university of groningen / faculty of science and engineering

Exploring Physics-Based Human Musculoskeletal Control using Deep Reinforcement Learning

Bachelor's Project Thesis

Robin Kock, s3670600, r.kock@student.rug.nl
Supervisors: V. Raveendranathan & Prof Dr R. Carloni

Abstract: This paper presents a Python library which provides reinforcement learning environments for bipedal musculoskeletal control. The environments can be configured with different rewards, observations and actuator controllers, while enabling easy cooperation between researchers. To demonstrate the library, three different rewards and three different observation spaces are tested. Our results suggest that the best performing observation space includes local body part positions but not the imitation data. Additionally, our best performing reward includes both a goal reward (reward for moving forward) as well as velocity-based and positional imitation reward.

1 Introduction

Computer simulations have been used extensively to create novel control algorithms (He et al., 2021; Adam et al., 2012). However, human-in-the-loop control is still very challenging (Mayag et al., 2022), since it is hard to simulate a human alongside the controlled mechanics. One possible way to get around this limitation is to simulate a human using musculoskeletal simulations. These simulations produce plausible human movements, but still do not account for the behavior of the operator. This thesis explores different simulation environments used for human-in-the-loop control. Paving the way to control a prosthesis along with the operator in future research.

The simulated environments consists out of a musculoskeletal character with or without a prosthesis and some supporting geometry. The character should perform different tasks, such as walking, standing, sitting, going up or down ramps, ascending or descending stairs, or traversing rougher terrain. To achieve this it is necessary to control the muscles of the character as well as any actuators used for the prosthesis. Once a suitable controller is found further work is then needed to control the mechanical actuators independently from the character. Controlling up to 22 muscles and up to two actuators is no easy feat and so a sophisticated control algorithm is needed.

There are many different control architectures for musculoskeletal models and prostheses, ranging from hand written controllers (Huff et al., 2012), to evolutionary algorithms (Davis et al., 2014). Reinforcement learning is a class of algorithms which create a policy (controller) through interaction with an environment. Policies can be many different models, but using a neural network which is updated using gradient decent is often chosen for complex controllers. This gives rise to deep reinforcement learning (DRL). There are many different DRL algorithms of which I will be using proximal policy optimization with covariance matrix adaption (PPO-CMA) (Hämäläinen et al., 2020). This algorithm is based upon proximal policy optimization (Schulman et al., 2017) and covariance matrix adaption, where the former is a different DRL algorithm and the latter is an evolutionary algorithm. Other usages of CMA include trajectory optimization (Babadi et al., 2018), which can be computationally expensive if a general controller is required. PPO-CMA has been shown to work well with prosthesis simulations (Surana, 2022).

To learn a good controller an accurate simulation environment is needed. Such an environment must perform the physical simulations, provide the observation state for the policy, take the policy outputs to activate the actuators/muscles, and give feedback to the model using a reward signal. The main contribution of this project is in creating a library which can be used to configure and extend

such an environment. The physical simulations are done using OpenSim (Delp et al., 2007), and the library is written in Python (Van Rossum & Drake, 2009). The library is constructed to facilitate easy cooperation between different researchers all working on similar environments, and easily integrates with different DRL algorithms, through common interfaces.

In addition to explaining the rational behind the composable environment, this thesis also compares multiple different environments. These differ in the structure of the reward, as well as the observation state provided to the policy. Three different reward conditions and three different observation conditions are compared.

Section 2 gives a quick overview over the theory and explains the terminology used. Then, Section 3 describes the library and other algorithms used as well as the different experimental conditions used to test the library. Next, Section 4 shows the results of the experiments. Finally, Sections 5 & 6 discuss the results and describe some possible future work.

2 Theoretical background

This section describes the theoretical background necessary to understand the methods and results sections. First, reinforcement learning is reintroduced, to explain the terminology used in this paper. Then, a quick description of the PPO-CMA algorithm is given, along with a comparison to PPO and a few implementation details used. Finally, I explain the fundamentals of the OpenSim simulation software and how it is used to build a reinforcement learning environment.

Reinforcement learning The goal of reinforcement learning is to learn a policy through interaction with an environment. The policy π is a probability distribution over actions $\mathbf{a}$ depending on some parameters θ and the current state $\mathbf{s}_t$.

$$\mathbf{a}_t \sim \pi_\theta(\mathbf{s}_t) \tag{2.1}$$

For continuous action control, which is necessary to control the actuators of a prostheses, the policy is generally a multivariate gaussian distribution.

Using the policy and the environment, trajectories can be sampled. The environment provides an initial state, which is then used to sample an action from the policy. This action is then used to perform a step in the environment, which produces a new observation. These steps are repeated until the environment reports that a terminal state is reached, leading to a trajectory τ.

$$\tau = (\mathbf{s}_0, \mathbf{a}_0, \mathbf{s}_1, \mathbf{a}_1, ...) \tag{2.2}$$

To learn from the environment, a reward r is provided for every time step t. The reward may depend on the current state $\mathbf{s}_t$, the action $\mathbf{a}_t$, and the next state $\mathbf{s}_{t+1}$. However, for the environment presented in this paper the reward only depends on the current state. It is computed by the reward function R. Note, in our case, the reward function also depends on the imitation data Θ_I, although the subscript will be omitted for brevity.

$$r_t = R_{\Theta_\mathrm{I}}(\mathbf{s}_t) = R(\mathbf{s}_t) \tag{2.3}$$

Using the reward, the return can be computed. Which is defined over the entire trajectory, using a discount factor γ to ensure convergence.

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t R(\mathbf{s}_t) \tag{2.4}$$

For DRL, besides the policy there is also a learned approximation of the value function $\hat{V}^\pi(\mathbf{s})$. The value function V^π, which is approximated by $\hat{V}^\pi$, is the expected return from the current state under the current policy. It is defined as follows:

$$V^\pi(\mathbf{s}) = \mathop{\mathrm{E}}_{\tau\sim\pi}[R(\tau) \mid \mathbf{s}_0 = \mathbf{s}] \tag{2.5}$$

The learned value function often shares some parameters θ with the policy, to improve sample efficiency.

Note, vectors are being typeset in boldface.

PPO-CMA Hämäläinen et al. (2020) based their PPO-CMA algorithm on PPO (Schulman et al., 2017). However, they do not use the clipped surrogate loss. Hence, it is not necessary to fully understand PPO to grasp PPO-CMA, so only an overview over general policy gradient learning is given. After the description of PPO-CMA a small comparison with PPO is given. First, to learn from the experience gathered in the trajectories, the policy needs some means to estimate the quality of any action $\mathbf{a}_t$.

For this, the generalized advantage estimate A^π (Schulman et al., 2015) is used. The estimate is computed using the value function approximation $\hat{V}_\theta^\pi$ and intuitively represents the difference between the value of the current state and the value of the state after taking action $\mathbf{a}_t$. Next, we split up the policy into two policies, producing two vectors $\mu^\pi(\mathbf{s})$ and $\mathbf{c}^\pi(\mathbf{s})$, the mean and diagonal covariance of the multivariate gaussian respectively. Leading us to the policy loss below:

$$\mathcal{L}_\theta = \frac{1}{M}\sum_{i=0}^{M} A_i^\pi \sum_j \left\langle \frac{(\mathbf{a}_{i,j} - \mu_j^\pi(\mathbf{s}_i))^2}{\mathbf{c}_j^\pi(\mathbf{s}_i)} - \frac{1}{2}\log \mathbf{c}_j^\pi(\mathbf{s}_i) \right\rangle \tag{2.6}$$

Here, M is the size of a minibatch, each minibatch may contain multiple trajectories. The index variable i refers to a specific timestep within a specific trajectory within the minibatch, while j refers to a component of the action vector. Note, the first part in the angle brackets is a mean square error like term, updating the mean of the action, while the second part acts to increase or decrease the diagonal covariance component.

PPO-CMA improves over the simple policy gradient method presented above using three key differences.

1. Negative advantages are clipped to zero, or mirrored and converted to positive advantages. The authors claim that without this the negative advantages tend to "push" the policy actions to far. Since the gradients will tend to drive the actions away from negative advantages they can overshoot the correct value. Unfortunately, setting negative advantages to zero discards about half of the gathered experience, so the authors propose to mirror the actions along their mean and negate the advantages. This might not always be completely correct, but it allows the policy to also learn from negative advantages.

2. The policy variance and means are trained separately. The variance is trained first, and the mean is trained after. Note, for this to work $\mu^\pi(\mathbf{s})$ and $\mathbf{c}^\pi(\mathbf{s})$ may not share any weights. Inspired by covariance matrix adaption, this results in the variance being elongated along the best exploration direction. Which is an improvement over PPO, since PPO often suffers from premature reduction of variances.

3. The variance is trained using a history buffer, which contains multiple trajectories from previous epochs. This makes PPO-CMA partially work off-policy, which significantly increases sample efficiency. Additionally, using the history buffer further improves the elongation of variances along the best exploration direction.

PPO uses a clipped surrogate loss function to to prevent large updates to the policy.

$$\mathcal{L}_\theta^{CLIP} = \frac{1}{M}\sum_{i=0}^{M}\left[\min\left(\frac{\pi(\mathbf{a}_i \mid \mathbf{s}_i)}{\pi_{old}(\mathbf{a}_i \mid \mathbf{s}_i)}A_i^\pi, \operatorname{clip}\left(\frac{\pi(\mathbf{a}_i \mid \mathbf{s}_i)}{\pi_{old}(\mathbf{a}_i \mid \mathbf{s}_i)}, 1-\epsilon, 1+\epsilon\right)A_i^\pi\right)\right] \tag{2.7}$$

The loss function clips the ratios of old vs new policy, to prevent the policy from changing too much during policy updates. PPO-CMA achieves a similar objective by mirroring/removing negative advantages. Note, there are large differences between PPO implementations, especially on how they handle variances. Some implementations tune the variance as hyper parameters and some learn it similar to how it is done in PPO-CMA, albeit without the separate variance phase. Next, some implementation details of PPO-CMA are explained.

The PPO-CMA implementation used for this thesis was adopted from Surana (2022). The algorithm is used as above, with an action repeat of two, so every action will be used for two time steps. Additionally, this implementation normalizes the observations to be roughly normally distributed. Which is performed using individual running averages for each observation space dimension.

To improve performance on modern system the experience trajectories are collected in parallel. Note, the system only runs in parallel while the experience is collected, the PPO-CMA updates are computed on a centralized worker. During the parallel sections, the parallel workers advance the environment and send the resulting observations to the central worker, the central worker evaluates the policy and then sends the actions back to the corresponding worker.

OpenSim To simulate the dynamics and interactions between muscles, skeleton, prosthetics and the supporting geometry, the OpenSim framework is used (Delp et al., 2007). OpenSim has been used for similar simulations, with good results (van der Krogt et al., 2012; Steele et al., 2010). Internally, OpenSim reduces the model to multiple differential equations (Sherman et al., 2011) which are then integrated using the Runge-Kutta-Merson Algorithm (Merson, 1952, as cited in Burgin, 1970). Many different models are supported by OpenSim, including simple rigid bodies connected by actuated joints, up to complex musculoskeletal simulations with hundreds of muscles.

The environment presented here works with musculoskeletal models with up to 17 degrees of freedom (DOF). Five in each leg, six for the position and orientation of the pelvis, and one for the lumbar extension. However, generally the lumbar extension and hip rotations are locked and can not be moved, resulting in 14 DOF. The exact position of all rigid bodies in the simulation can be reconstructed from the DOF. Hence, it is sufficient to store the DOF to replay the simulation. The vector containing all the DOF is called the joint state Θ. Additionally, the velocities of the joint positions are referred to as $\dot{\Theta}$ and as mentioned above the imitation data is referred to as Θ_{I}. Note, there are more variables within the simulation besides the joint state, since the muscles themselves are stateful. Thus, even though we can replay the simulation using the history of joint states, we can not reproduce muscle activations or joint torques.

3 Methods

In the methods section, first the environment is explained from an organizational and collaborative standpoint. Additionally, the three main components of each environment: observation, reward and action are explained. Finally, the different simulation conditions presented in the results will be given along with quick explanations.

3.1 Environment

Below, the environment library will be presented. The goal of the library is to facilitate easy cooperation between a team of researchers, reducing code duplication and enabling easy switching between different environments with different reinforcement learning algorithms. There are multiple standardized interfaces for reinforcement learning, the most commonly used interface is OpenAI's Gym (Brockman et al., 2016), but others like ACME (Hoffman et al., 2020) do exist. Our environment implements an interface similar to Gym and provides wrappers to work with multiple interfaces. The next paragraph gives a high level overview of how the library can be used.

There are two possible use cases for the library, the first being the usage of already existing components to create an environment and using that for DRL. The second use case is more advanced and requires creation of new source files to create a custom environment. The benefit of modifying files in the library and pushing them back to the central repository is that other researchers can take advantage of this new environment, by simply fetching the new code. Next, I explain the architecture of the library and how the components can be combined to create an environment.

In our library an environment is represented by the `Environment` class. To create such a class, four components are needed. The first is a configuration object, holding parameters like step size, and a path to the OpenSim model file. The second is an `Observer` object, which defines how the observation state is created from the actual state of the simulation. The third component is an `Evaluator` object, which defines the reward $R(\mathbf{s}_t)$ as well as whether a state is terminal. Finally, the fourth component is a `Controller` object, which defines how the output from the policy are used to actuate the muscles and actuated joints. The latter three components will be described in more details below. For an overview of the architecture, see Figure 3.1.

After the environment was created it can be used in multiple ways. It can either be wrapped using different wrapper classes to be used with Gym or ACME, or the corresponding methods can be called directly. To start a trajectory the `reset` method is used. It optionally accepts a reset pose Θ_r to start the model in a specific pose. The `reset` method returns an observation state $\mathbf{s}_0$, to be passed to the policy to compute the first action $\mathbf{a}_0$.

$$\mathbf{s}_0 \coloneqq e.\texttt{reset}_{\Theta_r}() \tag{3.1}$$

Where e is the environment.

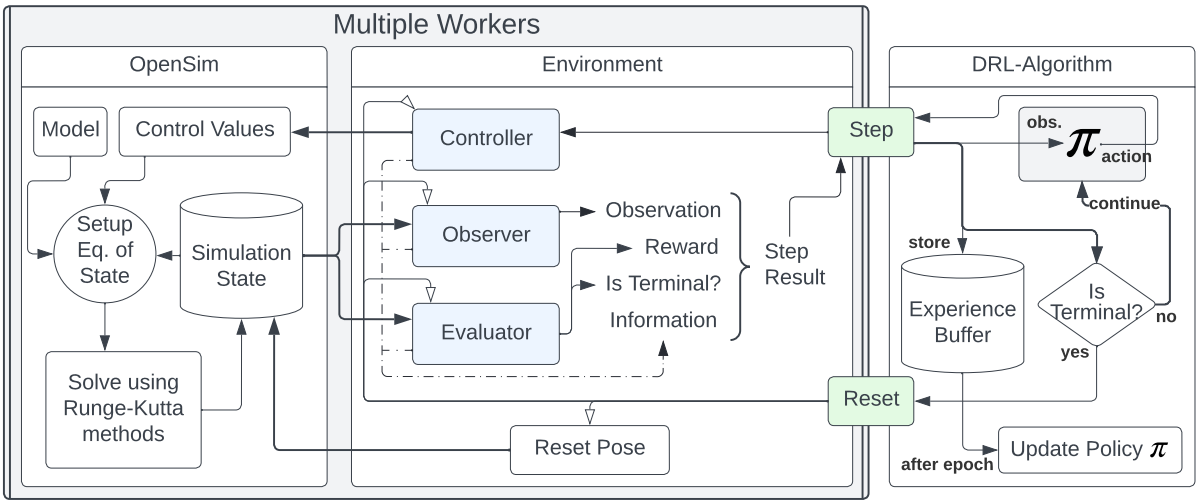


Figure 3.1: Software architecture of the environment and a simple DRL-algorithm. Blue: the three components `Observer`, `Evaluator` and `Controller`; green: the interface between the environment and the DRL-algorithm.

To advance the environment, the `step` method needs to be called, it requires an action and returns a tuple of four values:

$$(\mathbf{s}_{t+1}, r_{t+1}, \kappa_{t+1}, d_{t+1}) \coloneqq e.\texttt{step}(\mathbf{a}_t) \tag{3.2}$$

Where κ is a boolean value that is true when a terminal state is reached and d is a dictionary, containing user defined values. d can be used to extract values from different components within the environment.

In the following paragraphs the three main components making up each environment are explained. These are all implemented as abstract classes which need to be inherited by concrete implementation classes to be used with the environment.

Observation The observation states $\mathbf{s}$ are computed by an `Observer` class, as explained above. This is done by calling the `observation` method on the `Observer` which returns $\mathbf{s}$. The `Observer` has access to a `Context`, which contains references to all relevant objects of the environment. Using the `Context` the `Observer` may access the current joint positions Θ and joint velocities $\dot{\Theta}$; the position and orientation of all rigid bodies in the simulation; current simulation time and step size; interaction forces between colliders (like ground reaction forces); as well as muscle forces and joint torques. For the exact parameters used, see Section 3.2.

Each observation state $\mathbf{s}_i$ is assigned a unique name when the vector is generated. The names are completely ignored by the DRL algorithm, but can be recovered from the index i. Assigning a name to every observation value has a small run time performance impact, but it greatly improves the ability to debug observations. When specific entries fall outside the expected distribution or contain invalid values, the name can be looked up and used for debugging. Our library provides utility functions to quickly add different collections of values, like joint state vectors Θ or three dimensional vectors to the observation space. These utility functions automatically assign corresponding names to each vector component.

Reward The rewards r are computed by an `Evaluator` class. Additionally, the `Evaluator` class determines if a state is terminal. Hence, each evaluator must provide two methods. First, the `reward` method calculates the reward, using the same `Context` as the `Observer`, returning a number. Second, the `is_done` method computes whether the state is terminal, returning a boolean.

In most cases, to learn human like locomotion with reinforcement learning imitation data is re-

quired. For this purpose, a special `TrainingData` class has been created. This class is instantiated with a path to a character separated value (CSV) file, which contains values for all the DOF given in Section 2, OpenSim. A `TrainingData` object, provides methods to get $\Theta_{\mathrm{I},t}$ for any t within the range of the data. Additionally, the data may be shifted in time, slowed down or sped up.

Often times, it is easier to create a penalty instead of a reward. For example, the squared distance between components of Θ and Θ_{I}:

$$\rho_{\mathrm{I}} = \sum_{j \in \mathrm{DOF}} (\Theta_j - \Theta_{\mathrm{I},j})^2 \tag{3.3}$$

Where ρ_{I} is the imitation penalty and j iterates over all DOF or a subset of them. A similar penalty may be constructed for the velocities, called ρ_{Iv}. Note, we assume that a penalty is never negative.

There are many possible ways to turn such a penalty into a reward. In the following, two simple functions are proposed as well as a combination of them which allow for more precise tuning.

The first function R_{lin} turns the penalty into a reward using a linear relationship. It is necessary to clamp the reward at zero to avoid negative rewards, unless that is desired. This function has two parameters, a controlling the maximum possible reward, and b controlling the lowest penalty for which zero reward is given. a and b also control the intersection of the linear part with the y- and x-axis respectively.

$$R_{\mathrm{lin}}(\rho) = a \max\left(1 - \frac{\rho}{b}, 0\right) \tag{3.4}$$

The next function R_{exp} converts the penalty using an exponential decay. This function also has two parameters. First, a is the highest possible reward, when the penalty is zero. Second, b determines the decay rate, a large b will give more reward for high penalties. The exponential decay never reaches zero, hence some reward is always given no matter how large the penalty.

$$R_{\mathrm{exp}}(\rho) = a \exp\left(-\frac{\rho}{b}\right) \tag{3.5}$$

Note, $\exp(x) = e^x$.

Both of these functions have drawbacks, the linear function does not contain any information when $\rho > b$, so b needs to be sufficiently large. However, the derivative of R_{lin} for $\rho < b$ is $-\frac{a}{b}$, so changes in ρ produce small changes in R_{lin} when b is large. This is undesirable, since the policy learns through changes in the reward. The exponential function has a different drawback, which becomes evident when we take the derivative. $R'_{\mathrm{exp}} = -\frac{1}{b} R_{\mathrm{exp}}$ the derivative is also an exponential, so for small b the change in reward quickly goes towards zero for larger ρ. Additionally, for large b the derivative is small, since it is inversely proportional. Next, a different function is proposed which addresses both of these issues.

First, let us consider the derivative of an improved function, it should be constant initially, and then have a transition period until it is zero. There are many such functions and one of them is presented here: R_{log}. This function has three parameters. a scales the reward, b determines the slope before the transition period and c determines the size of the transition period.

$$R_{\mathrm{log}}(\rho) = \frac{c}{b} \log\left(1 + \exp\left(\frac{a}{c}(b - \rho)\right)\right) \tag{3.6}$$

When c approaches zero, the transition period also approaches zero and the R_{log} becomes R_{lin}, with a and b controlling intersection with the x- and y-axis. Increasing c, increases the size of the transition period which is approximately $\rho \in [b - c/\log(a), b + c/\log(a)]$. Note, the transition period is infinite, but the differences from the linear function outside the given range are negligible for most use cases. Finally, if $c/\log(a) > b$ then a does not represent the maximum possible reward, so care must be taken if the maximum reward is important.

Action In most DRL algorithms the output of the policy is constrained to be in the range $(-1, 1)$ or $(0, 1)$, although some algorithms do not constrain the activation. In any case, the value should be transformed to actuate either muscles or actuators. This might be a simple linear function followed by a clamping operation, but could also be more complicated if different types of control are desired. In our environment, this conversation is done by the `Controller`. To implement a `Controller`, the `action_to_control` method must be provided.

$$\psi_{t,j} \coloneqq \texttt{action\_to\_control}(\mathbf{a}_{t,j}, \mathcal{C}_j) \tag{3.7}$$

Where j is an index over the OpenSim actuators, $\psi_{t,j}$ is the control signal and $\mathcal{C}_j$ is a reference to the OpenSim actuator. Using $\mathcal{C}_j$ the method may access current motor torque and joint position as well as muscle forces and muscle fibre length. Note, OpenSim actuators may be muscles, or motor driven joints.

One possible way to customize a `Controller` for joint actuator control is using stable-proportional-derivative control (Jie Tan et al., 2011). It has been shown that this can improve the performance of DRL agents (Ma et al., 2021).

Each of the three classes `Observer`, `Evaluator` and `Controller` have an additional `reset` method, which can be implemented. This method gets called at the beginning of every trajectory, so that the three environment components can reset any internal state. Finally, some care must be taken when passing the three components to the environment constructor, as explained below.

The environment constructor does not expect instances of `Observer`, `Evaluator` and `Controller`, but rather a function which takes a `Context` and returns an instance. Often, the constructor of a component implementation is such a function. However, if more arguments need to be passed to the constructor a different creation function is required. The simplest way is to create an inline function which takes a context and returns the desired instance, in python this is done using the `lambda` keyword.

3.2 Experimental Conditions

In this thesis I compare five different combinations of reward and observation components. Three different rewards and three different observations. The different rewards and observations have the same hyper parameters, however in the different conditions some summands are omitted from the reward and some components are omitted from the observation vector. In the following paragraphs the full reward and observation are explained, afterwards the exact conditions are given.

Reward The full reward consist out of four summands. Positional joint imitation reward, joint imitation velocity reward, pelvis position reward, and pelvis velocity reward. The two joint imitation rewards are computed from the penalties ρ_{I} and ρ_{Iv}, see Equation 3.3. Note, the penalties do not include pelvis position or lumbar extension. Next, $R_{\log}$ is used to convert the penalties to reward, for the values of a, b and c, see Table 3.1. The hyper parameters given in the table were found by starting with parameters producing similar penalty functions to the ones used by De Vree & Carloni (2021). Then the parameters were updated manually to increase the variation of reward summands as seen in Figure 4.4.

reward part	**parameters**		
imitation pos.	$a = 0.6$	$b = 3$	$c = 0.4$
imitation vel.	$a = 0.25$	$b = 10$	$c = 2$
pelvis vel.	$a = 0.3$	$b = 0.15$	$c = 0.02$
pelvis pos.	$a_p = 10$		

Table 3.1: Reward hyper parameters

The pelvis velocity reward is also computed using a penalty ρ_{v} and the $R_{\log}$ function. The penalty is computed from the pelvis velocity $\mathbf{v}_{\mathrm{p}}$ and the target pelvis velocity $\mathbf{v}_{\mathrm{I,p}}$:

$$\rho_{\mathrm{v}} = |\mathbf{v}_{\mathrm{p}} - \mathbf{v}_{\mathrm{I,p}}| \tag{3.8}$$

Finally, the pelvis position reward is the simplest of the four, consisting out of the distance traversed since the previous step multiplied by a constant scalar a_p. The selected reward components are added to form the preliminary reward which is scaled by a factor to form the final reward. The factor is one when the pelvis is above a threshold, and goes towards zero the further the pelvis is below the threshold.

$$r_t = \sum_k r_{k,t} \begin{cases} 1 & \mathbf{p}_{p,y,t} > 0.8 \\ \frac{1}{1+(10\mathbf{p}_{p,y,t}-8)^2} & \text{otherwise} \end{cases} \tag{3.9}$$

Where k iterates over the different reward summands and $\mathbf{p}_{p,y,t}$ is the height of the pelvis at the current time step. The trajectory ends, once $\mathbf{p}_{p,y,t} < 0.6$.

Observation The observation produces a single vector, the length depends on what we choose to include. The smallest observation is of length 32. This includes the pelvis position (3) and orientation (3) as well as the linear and angular velocity of

the pelvis (6); then we include a subset of the joint state (10), excluding the pelvis and the lumbar extension; finally we also include the velocity of the aforementioned joint state (10). For the pelvis orientation an euler axis representation is used, great care was taken to ensure that the values stay continuous and do not approach singularities when the model is walking.

Additionally, we can include the position and linear velocity of different body parts. There are 12 applicable body parts, each contributing 6 values, leading to 72 additional observation state components. Finally, we may also include the imitation data in the observation state. Even though the environment does not depend on the imitation data, it has been shown that including the data can improve model performance (Ma et al., 2021). For a table with all states, see Table A.1.

Note, the observation space is not scaled. This poses a problem for simple DRL algorithms which expect roughly equally distributed observations. However, our implementation of PPO-CMA computes a running average of each observation component and normalizes them, making prior normalization redundant.

Conditions As mentioned above, five different experiments are run:

- *base*: simple observation space (i.e. 32 components), with full reward (all 4 summands and low pelvis penalty);
- *no-vel-rew*: simple observation space, with reward which does not include the imitation data velocity summand;
- *no-goal-rew*: simple observation space, with reward which does not include the pelvis position or velocity reward, nor the low pelvis penalty;
- *body-part-obs*: observation includes body part positions and velocities (i.e. 104 components), with full reward; and finally
- *imi-data-obs*: observation includes imitation data (i.e. 52 components), with full reward.

Each condition is trained for 200 epochs using the PPO-CMA algorithm. For the hyper parameters, see Table 3.2 The hyper parameters were not changed between runs, to keep the results comparable.

name	value
worker count	20
discount factor γ	0.99
adv. estimation λ	0.95
steps per epoch	20000
time per step	0.01s
MLP	
hidden layer count	2
layer neuron count	256
layer activation	leaky ReLU
learning rate	0.001
PPO-CMA	
mirroring	yes
hist. buf. length	3
batch size	2048
batches per epoch	128

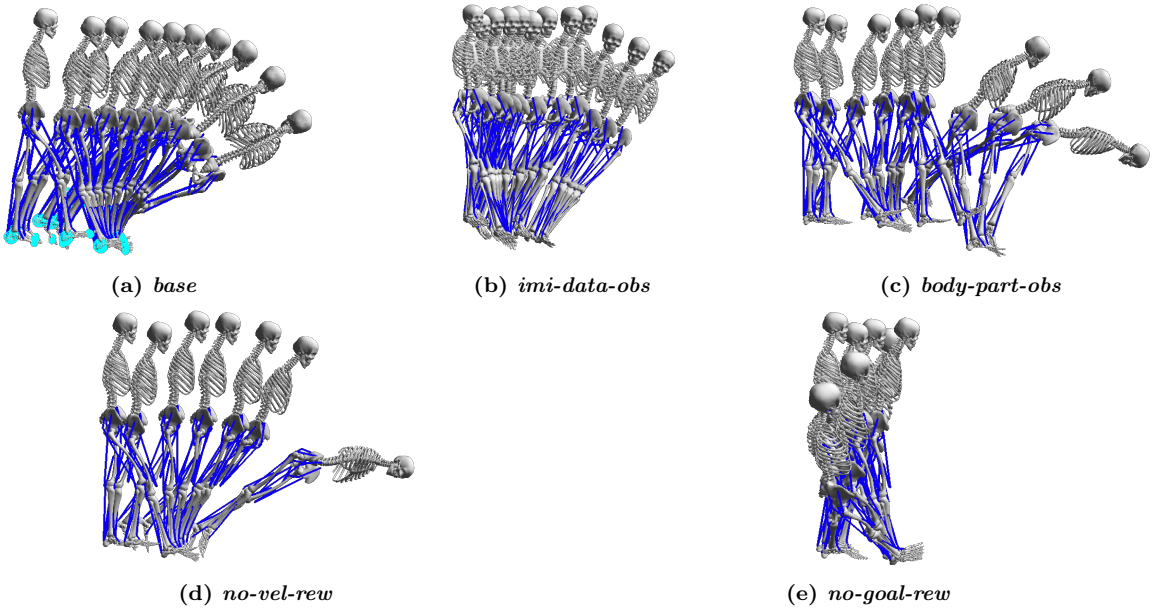
Table 3.2: PPO-CMA hyper parameters

4 Results

In this section the results of the five simulations given above are presented. First, an overview over the behavior of the trained models is given, then I will compare the performance of different runs using multiple metrics.

From the illustrations of trajectories at epoch 200 in Figure 4.1, it can be noted that the best performing condition is *body-part-obs*, here the musculoskeletal model learned to take three steps before falling forward. Next, the *base* learned to take one step, but then twists and falls. This is similar to the *no-vel-rew* condition, which also takes one step and then falls forward. The last two conditions (*imi-data-obs* & *no-goal-rew*) also take a first step forward, but they do not attempt the next step. Rather, the models just keep standing until they fall over. Without trying to take a next step, these two models learn to avoid falling for the longest duration.

Next, the duration each agent survives is investigated. This varies a lot from trajectory to trajectory, so the values presented here have been averaged and their standard deviation computed, see Figure 4.2. Note, the longer trajectories take, the fewer trajectories there are in each epoch, since the number os steps is being kept constant. As mentioned above, the conditions *imi-data-obs* and *no-goal-rew* achieve the highest step counts, however



(a) *base* (b) *imi-data-obs* (c) *body-part-obs*

(d) *no-vel-rew* (e) *no-goal-rew*

Figure 4.1: Each sub-figure contains a representative trajectory at epoch 200 for the five conditions. The positions were sampled at an interval of 0.2 seconds.

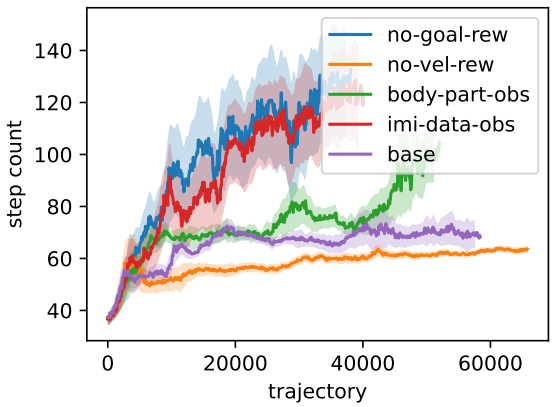


Figure 4.2: Step count per trajectory for the five different runs. The counts have been averaged over every 100 trajectories with standard deviation given by the highlighted area.

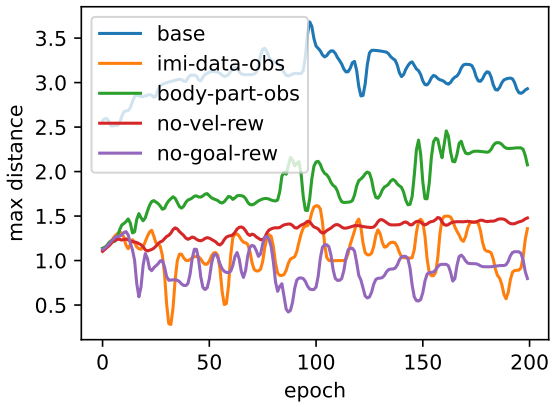
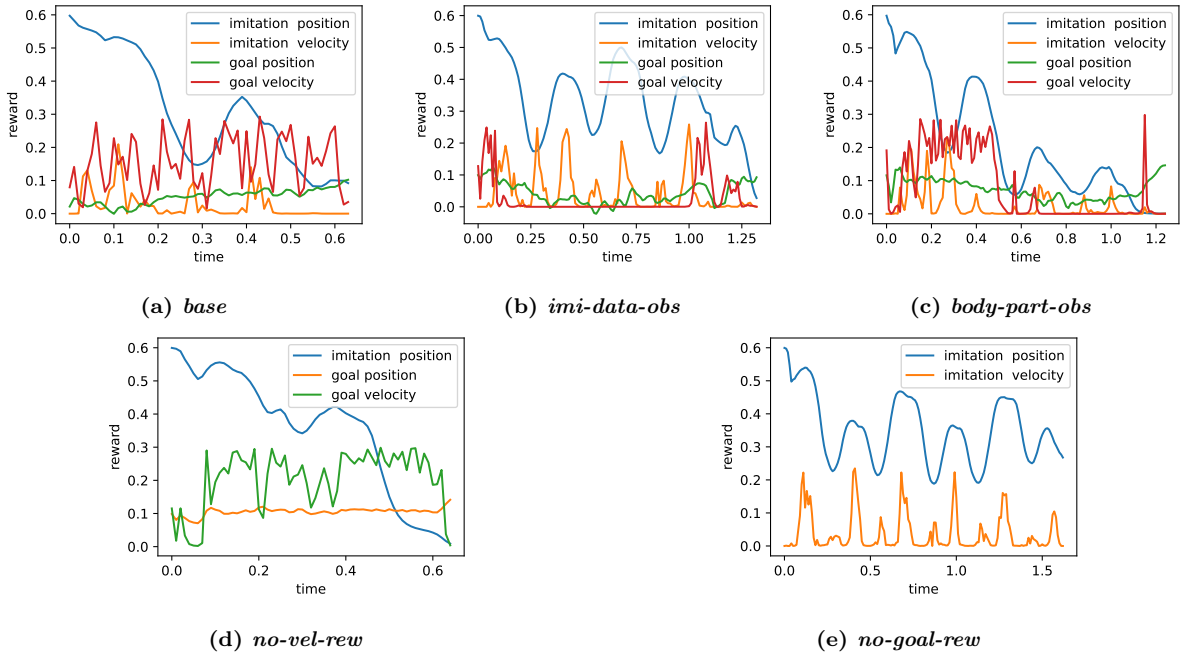


Figure 4.3: The final pelvis x-position averaged over trajectories per epoch.

they also have the largest variance in step count, so although some trajectories lead to the model balancing for a while, often it also falls over.

The *no-goal-rew* condition has the highest step count, this is inline with our expectation, since the goal reward is the largest incentive pushing the model forward. Without the goal reward, the model



(a) ***base*** **(b)** ***imi-data-obs*** **(c)** ***body-part-obs***

(d) ***no-vel-rew*** **(e)** ***no-goal-rew***

Figure 4.4: Each sub-figure contains the reward summands for a representative trajectory at epoch 200.

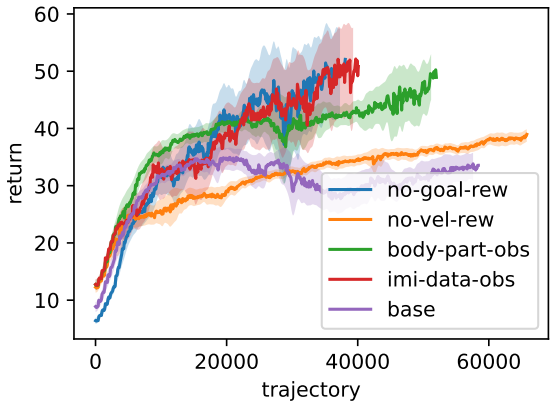


Figure 4.5: Returns per trajectory for the five different runs. The returns have been averaged over every 100 trajectories with standard deviation given by the highlighted area.

learns to stand instead of walking forward. Interestingly, removing the velocity imitation reward, reduces the step count and variance in step count, compared to the *base* condition.

The next metric presented here is the distance the model traverses, more specifically the distance from the origin to the pelvis, right before the trajectory ends. The distances are presented in Figure 4.3. These distances fluctuates quite a lot for the *imi-data-obs* and *no-goal-rew* conditions, this is likely due to the model falling in different directions in different epochs. The other three conditions walk forward, so their distances are more consistent, since they almost always fall forward. The largest distance is achieved by the *body-part-obs*, which is inline with our visual inspection.

The returns are plotted in Figure 4.5. Note, it does not make sense to compare the rewards between *base*, *no-goal-rew* and *no-vel-rew* directly, since the reward function changes between those conditions. Hence, we first look at the conditions where the reward function stays constant (*base*, *imi-data-obs* & *body-part-obs*).

Out of the three, *imi-data-obs* achieves the high-

est return. However, this is a local maxima, because the model does not attempt to walk and so the returns are only high because the model learned to survive for a long time. The *body-part-obs* condition does not survive as long, but after 200 epochs almost has the same return as the standing model. Finally, the *base* model performs the worst, which indicates that this model is too simplistic and a larger observation space is required for learning a good policy.

In Figure 4.4 the individual reward summands are plotted for one trajectory at epoch 200 for each condition. The periodic change in reward visible in Figure 4.4e, comes from the training data playing, as the model is standing. Whenever the training data is in a similar position to the standing model, the positional imitation reward goes up. Likewise, whenever the imitation velocities are similar the corresponding reward increases. A similar effect can be observed in the other partial reward plots.

5 Discussion

Even though no model in the conditions managed to walk more than a few steps, there are still clear differences in performance. First, both the velocity reward and the goal reward improve model performance, this is inline with our expectations, since a reward with all those summands is similar to th reward used by De Vree & Carloni (2021).

The *base* observation space is likely too simple, the model learns quick initially, but after about 50 epochs a limit seams to be reached and all the performance measures seem to stagnate. Likewise, including the imitation data does not improve the performance of the model. One possible reason for the model not learning to walk when including the imitation data is the noise introduced by the additional observation space components which do not correspond to anything physical in the simulation.

Finally, the best performing model used the full reward and included the local position and velocities of all body parts. This finding is inline with current research on DRL and bipedal locomotion (Brockman et al. (2016), see `Humanoid v1`).

Hyper parameters In the following paragraphs, the method used to tune the hyper parameters is described. The hyper parameters for PPO-CMA were initially taken from Surana (2022) and were then tuned using a hyper parameter grid search. The search included: the learning rate, the number of units per layer for the policy, the amount of steps per epoch, the batch size, and the number of training batches per epoch. After tuning these hyper parameters, the history buffer length has been changed from nine to three. This change did not alter the performance of a test model significantly, but decreased the memory usage and runtime per epoch. After training the PPO-CMA hyper parameters, the reward function parameters need to be tuned.

Initially, the hyper parameters were set to values which result in similar reward functions as used by Surana (2022) and De Vree & Carloni (2021). Then over the course of multiple simulations, the parameters were adjusted to maximize the variance of each summand, while minimizing the duration where the summand is zero. This process was done manually, but could be automated in the future. The reason for maximizing reward and minimizing zero-duration is to increase the information contained in the reward. When the reward is zero, then a small change in penalty will not change the reward. Otherwise, we want to maximize variance, so the information contained in the reward is as clear as possible. Note, the scale (a in $R_{\log}$) of each summand was taken from De Vree & Carloni (2021) and not changed.

Future research As mentioned in the introduction, the ultimate goal of this research is to develop an actuated transfemoral prosthesis. Our library can be used to simulate transfemoral amputee models, just as it can be used for able-bodied models. However, different `Observers` and `Controllers` are needed to interface with the different model.

To improve training results and iteration times, it might make sense to constrain the models movement to the x/y-plane in future research, as it is done by De Vree & Carloni (2021). With this change it should be easier to train different models, and so hyper parameters could be tuned with higher precision.

Finally, it might make sense to train a "narrow" model. This model would be stopped as soon as it deviates too much from the training data. Ma et al. (2021) shows that this can improve perfor-

mance. However, the model would only train from joint states very close to the training data. Thus, it might not be able to recover from unseen joint states and the training data needs to be of high quality.

6 Conclusions

This research developed a python library to simulate musculoskeletal models and learn control algorithms using DRL, more specifically PPO-CMA. Using the library multiple different reward functions and observations spaces were tested. The best performing observation space contained joint states, body part positions and their respective velocities. The best reward function contained joint state imitation reward, joint state velocity imitation reward, pelvis position reward and pelvis velocity reward. Removing the pelvis rewards, or the joint velocity reward decreased model performance. Finally, adding the imitation data joint states to the observation state decreased model performance.

The library we developed is ready to be used with different models and different DRL algorithms. The next step is to build up a repertoire of different combinations of DRL algorithms, reward functions (`Evaluator`), observation spaces (`Observer`) and models.

7 Acknowledgements

The author would like to thank his supervisors prof. dr. R. Carloni and V. Raveendranathan. Additionally, the author wants to acknowledge the contributions made by other students to the library, especially M. Falzari who helped build the codebase and wrote a lot of documentation.

References

Adam, S., Busoniu, L., & Babuska, R. (2012). Experience replay for real-time reinforcement learning control. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, *42*(2), 201-212.

Babadi, A., Naderi, K., & Hämäläinen, P. (2018). Intelligent middle-level game control. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)* (p. 1-8).

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). *OpenAi Gym.*

Burgin, G. H. (1970). Comment on the runge-kutta-merson algorithm. *SIMULATION*, *15*(2), 89-91.

Davis, R., Richter, H., Simon, D., & van den Bogert, A. (2014). Evolutionary optimization of ground reaction force for a prosthetic leg testing robot. In *2014 american control conference* (p. 4081-4086).

Delp, S. L., Anderson, F. C., Arnold, A. S., Loan, P., Habib, A., John, C. T., ... Thelen, D. G. (2007). OpenSim: Open-source software to create and analyze dynamic simulations of movement. *IEEE Transactions on Biomedical Engineering*, *54*(11), 1940-1950.

De Vree, L., & Carloni, R. (2021). Deep reinforcement learning for physics-based musculoskeletal simulations of healthy subjects and transfemoral prostheses' users during normal walking. *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, *29*, 607–618.

He, W., Gao, H., Zhou, C., Yang, C., & Li, Z. (2021). Reinforcement learning control of a flexible two-link manipulator: An experimental investigation. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, *51*(12), 7326-7336.

Hoffman, M., Shahriari, B., Aslanides, J., Barth-Maron, G., Behbahani, F., Norman, T., ... de Freitas, N. (2020). Acme: A research framework for distributed reinforcement learning. *arXiv preprint arXiv:2006.00979*.

Huff, A. M., Lawson, B. E., & Goldfarb, M. (2012). A running controller for a powered transfemoral prosthesis. In *2012 Annual International Conference of the IEEE Engineering in Medicine and Biology Society* (p. 4168-4171).

Hämäläinen, P., Babadi, A., Ma, X., & Lehtinen, J. (2020). PPO-CMA: Proximal policy optimization with covariance matrix adaptation. In *2020 IEEE 30th International Workshop on Machine Learning for Signal Processing (MLSP)* (p. 1-6).

Jie Tan, Liu, K., & Turk, G. (2011). Stable proportional-derivative controllers. *IEEE Computer Graphics and Applications*, *31*(4), 34–44.

Ma, L.-K., Yang, Z., Tong, X., Guo, B., & Yin, K. (2021). *Learning and exploring motor skills with spacetime bounds.* arXiv. (Number: arXiv:2103.16807)

Mayag, L. J. A., Múnera, M., & Cifuentes, C. A. (2022). Human-in-the-loop control for agora unilateral lower-limb exoskeleton. *Journal of Intelligent & Robotic Systems*, *104*(1), 1–19.

Merson, R. H. (1952). An operational method for the study of integration preocesses. *SIMULATION*.

Schulman, J., Moritz, P., Levine, S., Jordan, M., & Abbeel, P. (2015). High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Sherman, M. A., Seth, A., & Delp, S. L. (2011). Simbody: multibody dynamics for biomedical research. *Procedia IUTAM*, *2*, 241–261.

Steele, K. M., Seth, A., Hicks, J. L., Schwartz, M. S., & Delp, S. L. (2010). Muscle contributions to support and progression during single-limb stance in crouch gait. *Journal of biomechanics*, *43*(11), 2099–2105.

Surana, S. (2022). *Evaluating deep reinforcement learning algorithms for physics-based musculoskeletal transfemoral model with a prosthetic leg performing ground-level walking.*

van der Krogt, M. M., Delp, S. L., & Schwartz, M. H. (2012). How robust is human gait to muscle weakness? *Gait & posture*, *36*(1), 113–119.

Van Rossum, G., & Drake, F. L. (2009). *Python 3 reference manual*. Scotts Valley, CA: CreateSpace.

A Observation Space

state name	#	exact components
pelvis position	3	x, y, z
pelvis orientation	3	pitch, yaw, roll
linear velocity of the pelvis	3	x, y, z
angular velocity of the pelvis	3	pitch, yaw, roll
10 joint states:	20	ankle, knee, hip-flexion, -abduction, -rotation for left and right side each.
position	1	
velocity	1	
Only *body-part-obs*		
12 body parts:	72	left/right femur, tibia, talus, calcn, toes respectively as well as torso, head
position	3	
linear velocity	3	
Only *imi-data-obs*		
10 imitation joint states	20	ankle, knee, hip-flexion, -abduction, and -rotation for left and right side each.
position	1	
velocity	1	

Table A.1: Observation space, the left column describes a group of states, the center column gives the number of states in the group, while the right column gives the components/states/body-parts forming the group. Note, indented states exist once for every state/body-part listed on the right.