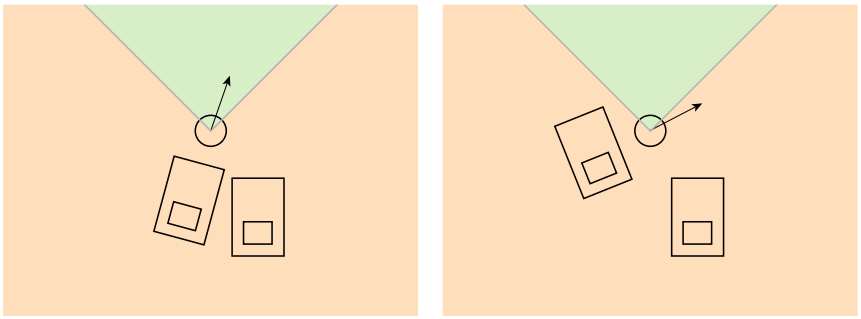
To formalize that idea, let b be the vector describing the ball position from the support foot, d the kick direction and w the windup distance. Then the kick ready position p can be calculated as follows:

$$p = b - \frac{w}{|d|} \cdot d \tag{2}$$

3.4 Kicking foot rotation[FB]

The preceding paragraphs provide a solution without rotating the kicking foot. However, we strive to optimize the ball contact. So the foot should be turned in such a way that it touches the ball either with its front or its side. There is no special preference for one of these options in general. That is why the robot shall choose the option easier to reach, e.g. the foot should not be turned by more than 45° around the z-axis.

That means, if the kick direction is closer than 45° to the x-axis, we will rotate the foot so that its yaw matches the kick direction (Figure 3a). Otherwise, we will rotate the foot in such a way that either the outer or the inner side of the foot directly faces the ball depending on the difference between kick direction and the x-axis.



(a) Kick direction is closer than 45° to the x-axis, so the kick is performed with the front.

(b) Kick direction differs more than 45° from the x-axis, the kick is performed with the side of the foot.

Fig. 3: Kicks in the direction of the green area should be performed with the front, kicks outside of that with the side.

3.5 Stabilizing[TE]

Unfortunately, it is not sufficient to calculate only the kicking foot pose because the robot will quickly reach an unstable state and fall to the side. Therefore, a stabilizing method must be added. We present two possible methods with and without sensor input respectively.

Open Loop Approach First, we implemented an open loop approach, i.e. the deviation from the output value is not monitored [6]. The underlying idea was to always keep the center of mass above the support polygon. The support polygon is the convex hull including all points that have contact to the ground [7]. Before and after the kick, both feet form the support polygon, during the kick it only consists of the support foot. The center of mass can be calculated from the

robot's pose and the knowledge about weight and individual centers of mass of the robot's links obtained from the URDF[4]. The approach was very simple and static, not regarding the dynamics of the kick, e.g. the accelerations when the ball is kicked abruptly. Therefore, some manual fine tuning was necessary to find an optimal point in the support polygon over which the center of mass could be placed.

Closed Loop Approach Our second approach was a closed loop approach, i.e. sensor input was used to readjust the movement based on the measured error [6]. From the four strain gauges installed in each foot of the used robot [3], the individual centers of pressure were calculated. During the kicking phase, the robot is only standing on one foot, the support foot. To stabilize over the support foot, the center of pressure of this foot should be in the center of the foot, i.e. all pressure sensors on the foot are equally loaded. When the robot leans significantly in one direction, the change is perceived by the sensors and the current position can be corrected. To achieve a useful correction, a PID controller [5] was implemented.

A PID controller is a controller that corrects an output signal based on its measurement of the current and past deviations from the desired state. In our case, the output is the point over which the center of mass (or, to simplify the calculations, the base link[5], which is very close to the center of mass), is positioned. The error is the difference between the actual center of pressure and the center of the foot. A PID controller is adjustable via three values, the proportional, integral and derivative factors.

The proportional factor is directly multiplied with the measured error. The result is set as the output. This alone is already a simple controller, but there are two problems that are treated by the other two factor. First, the P term alone is not able to correct errors that persist through time. Second, is often leads to overshooting, where the goal value is surpassed several times resulting in a quivering motion.

The integral factor approaches the first problem. It takes the accumulated error into account. Therefore, the errors of every preceding measurement are summed up, multiplied by the factor and added to the P term result. By adding the I term to the controller, steady offsets through time are treated correctly.

The derivative factor approaches the overshooting problem. In its simple form, the difference between the current and the last error is calculated (more sophisticated approaches use the derivative over a larger period of time), multiplied by the D factor and added to the P and I terms. Thereby, the error change is taken into account which adds a predictive effect and therefore further reduces overshooting.

In our case, separate PID controllers were used for the x and y axis of the foot because we noticed that the robot is much more unstable in the x axis than in

[4] Universal Robot Description Format
[5] ROS Enhancement Proposal 120 (Coordinate Frames for Humanoid Robots)

the y axis, i.e. it is rather falling to the side than to the front or back. Therefore, different PID configuration values where necessary.

4 Implementation[FTS]

The presented solution is implemented as a C++ application in combination with the ROS framework and this section will provide a more detailed explanation of our implementation. First the ROS-facing interface is explained. Then, a detailed explanation is given on how the positions explained in Section 3.1 are extrapolated to achieve smooth movement. Afterwards, the conversion from Cartesian coordinates to motor positions as well as implemented stabilization techniques are detailed. At last, an overview of the C++ code structure is given.

4.1 ROS Action Server[FTS]

First of all we need to present a stable and sufficiently complex yet easy to use interface to users of our software. Since it is a solution intended to be used in a ROS Context, a ROS node[6] was implemented. There are a few methods to define an interface for a node. The evaluated ones are:

Message passing [7] describes the idea of simply sending predefined messages between nodes on so called topics. While this should be a familiar approach to anyone using ROS and is therefore the most straight-forward and easy to use mechanism it does not provide the possibility to easily model more complex use cases such as the canceling of previous commands.

Service calls [8] provide another way of communication and are the equivalent to remote-procedure-calls in a ROS environment. This means they call a predefined procedure of another node and expect a result from that procedure. But kicking is a longer-lasting process with movement over time and not an action with an immediate result. That fact alone prevents the usage of this method.

ROS Action server [9] is a framework implemented on top of standard ROS messages and requires the definition of three types of communication:

1. Goal - Any client can send a new goal to the action server
2. Feedback - The server can periodically publish feedback on its current progress
3. Result - Once the goal is reached the server publishes a result

The action server has the choice to either accept a new goal or reject it. Goals can also be canceled by clients. Additionally the framework is meant to communicate between one server and a number of clients.

[6] `https://wiki.ros.org/Nodes`, Accessed August 29, 2019
[7] `https://wiki.ros.org/Messages`, Accessed August 29, 2019
[8] `https://wiki.ros.org/Services`, Accessed August 29, 2019
[9] `https://wiki.ros.org/actionlib`, Accessed August 29, 2019

The ROS Action Server is the chosen solution because of its ability to easily be used by other software to schedule a new kick-goal, receive periodic status updates and get notified when the kick was successfully executed while keeping the flexibility of canceling an invalid old goal or updating the ball position by simply sending a new goal.

4.2 Splines[FB]

As our kick planning approach only provides poses for certain points in time, we need a way to obtain reasonable information about the poses for any desired point in time from that, i.e. interpolate between them. Therefore we use quintic splines. By means of that, we only need to specify the position, speed and acceleration for these key frames at each axis in Cartesian space.

In the quintic spline interpolation, fifth degree polynomial functions whose first three derivations match the information given by the key frames are calculated, so that the resulting functions are continuous and smooth.

4.3 Inverse Kinematics[TE]

When Cartesian positions for the foot and the trunk are obtained by the splines, they have to be converted to motor positions. This calculation is called inverse kinematics and solutions are available in the form of libraries. As inverse kinematics library, we use BioIK [14,13]. It provides a large set of goal types that can be used to describe the desired position. The available goals contain conventional goals, e.g. PoseGoal, OrientationGoal or PositionGoal that are used to directly specify a links pose, orientation or position relative to the base link, as well as more high-level goals like the DynamicBalancingGoal, that can be used to balance the robot over a certain point by using weight information provided by the URDF. By instantiating one or multiple goals and assigning weights to them, a custom goal combination can be assembled.

In our approach, we used a PoseGoal to specify the pose of the kicking foot relative to the support foot. However, the pose is usually specified relative to the base link, which, though useful for non-humanoid robots, does not always make sense for humanoid robots where the base link is located near the waist. As a solution, it would be possible to provide a fixed position for the base link relative to the support foot and to calculate the position of the kicking foot relative to the base link. This solution makes sense when the position of the waist is to be explicitly set and is therefore used in the BitBots QuinticWalk. Since we wanted to use the waist for stabilization purposes (see next paragraph), we wanted to be able to specify a link's pose relative to an arbitrary other link. Fortunately, BioIk makes it very easy to implement new goal types by implementing a class derived from the base goal class. The new goal was called ReferencePoseGoal and provides an additional method, `void setReferenceLink(std::string name)`, that can be used to set the link that is the origin of the pose given to the goal. When the base link is provided as the reference link, this goal behaves exactly like the original PoseGoal.

For our stabilizing approaches, we were also able to use BioIK. We used the DynamicBalancingGoal, which, though not part of the original BioIK code base, was provided by its author. It was adapted like the PoseGoal to enable us to use a different reference frame than the base link. In our case, this reference frame was the support foot. The target over which the body should be balanced was adjustable by parameters, but has always been near the center of the foot. In our second stabilizing approach, this point was not manually configured but was instead the output of the PID controller.

When testing our approach, we observed that due to the DynamicBalancingGoal, often solutions were preferred where the kicking foot position differed from the input spline. The reason was that it is easier to stabilize the robot with the kicking leg than with the rest of the body. Therefore, we tried to control the base link position directly instead of relying on the DynamicBalancingGoal. Although the base link position is not the same as the center of mass, they are usually very close together and further differences would be eliminated by the controller.

4.4 Software Architecture[FTS]

In general the software is separated into three main areas which are

- Interacting with the ROS framework
- Constructing splines and extracting Cartesian positions in respect to the discrete current time from them
- Transforming these Cartesian positions into JointSpace and stabilizing the robot

Accordingly three C++ classes are implemented to do these tasks.

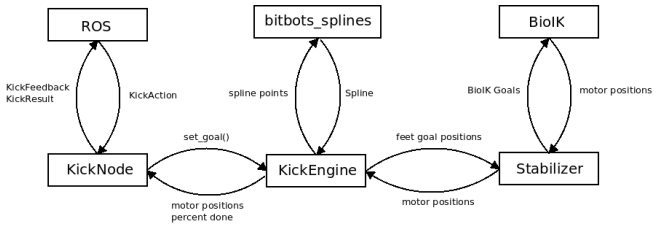


Fig. 4: Interaction of C++ classes with each other and external software

Figure 4 shows how these three classes each handle one responsibility or library. As this conceptual splitting also satisfies the *Separation of concerns*[9] design-pattern it was easier to develop and test single parts of the software. It also eases the development of additional features like control theory as described in Section 3.5.

Another important aspect which is accomplished through this splitting is the availability of clearly defined interfaces between not just to the outside world

(ROS) but also between different parts of the software. Figure 5 demonstrates how data flows through our software. It is obvious that the complete approach is much more complex and requires more steps but that information is hidden from the classes public interface and therefore reduces the complexity of the software as a whole.

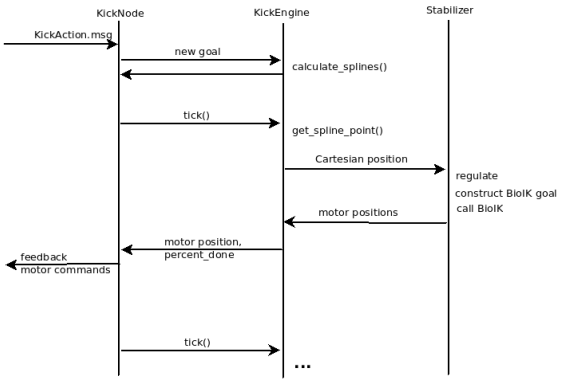


Fig. 5: Data flow between C++ classes

5 Results[FB]

Testing the Dynamic Kick in contrived situations and during games shows mixed results: While in the contrived situations the robot sometimes stays stable and is able to kick the ball at the specified position, we were not able to reproduce these results in game. But apparently, there is no significant improvement in stability compared to static animations (see Section 2.1).

On the upside, it is important to notice that our solution is easier to adapt to new situations by varying parameters, while static animations would require recording a new animation. In addition to that, the dynamic kick has a high potential, as it makes the kick adaptable to different ball positions at runtime and the stability was already improved by including the foot pressure sensors in the stabilizing process.

However, further evaluation with various ball positions and conditions is needed, since the methods described in the approach (see Section 3) could produce unreachable or unstable motor positions.

6 Future Work[TE]

In the future, we are planning to improve each of the three sections of our kick. In the ROS interface, we want to be able to reject unrealistic goals, i.e. goals that are too far away to be reachable. To achieve this, either a criteria for the

distance should be found or the trajectory should be simulated in advance to detect such unrealistic goals.

In the kick engine, we want to achieve a more fluent kick movement by not setting static time steps at which the foot should have reached a certain position but instead calculating the time steps based on the distance that should be covered. Additionally, our current foot selection criteria should be evaluated thoroughly.

In our stabilizer, self collision should be detected and avoided. A closer integration with the MoveIt! Motion Planning Framework [4] would be possible. To improve the stabilizing, further criteria could be included, most prominently the Inertial Measurement Unit. A better model that also includes the dynamics of the robot would also improve the stability since the kick motion contains abrupt movements that cannot be covered by a static model.

References

1. Bestmann, M.: Towards using ros in the robocup humanoid soccer league (2017)
2. Bestmann, M., Brandt, H., Engelke, T., Fiedler, N., Gabel, A., Güldenstein, J., Hagge, J., Hartfill, J., Lorenz, T., Heuer, T., Poppinga, M., Salamanca, I., Speck, D.: Hamburg bit-bots and wf wolves team description for robocup 2019 humanoid teensize (2019)
3. Bestmann, M., Güldenstein, J., Zhang, J.: High-frequency multi bus servo and sensor communication using the dynamixel protocol. In: RoboCup 2019: Robot World Cup XXIII. Springer (2019), accepted
4. Chitta, S., Sucan, I., Cousins, S.: Moveit![ros topics]. IEEE Robotics & Automation Magazine **19**(1), 18–19 (2012)
5. Crowe, J., Chen, G., Ferdous, R., Greenwood, D., Grimble, M., Huang, H., Jeng, J., Johnson, M.A., Katebi, M., Kwong, S., et al.: PID control: new identification and design methods. Springer (2005)
6. Dudek, G., Jenkin, M.: Computational principles of mobile robotics. Cambridge university press (2010)
7. Kajita, S., Hirukawa, H., Harada, K., Yokoi, K.: Introduction to humanoid robotics, vol. 101. Springer (2014)
8. Lopez-Mobilia, A.: Inverse kinematics kicking in the humanoid robocup simulation league (2012)
9. Mitchell, R.J.: Managing complexity in software engineering. No. 17, IET (1990)
10. Pena, P., Visser, U.: Adaptive walk-kick on a bipedal robot (2019)
11. Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: ROS: an open-source Robot Operating System. In: ICRA workshop on open source software. vol. 3, p. 5. Kobe, Japan (2009)
12. RoboCup Federation: A brief history of robocup, `https://www.robocup.org/a_brief_history_of_robocup`
13. Ruppel, P.: Performance optimization and implementation of evolutionary inverse kinematics in ros (2017), `https://tams.informatik.uni-hamburg.de/publications/2017/MSc_Philipp_Ruppel.pdf`
14. Starke, S., Hendrich, N., Magg, S., Zhang, J.: An efficient hybridization of genetic algorithms and particle swarm optimization for inverse kinematics. In: 2016 IEEE International Conference on Robotics and Biomimetics (ROBIO). pp. 1782–1789 (Dec 2016). https://doi.org/10.1109/ROBIO.2016.7866587