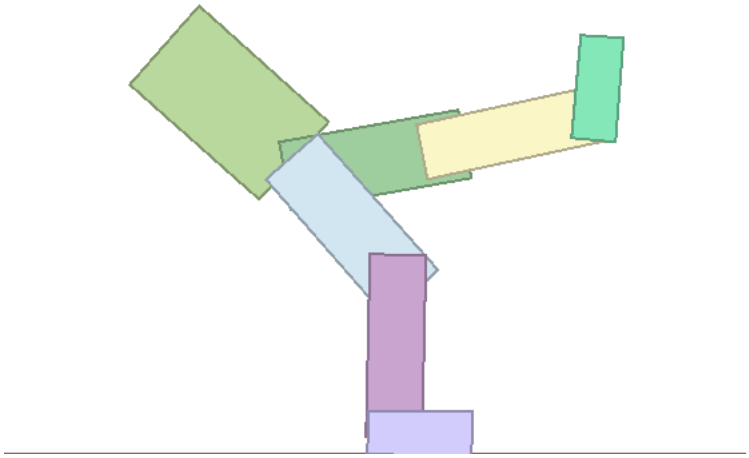
Q-Learning for a Bipedal Walker

Finding complexity in simple actions



Roskilde University, Spring 2015
Computer Science Semester Project
Supervisor: Ole Torp Lassen

Frederik Tollund Juutilainen, 52474
Sara Almeida Santos Daugbjerg, 52175
Younes Haroun Bakhti, 55485

Abstract

This project studies how a bipedal body can learn forward movement, within a simulated 2D physics environment through the reinforcement learning method Q-Learning. Q-Learning is a model-free form of reinforcement learning, which can be used to find the optimal policy for solving a Markov Decision Process. Through experimentation and parametrisation, it was possible to create several types of test-cases that could be used for further testing the extent of the learning agents abilities, in order to analyse the results and optimise the designed software. We could conclude, from the findings, that the learning algorithm was unable to find a stable pattern of state-actions, which resulted in forward movement. This could be due to the high complexity and large number of states, which restricted performance of the designed software. However, in the less complex cases, the agent demonstrated that, even in a short amount of time, it could find a stable position that would yield a high enough reward to be regarded as successful and was thereby succesful in learning from experience, albeit at a smaller scale.

Contents

Chapter 1

Intro

"I'm sorry, Dave. I'm afraid I can't do that."

Hal

a Introduction

The idea of self-aware machine is often depicted as a venture, not without a great deal of risk. Namely, in popular media such as books, films and games, the machine often reaches a point of such potent intellect, that it outmatches that of its creators, often with fatal consequences to the latter. This mostly occurs in scenarios where the machine is put in control of protecting assets from potential threats, and eventually deems the creators themselves as threats.

Within the field of study that is machine learning, the notion of an artificial intelligence that can learn from experience has peaked the interest of people all over the world, despite the potential risks involved. Generally speaking, the concept of having an AI teach itself is fascinating and holds practical purposes. Traditionally, it can be assumed that the designer would have a complete idea of the desired behaviour. In reinforcement learning one somehow steps away from this notion and instead leaves it to the AI itself to find optimal ways of solving problems. This approach to machine learning has captivated our attention and inspired us to study the subject further, in relation to our project.

b Inspiration for the project

The "*genetic walker*" simulation found on the *Rednuht* website[1] has been a major source of inspiration for this project. This website features a somple, yet intriguing, machine learning algorithm, where a bipedal walker slowly learns how to walk and becomes increasingly better for each generation. This is not presented as a "state-of-the-art"-approach, but it still managed to inspire us to explore this field of study.

[1] Genetic Algorithm Walkers

This particular example serves as the core inspiration of our project as it manages to demonstrate the area of attention for our project perfectly: Creating and simulating a body and providing it with a mind of its own, so to speak, through a learning agent that can take actions based on previous experience and thereby evolve into an artificial intelligence - with the potential of eventually reaching a point where it is able to accomplish, within a certain framework, whatever goal it might have.

c Research question

The above has lead to the following research for this project:

- How can a bipedal body learn forward movement, within a simulated 2D physics environment, using a Q-learning algorithm?

d Project Requirements

In the following section we will describe the requirements for a Computer Science project at Roskilde University. We will elaborate on how this project relates to these requirements.

Knowledge about software development, including programming, algorithms and data structures.

This project will include software development of a relative high complexity, since it includes use of external libraries, object-oriented programming and a need for a certain amount of modularity in order to deal with new requirements discovered during the research process. In addition to this the project will involve the use of primitive and more complex data structures as well as implementing objects designed from scratch.

Skills in programming, testing and documenting a program in a higher, general programming language.

This project is written in Java, which is a high-level programming language. Since the general research question posed has to be explored using software, this specific software has to be programmed and will be documented extensively. In addition to this, the project will implement third-party classes which will be customised in order to fit the needs of the program.

Skills in choosing and arguing for the choice of design, data structures and algorithms for the specific project.

This project will feature chapters on design choices including more general choices concerning architecture as well as more specific choices involving optimisation of the code. The software in this project contains algorithms that are written from scratch as well as more complex algorithms from third-party sources, which will be explained and discussed.

Skills in specifying and modelling requirements for the functionality of information systems.

Each chapter concerning a specific part of the program features a section which, in-depth, describes the requirements for this part of the software. These requirements are in focus, when explaining code examples and implementation.

Competencies in planning, specifying standards for and leading a small software development process.

This entire development process will require planning and adaptability, since the entire scope of the software is not known before the actual programming begins. In addition to this the software isn't developed by a single person, which sets higher requirements for the team-work in which a high level of abstraction is needed.

Chapter 2

Design Choices

With regards to the research question, we sought out to delimit the overall design for the project and define the *requirements* for the simulation. This chapter concentrates mainly on the choice of tools for the program, such as the *programming language* and third-party *libraries*.

a Requirements

To accomplish our primary goal of simulating a body in a world which will ultimately allow for it to learn from an algorithm, there are several aspects of such a program that have to be considered. Below, these are reviewed along with the reasoning behind our selection of which *programming language* to utilize, what *functionality* should be included and, regarding the latter, a large amount of experimentation as well, as it will most likely necessitate the use of one or more *libraries*.

Language

When considering choosing the appropriate programming language for the project, it was decided early on that *Java*[1] was the language of choice and would serve as the overall software and computing platform. In addition to the fact that all members have pre-existing knowledge and experience with the language, Java is among the most popular and widely used programming languages for a significant variety of program types. It is accepted as an established platform for software development, including as the basis for both advanced programs, such as networking applications, as well as in the development of games.

Another illustration of why Java is fundamentally the right choice for our program, is the *WORA* mantra. This stands for "*Write once, run anywhere*" and exemplifies Java's focus on cross-platform compatibility, which is important when testing on multiple machines, both laptop and desktop and on the Windows, Mac and Linux operating systems.

There also exists a vast amount of documentation and resources on this particular language, in addition to a great number of available online communities that are able to assist with everything from general help with the language to unique program troubleshooting. This will certainly prove useful, as there are many advanced methods within

[1] Learn About Java Technology

the APIs that require further study before being applicable in our simulation. In terms of third party packages, Java is widely supported by software developers and numerous libraries that can extend or add new functionality to the platform and are made ready for the user all over the internet.

Functionality

When looking at the overall goal of the simulation, which is to simulate a body that can act on and receive feedback from its environment in order to learn and become increasingly better at performing optimally in a given situation, it is possible to break down the requirements, classify, analyse and, as a result of this, define what functionality should be included in the program.

From our inquiry on programming language, we learned that Java and its relating online communities provide enough documentation and external libraries to support our vision of the simulation. Although this will require some further study in order to confirm the right choice, it will be necessary to investigate the third-party library options available so that we may confirm what features are essential to the functionality of the program. Regarding Java's limitations as a development platform, it is the physical modelling aspects that need to be enhanced through these libraries. Basically, the core API of Java is cumbersome and not easily modifiable in this regard, which is a moderate hindrance as it would mean that these limitations would dictate how the simulation should run and could impede experimentation with the parameters of the simulation.

Libraries

The first physics engine library chosen for the simulation was jbox2D[2]. Although the feature set of jbox2D holds the functionality of a rigid body physics engine, and therefore suitable in our venture, the foundation of this library is rooted in another library called box2D. This library is written in C++ and as jbox2D is a port of box2D to Java, it was deemed more proficient to seek out a library intended for Java, insuring that any documentation and support is more readily available and directly related to the library.

In order to accomplish our goal of simulating a bipedal walker, that will output the actions deemed appropriate by the machine learning algorithm, we sought out to find a Java-compatible physics engine that could meet our requirements. We came across such a library called Dynamics for Java, or *dyn4j* for short.

Dyn4j is a 2D collision detection and rigid body physics engine, primarily aimed at game development. It is both stable and supports various platforms which should minimize the amount of cross-platform compatibility issues that may arise, namely between the Mac and Windows operative systems[3].

Finally, in accordance to the research area for the project and its related subject, namely artificial intelligence, there was a need to acquire third party packages from the implementation examples[4] found in the additional documentation for the book *Artificial Intelligence A Modern Approach*, by Russell and Norvig from 1995. This book has, furthermore, served as the main source of theoretical understanding of this field of study.

[2]Daniel Murphy 2014
[3]dyn4j website
[4]AIMA Github

Chapter 3

Physics Modelling

This chapter will explain the main classes for physics modelling and rendering in the program. As mentioned, this was done using dyn4j[1], which is an open-source library for physics modelling and collision detection, mostly designed for game-design. The main classes at play here are *Simulation*, *Graphics2DRenderer* and *GameObject*, which all came from an example at the dyn4j-website. From these three classes *Graphics2DRenderer* and *GameObject* have not been altered. The functionality will be explained, while the main focus of this section will generally remain on how *Simulation* has been modified to fit our specific needs.

a Rendering

a.1 Requirements

Our requirements for the rendering are relatively simple. The rendering must be in 2D and go easy on the GPU. Aside from this, the rendering should, as much as possible, be in-sync with the simulation running. In addition to this we would like the rendering to use mostly Java-native methods and to have as limited reliance on third-party libraries as possible, primarily in order to avoid or at least limit the need for the time-consuming processes of learning how to implement these.

a.2 Implementation

Graphical rendering is, as mentioned above, handled mainly in *Graphics2DRenderer*. This class consists of a number of overloaded methods that can take different shapes as one of the input arguments. The main use of this class is to be able to draw shapes and this is done using the *Graphics2D* component which it is in the standard Java library. All the rendering methods in the *Graphics2DRenderer* are static methods and they are called from the GameObject class.

[1] dyn4j Website

GameObject

The GameObject class was also included the example at the dyn4j website. It is a sub-class of *Body*, which is a class in the *dyn4j*-library. *Body* is the object that simulates bodies that can collide, have mass and be manipulated by force and torque. Fixtures are added to a *Body*, but the core simulation in dyn4j is through the use of bodies. *GameObject*, as a sub-class of *Body*, has one method added to the *Body*-methods, which is *render*().

```
for (BodyFixture fixture : this.fixtures) {
Convex convex = fixture.getShape();
Graphics2DRenderer.render(g, convex, Simulation.SCALE, color);
}
```

In this method all *BodyFixtures*, which represent a part of the body, are iterated through and the static render method from Graphics2DRenderer is called. By using the *fixture.getShape*()-method a *Convex* is returned, which is then set as an argument in the overloaded *render*() method in the *Graphics2DRenderer*.

b Simulation

The environment, in which the simulation will take place, must first of all reflect an environment that is similar to our own physical reality. This is in order to ensure that when the bipedal walker is placed within this environment, our expectations of its actions are relatable to the simulated environment.

b.1 Requirements

One of the most important physics-aspects of the simulation is *gravity*. This means that there needs to be a functioning gravitational field simulated in the program. Within the library is a method, namely the *setGravityScale*() method, which can be used to control the gravity parameters of bodies in the world. This is not directly implemented in the code, as the method is active nonetheless and the default value is 1.0, which is similar to its real world equivalent.

Additionally, there must be a way for time to be simulated - since without the passage of time, gravity cannot have any effect on the simulation of the bipedal walker.

In order to confine the bipedal walker to an area within the environment, we must also create a surface that can serve as the *floor*. This will ensure that the walker does not fall out of bounds and that it has a surface to move on.

b.2 Implementation

The physics modelling is handled in the *Simulation*-class. Originally this was named *ExampleGraphics2D* in the dyn4j example, but has since been renamed in order to avoid confusion with the *Graphics2D*-component from the Java API. In the example provided, this class contained the main method for the program, but now acts as an object that

can be instantiated in other classes. We've only modified two methods, *GameLoop*() and *initializeWorld*(), in this class and these will be explained below.

initializeWorld()-method

This method is a private method, which is called once in the *Simulation*-constructor. It has the purpose of initialising the bodies that are to be used in the simulation, which can then be added to a *World*-object. A *World*-object is a dynamics engine provided by the dyn4j-library. We have changed it, so the World is a public static object, which will later prove to be useful elsewhere in the program. The constructor for the *World*-object is run in the *initializeWorld*()-method:

```
this.world = new World();
```

After the constructor for the *world*-object is run, we're then able to add bodies to this world:

```
floor = new GameObject();
BodyFixture floorFixture = new BodyFixture(Geometry.createRectangle(15.0, 1.0));
floorFixture.setFriction(1000.0);
floor.addFixture(floorFixture);
floor.setMass(Mass.Type.INFINITE);

floor.translate(0.0, -2.95);
this.world.addBody(floor);
```

The method-calls in dyn4j are mostly self-explanatory. It's worth noting that the *floor.setMass*() is set to *Mass.Type.INFINITE*,. This affects the mass of the floor, so it is not affected by gravity and forces in dyn4j, - thereby creating a static or a *dead* body. There's also a method for setting the friction of a body. Here *setFriction*() is set to 1000. This was done through tests and makes it less likely for our biped walker to slide around on the floor.

Walls

There are cases where we would like walls to be added to the world, but this only happens in appropriate test-cases. This is simply done by adding more GameObjects with *setMass*(*Mass.Type.INFINITE*); and adding these to the world. Finally the *BipedBody* is created and added to the world:

```
this.world.addBody(floor);

if(Main.mode == 1 || Main.mode == 2)
{
Rectangle wall1Rect = new Rectangle(1.0, 15.0);
GameObject wall1 = new GameObject();
BodyFixture wall1Fixture = new BodyFixture(Geometry.createRectangle(1.0, 15.0));
wall1.addFixture(wall1Fixture);
wall1.setMass(Mass.Type.INFINITE);
wall1.translate(-6,0);
```

```
GameObject wall2 = new GameObject();
BodyFixture wall2Fixture = new BodyFixture(Geometry.createRectangle(1.0, 15.0));
wall2.addFixture(wall2Fixture);
wall2.setMass(Mass.Type.INFINITE);
wall2.translate(6, 0);

world.addBody(wall1);
world.addBody(wall2);
}

walker = new BipedBody();
```

This *BipedBody* is not added to the world in the *initializeWorld*()-method, this is instead done in the *BipedBody*-constructor, which will be explained more detailed later.

gameLoop()-method

While the *initializeWorld*()-method is only run once, the *gameLoop*()-method is run as long as the physics simulation is running. This method calls the above-mentioned *render*()-method for all bodies in the *world*. This method has been left mostly unchanged, but we have made some changes in order to make it possible to change simulation speed. The main simulation is handled by running *world.update*(), where the input argument is elapsed time.

```
elapsedTime = elapsedTime*simulationSpeed;
```

In the above code *elapsedTime* is a double that indicates the difference in time since last simulation step. This value is based on *System.nanoTime*() and if left unchanged, it simply matches simulation speed to the speed of the time on the computer. We multiply this with *simluationSpeed* which is an integer that can be adjusted from the GUI and thereby make it possible to speed up the simulation.

```
synchronized (ThreadSync.lock) {
this.world.update(elapsedTime, Integer.MAX_VALUE);
}
```

Lastly the *world.update*()-method is called with the "updated" *elapsedTime*, which, as mentioned above, is dependent on a slider in the GUI. This update is forced to be synchronized to a *lock*-object in order to avoid threading issues. This will also be described more thoroughly later in this report.

c BipedBody

The design of the body, effectively the bipedal walker itself, will be derived from the functions of an actual real-life body.

c.1 Requirements

The bipedal walker will require a body composed of limbs. Furthermore, in order to hold these limbs together, the equivalent to joints must also be present. These shall serve the purpose of connecting the limbs together in such a manner that they mimic the physical limitations of bipedal bodies. In essence, this means that the joints need to impose angular and rotational limits to the connected limbs.

c.2 Implementation

In order to simulate the body for the walker, the library called Dyn4j is used. Therefore, the functionality derived from said library becomes essential for creating the walker.

BipedBody-**class**

The *"hero"* in this program is the *BipedBody*.

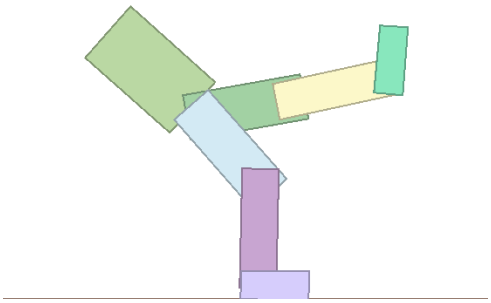


Figure 3.1: The *BipedBody* rendered

This class is basically used to contain the limbs and joints of the bipedal walker. All limbs are of type *GameObject*, which is explained above. This makes it possible to have body parts on the Biped that are manipulated individually and rendered by the methods in the *Graphics2DRenderer*-class.

In our simulation, these bodies can serve as limbs for the walker. A body can be assigned a shape that will define how it is displayed visually. The shape is then given a *BodyFixture*, which can provide more information on how the body performs, such as its mass.

Below is an example of the *torso* GameObject:

```
torso = new GameObject();
{
Convex c = Geometry.createRectangle(0.6, 1.0);
BodyFixture bf = new BodyFixture(c);
torso.addFixture(bf);
```

```
torso.translate(0, 0);
torso.setMass(Mass.Type.NORMAL);
}
world.addBody(torso);
```

This is done in the *BipedBody*-constructor a long with method-calls for all other body parts. Which are as follows:

- GameObject torso;
- GameObject upperLeg1;
- GameObject upperLeg2;
- GameObject lowerLeg1;
- GameObject lowerLeg2;
- GameObject foot1;
- GameObject foot2;

These limbs are all in an ArrayList, appropriately named *limbs*, which enables easier access in other parts of the program. The size and world-coordinates of these *GameObjects* are all based on an estimate on what looked reasonably able to emulate a bipedal walker. These body parts are then connected using joints.

Joints

The bipedal walker also requires that the individual bodies are connected to each other, so that they may together emulate the body and limbs of our biped. The following image illustrates how the walker has been constructed:

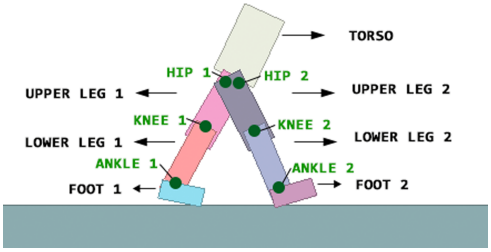


Figure 3.2: Illustration of how the body is constructed

A *joint* is a tool that can connect two bodies to each other in order to constrict their movement in way relative to each other. For the bodies, the joints will literally serve as joints that connect the parts to each other and, as such, restrict how the biped body performs.

Below is an example from the program of the joints:

```
public static ArrayList<RevoluteJoint> joints = new ArrayList<>();
RevoluteJoint hip1;
RevoluteJoint hip2;
RevoluteJoint knee1;
RevoluteJoint knee2;
RevoluteJoint ankle1;
RevoluteJoint ankle2;
```

The joint that functions most like its human counterpart is the *RevoluteJoint*:

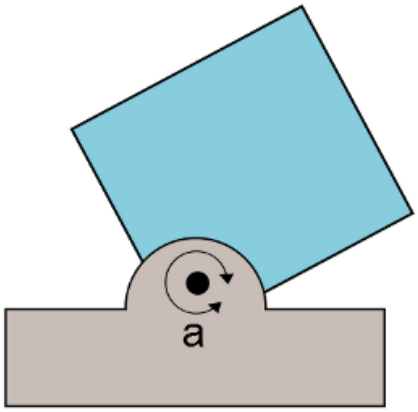


Figure 3.3: Two bodies connected through a joint with a as the pivot point

As depicted in 3.3, the revolute joint allows only rotation between each connected pair of bodies through a single pivot point. The maximum angle of rotation can be set through the *setLimits* method.

Here is a snippet from the code, demonstrating the ability to define angular limits of rotation:

```
knee1 = new RevoluteJoint(upperLeg1, lowerLeg1, new Vector2(0.0, -1.4));
knee1.setLimitEnabled(true);
knee1.setLimits(Math.toRadians(0.0), Math.toRadians(150.0));
knee1.setReferenceAngle(Math.toRadians(0.0));
knee1.setMotorEnabled(true);
```

The limits for these joints are based on estimates of the rotational limits of the human counterpart.

Motor

When using a revolute joint, the ability to use a motor becomes available. The motors will act as the muscles of the bipedal walker through the actions taken in order to move within a given world.

In our program, we define the maximum *torque* for the motor as:

```
Double maxHipTorque = 150.0;
Double maxKneeTorque = 150.0;
Double maxAnkleTorque = 70.0;
Double jointSpeed = 100.0;
```

The values set for each joint, are derived from the rotational limits of its human counterpart.

c.3 Conclusion

We have now presented how we have implemented classes from a dyn4j example and altered these in order to be able to manipulate simulation speed, initialize a $World$-object and render this using the built-in rendering capabilities in Java. We have then demonstrated how bodies can be added to this World object and described the $BipedBody$-class. This class consists of *limbs* GameObjects and joints in the shape of $RevoluteJoints$, which are objects from the dyn4j-library. By using these we have created a biped body, where the functionality of this resembles that of a bipedal walker.

Chapter 4

Learning

In the following chapter, the *requirements* for how we are going to approach the learning aspect of the simulation will be covered. This will include an overview of the *theory* behind such learning methods and a proposal for the *implementation* of these.

a Requirements

Learning is an important part of Artificial Intelligence (AI). There are various ways to deal with the learning element in AI and different ideas to be considered. Let's first introduce the concept of an *agent*. An agent needs to be thought of as the component that takes actions and operates rationally according to a given model, in order to achieve the best result or the best expected results if there is an element of uncertainty[1]. The agent needs to have or gain a perception of its own possibilities and limitations as well as the possibility of interacting with the environment in which the agent exists. It's fundamental that the agent knows or learns what actions are possible to take, considering the state it is in a given moment. Additionally the agent needs to somehow evaluate its performance and learn from it in order to improve its behaviour towards a successful result. The agent consists of two key elements: a *performance* element; an element that decides and executes actions based on knowledge that a *learning* element collects through iterations. To design the learning element the three following issues must be considered[2]:

- Which components of the performance element are to be learned?
- What feedback is available to learn these components?
- What representation is used for the components?

Based on these three issues we can consider the requirements for the learning agent we wish to develop.

Considering the problem of an agent that is programmed to learn how to walk, the component to be learned must be the actions that lead towards walking. In other words:

[1] Russell 4
[2] Russell 649

what body elements to move given a certain state.

The agent needs to receive feedback so it can improve its behaviour. This can be done by giving rewards to the agent, negative and positive, thereby providing it with information on which actions have the best *utility*. Utility is an AI-concept, which is used to describe states or actions, which have the highest value (or utility) to the agent.

Rewarding the agent can e.g. be done by giving positive reinforcement for moving forward and negative for falling. The reward is here seen as reinforcement for performing an action.

The representation of the component is directly related to the learning algorithm. We have chosen to work with active reinforcement learning - more specifically with a Q-learning algorithm, where the representation of the component to be learned by the agent is a value of a state-action pair, called *Q-value*. The concepts of active reinforcement learning, Q-learning and Q-value will be elaborated on in the following section in addition to an explanation of why we have chosen to work with this particular method for reinforcement learning.

b Reinforcement Learning

The use of reinforcement learning is applicable in a situation when an agent needs to learn how to act without prior knowledge of which actions have highest utility. [3] In other words it learns how to act in a given situation based on experience and not in prior knowledge. The general idea behind reinforcement learning is that an agent needs to explore to build a model that will help predict what action is best to take or have best utility for achieving its goal. The agent will then receive feedback its environment. Furthermore, there must be a way for the agent to know whether this feedback return of the performed action is good or bad for the agents utility, to properly create an applicable model that is compatible with its environment[4].

The concept of *rewards* is introduced as a vital source of feedback for the agent to utilize as reinforcement for the model. When an action is taken, the agent must therefore be programmed to evaluate the given action and receive feedback corresponding to the determined value of each state. This helps in maximizing the benefit from the reward that is returned and will ultimately result in a model that is optimized for the environment.

Exploration

In our approach to reinforcement learning the agent learns the model through exploration. This learned model is not the true representation of the environment, since the agent only has a limited knowledge of it, gained through exploration. But the more the agent explores the more it knows of the environment and the more it can begin to choose what

[3] Russell 763
[4] Russell 763

actions hold the best utility. But it is important that the agent somehow is aware that this collected knowledge is not fully reliable and that exploration must continue for reasons of always improving the model. Exploration must therefore be balanced with exploitation considering where the agent is in the learning process. Exploration must be weighted higher in the beginning where exploitation, meaning taking actions based on what the agent has learned so far, is more relevant later on in the process.

b.1 Markov Decision Processes

In artificial intelligence, Markov decision processes (MDPs) are used for decision-making when the outcome of a decision is non-deterministic. MDPs are also useful when working with sequential decision problems, where the agent's utility is dependent of a series of decisions[5]. In each step of a decision process, an agent finds itself in a state s and in this state a number of actions a are available. The agent can then try to go to state s', which will happen at certain probability dependent on the state and the action. The probability for ending up in this state s' is modelled in the **transition model**:

$$T(s, a, s') \tag{4.1}$$

This model indicates the probability of going from state s to state s' when action a is taken.

The motivation for each action is decided by an immediate **reward** $R(s)$ given to the agent. In the design of an MDP one could decide to give positive rewards for desired states and negative rewards for undesired states. If each state (even neutral ones) had a small negative reward, this would give the agent incentive to do actions that would lead to a desired state with a positive reward.
When dealing with a sequence of problems the rewards can be summed using a decay factor. This decay factor γ is a number between 0 and 1. The factor describes the preference for current rewards as compared to future rewards. If γ is 1 the rewards are additive, which means that rewards in the distant future have the same significance to the agent as rewards in the near future, while γ close to 0 makes sure future rewards of much less importance.

A solution to a MDP specifies what actions the agent should take in any state - this is called a policy, which is denoted by π or $\pi(s)$ for a certain state s. The optimal policy π^* is the policy with the highest expected utility (where the expected sum of the rewards is the highest).

Implementation

In this project we use Q-Learning to find the optimal policy for the biped walker. The theory behind this and the implementation in our code can be found in the chapter Q-Learning. In the following sections the classes *State* and *JointAction*, which represent the state and Action in a MDP will be explained. This section will also explain what function approximation is as well as our approach to this concept.

[5]Russell 613

b.2 *State*-class

A state is in our case defined by the angles of all the joints of the BipedBody and the BipedBody torsos relative angle to the world. In the *State*-class this can be seen by the variables of the class.

```
private double worldAngle; // Torso angle relative to World
private ArrayList<Double> jointAngles = new ArrayList<>(); // Angles of joints
```

In the constructor for the *State*-class a BipedBody is set as the input argument and therefore the state is based on the posture of the BipedBody in the instant that it is created. During the development process, we quickly discovered a need for using approximation instead of accurate states, to avoid an ever-increasing number of states throughout the learning process of the agent.

Function approximation

It can be challenging to work with Q-learning given a big amount of states [6]. To handle that, the concept of *function approximation* can be used. The basic concept of function approximation is that it generalizes states, so the agent no longer needs to know every single value associated to each state, or state-action pair. When a state is similar enough to another they will be seen as being the same state. This will considerably lower the number of states.
All joints have a maximum and minimum value and the difference between these two numbers would give the angle interval, in which the joint can be moved. This could, for *hip*1 be calculated as shown here:

```
int angleInterval = (int) Math.toDegrees(hip1.getUpperLimit()) - (int)
    Math.toDegrees(hip1.getLowerLimit());
```

One could argue that some form of function approximation is done here, since the doubles are cast as integers. We have a total number of six joints and if we numbered them from 0 to 5, we could calculate the theoretical number of states as shown here:

$$angle_0 * angle_1 * angle_2 * angle_3 * angle_4 * angle_5 * relativeAngle \tag{4.2}$$

We have included a method in the *State*-class, which is able to calculate the theoretical number of states based on the calculation above. This method is seen in code below:

```
public static long getTheoreticalNumberOfStates() {
// The numbers for these intervals are found by looking at the set upper- and
    lower limit in each joint
int hipInterval = Math.round(55) / roundFactor;
int kneeInterval = Math.round(150) / roundFactor;
int ankleInterval = Math.round(30) / roundFactor;
int relativeAngle = Math.round(360) / roundFactor;

return
    (hipInterval*hipInterval*kneeInterval*kneeInterval*ankleInterval*ankleInterval*relativeAng
```

[6] Russell 777

```
}
```

This method is used in the GUI, where the user can set the round factor and then see the approximate number of states as a label.

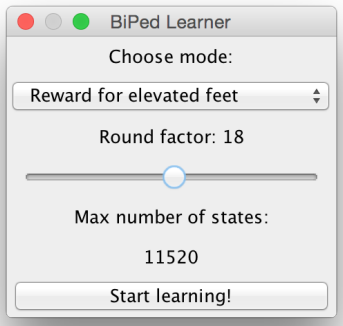


Figure 4.1: The round factor shown in the GUI of the simulation

The *round factor* is used for rounding the number of angles of a certain joint, and thereby minimize the number of states.
7 If roundFactor is set to 5, we have $282,268,800$ number of states, while when it is set to 25, this number is reduced drastically to $2,016$. This shows the necessity of using some sort of function approximation in cases like these with a large number of states.

If we didn't use any form function approximation, a state created from a BipedBody would be considered a new state unless it was exactly equal to an known state, down to the last decimal of every JointAngle. Our approach to this problem is quite simple. For every angle that defines a state we simply round this number by a factor. This can be seen in the constructor of the *State*-class.

```
public State(BipedBody walker) {
      for (RevoluteJoint j : walker.joints) {
          jointAngles.add((double)
              (Math.round(Math.toDegrees(j.getJointAngle()) / roundFactor)));
      }
      this.worldAngle = (double)
          (Math.round(Math.toDegrees(walker.getRelativeAngle()) /
          roundFactor));
}
```

In the above code snippet, we simply add each angle to the states field variables, but before doing so we round this by *roundFactor*, which is an integer set at runtime.

equals()- and *hashCode*()-methods

States are contained in *HashMap* in the *Agent*-class, which will be explained later. It's important to notice however that the methods *hashCode* and *equals* for the class have

both been overridden, in order to make search in the *hashMap* more effective. The hashCode is simply returned by using the built-in Java *hashCode*-method on each states *toString*-method. This toString, also overridden, simply returns a string of the angles that define the state.

fillAction()-method

The *State*-class also contains a method, which adds actions to a state. Firstly we will explain the *JointAction*-class, which is the class that contains actions for the *Q*-Learning algorithm, then lastly the method for adding actions in the *State*-class will be explained.

b.3 *JointAction*-class

The *JointAction*-class represents an action for the BipedBody. This class is later used in the *Agent*-class in order to determine the expected reward for executing an action. The variables for this class are as follows:

```
RevoluteJoint joint; // Joint used in action
  boolean motorOn; // is motor on or is joint relaxed in this action
  int a; // int indicating negative (-1), locked (0) or positive motor input
      (1)
  boolean noOp;
```

Each JointAction has a RevoluteJoint, which is the joint affected by the action. As described in the Physics Modelling chapter, a RevoluteJoint can be manipulated by a motor. We have decided that each joint has four possible actions. Either the joint is affected by the motor or it isn't, which is indicated by the boolean *motorOn*. If the motor isn't on, the joint is completely loose unless it has reached its maximum or minimum angle. If the motor is on, it can move backwards, lock or move forward, which is indicated by the integer *a*. There is a fifth option for an action to be a noOp, which will be used in the *Agent*-class. This means that there is is no action to do, and it's called noneAction in the *Agent*-class.

doAction()-method

The main purpose of the JointAction is being able to do an action, so this action can be paired with a reward later in the learning algorithm. Doing an action means manipulating the appropriate joints motor. The *doAction*()-method is as follows:

```
public void doAction() {
    // this method is called in order to execute an action
    synchronized (ThreadSync.lock) {
        if (noOp) {;}
        if (this.motorOn) {
            Simulation.walker.setJoint(this.joint, this.a);
        } else {
            Simulation.walker.relaxJoint(this.joint);
        }
    }
```

```
}
```

Simulation.walker is in fact the BipedBody, that is a static object which is instantiated in the *Simulation*-class. This is useful, since we're then able to manipulate it throughout the program, without having to pass it as an argument in methods. The *doAction*-method calls a method from the BipedBody class that sets the motorSpeed for this joint. MotorSpeed is a built-in function for the joints, where speed can be set and then the motor will move in either a clockwise or counter-clockwise rotation.

Adding actions to State

In the program each dynamic state refers to a static ArrayList called *actions*. This means that all states have the same actions available. The *actions*-Arraylist is filled using the *fillActions* method, which is called from the Main-method.

```
public static void fillActions() {
      // This methods creates actions for all joints.
      for (RevoluteJoint joint : BipedBody.joints) {
          actions.add(new JointAction(joint)); // New relaxed action (!motorOn)
          for (int i = -1; i <= 1; i++) { // Loop that creates three actions
              for increase, decrease and "lock" joint
              actions.add(new JointAction(joint, i));
          }
      }
  }
```

This for-each loop adds four actions per joint to the *actions* arraylist. Each state has 24 actions - 4 for each joint.

Reducing the number of actions

The initial idea behind having the same actions available to all states, was that we thought it was an interesting idea to have the agent learn which actions have an outcome and which actions that do not. The agent, as it will be explained later, has no idea of the concept of these actions, but treats them identically until it later learns the rewards associated with each one. In hindsight, this might not have been the best approach. As seen in the sections above, we have a large number of states and when connected to state-actions pairs, we have 24 values for each state, which, with a round factor of 25, would give a total number of state-actions part: $2,016 * 24 = 48,384$.

b.4 (

Conclusion) In the above sections, we have explained the concept of reinforcement learning, which is a machine learning method for having an agent teach itself the model of its environment. We then covered Markov Decision Processes which are used for complex decision-making and we elaborated on our implementation of states and actions in the program.

b.5 Q-learning

Q-learning is an *off-policy and temporal difference* control algorithm developed by Christopher J. C. H. Watkins in 1989. Off-policy means that there is no policy used in the algorithm, and temporal difference is a reinforcement learning approach that operates without a model of the environment [7].
The benefit of Q-learning associated with the off-policy and model-free learning is that an agent can learn without prior knowledge. In simpler terms, nothing is expected by the agent beforehand. This notion, of building an algorithm that could make a walker learn how to walk based only on experience, seemed exciting. As far as our research on learning algorithms went, the Q-learning algorithm seemed to be an accessible algorithm to be used in the research we wanted to do.

In Q-learning the component to be learned by the agent is an *action-value function*. An action-value function stores the utility of an action a in a state s. It can also be called Q-function and it is represented by $Q(s, a)$. In other words there is a Q-value associated to every state-action pair.

For the agent to choose what action to take in a given state, it needs to learn the Q-values associated to the state-action pairs based on experience. In the beginning all the Q-values are set by the designer. One can choose to set them to 0, because there has not been any reward associated to it, since nothing has been experienced yet.

As the agent iterates, the more experience it gains and more Q-values are known. The actions taken in a given state leading to the highest expected reward will have a high Q-value. The Q-values are updated constantly based on the obtained reward.

Learning Rate

One other important concept to be understood in Q-learning is *learning rate*. The *learning rate* α is what determines how the agent takes new information into consideration. New Q-values are learned through the iteration process and, since they play a role in determining the current Q-values, they are here considered. What the learning rate does is determine how new information is weighted when updating Q-values.

Updating Q-values

For updating Q-values the following equation is used:

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R(s) + \gamma max_{a'} Q(s', a') - Q(s, a)) \tag{4.3}$$

The left side of the equation represents the current Q-value $Q(s, a)$ and the right side the updated value. So the updated value becomes the current value in each iteration. This is represented by the arrow pointing left. When updating the Q-value, the current Q-value is added to the reward of the current state $R(s)$ and the estimated future utility $\gamma max_{a'} Q(s', a')$ multiplied by the learning rate α.

[7] Mark Lee 2005

c Implementation

The following section will showcase the various methods that were implemented in the *Agent*-class.

c.1 Agent-class

The *Agent*-class is the brainpower of the learning algorithm. It is where Q-values are stored and the methods for updating Q-values and getting optimal actions are located. Its primary goal is to be able to return the best possible action dependent on the state. The *Agent*-class is based largely on the *QLearningAgent*-class, which is found in the AIMA implementation examples. This has only been slightly altered in order to fit our needs and since this class is essential for the functionality of the program, we will explain all methods and data-types found here.

Data-types

The main field variables for the class are as follows:

```
private double alpha; // Learning rate
private double gamma; // Decay rate
private double Rplus; // Optimistic reward prediction
private int mode;

private State s = null; // S (previous State)
private JointAction a = null; // A (previous action)
private Double r = null;

private int Ne = 1;
private FrequencyCounter<Pair<State, JointAction>> Nsa = new
    FrequencyCounter<>();
public static Map<Pair<State, JointAction>, Double> Q = new HashMap<>();
```

Alpha α and gamma γ have both been explained earlier and they are the learning rate and decay rate for the agent. The values for these variables are set in the constructor. This is dependent on the reward mode chosen at run time and is explained in the Testing chapter.

Since Q-learning deals with state-action pairs we use the data type *Pair* from AIMA that can pair objects. One can add any kind of objects to a pair but in this case we pair state and action as follows: $Pair < State, JointAction >$. They are then used in a hashMap called Q where the state-action pairs are associated to the respective Q-value.

State s and JointAction a are set in the constructor as it can be seen here:

```
public Agent(int mode) {
  ...
  /* Parameters set dependent on mode */
      }
      this.s = Simulation.walker.getState();
```

```
        this.a = Main.initAction;
    }
```

The state is set to the walkers state when the agent is initialized while action a is set to a random initial jointAction, which is performed in the beginning.

Frenquency-Counter and Exploration

AIMA provides a frequency counter which is here called Nsa. This is for counting the frequency of visited state-action pairs. This is used for the reason of determining how much the agent should explore given a state-action pair. The more times the agent has executed a certain pair, the more it knows about it. Nsa as well as the current state-action pair is an input argument in the method $f()$. This method is explained below, but before two other variables need to be considered first; Ne is a parameter, which is used in a simple method for exploration. If a state-action pair has been visited more than Ne times, the agent uses the actual experience instead of $Rplus$ when updating the Q-value. Since we already deal with a large number of states, Ne has been set to 1. $Rplus$ represents an optimistic reward for the agent. It is used if the agent has not visited a state Ne times.

The $f()$ method is as follows:

```
protected double f(Double u, int n) {
    if (null == u || n < Ne) {
        return Rplus;}
    return u;}
```

In this method the input arguments are u, which is the Q- value for the given state-action pair and n, which is the state-action pair frequency. This method is used within the private method $argmaxAPrime(StatesPrime)$ that returns a JointAction based on the optimal policy.

Finding the best policy

For finding the optimal policy, the private method $argmaxAPrime()$ is used. It takes a state $sPrime$ as an argument and returns a JointAction. $sPrime$ and $aPrime$ are respectively the representations for the state following state s and the action associated to it. The method is as follows:

```
private JointAction argmaxAPrime(State sPrime) {
    JointAction a = null;
    Collections.shuffle(sPrime.getActions());
    double max = Double.NEGATIVE_INFINITY;
    for (JointAction aPrime : sPrime.getActions()) {
        Pair<State, JointAction> sPrimeAPrime = new Pair<State,
            JointAction>(sPrime, aPrime);
        double explorationValue = f(Q.get(sPrimeAPrime), Nsa
                .getCount(sPrimeAPrime));
        if (explorationValue > max) {
            max = explorationValue;
            a = aPrime;}}
```

```
        return a; }
```

The only thing we have added to this method is the shuffling functionality, that shuffles the list of possible actions. The reason is that if all the actions are equally good, the method always returns the same action. Shuffling the list emphasises the exploration element and makes exploration less predictable. Through a for-each loop all the possible actions in state $sPrime$ are analysed. A variable called $explorationValue$ is created and this is set to the double returned by $f()$. The JointAction with the highest exploration value is returned.

Terminal State

If the current state being analysed is terminal the agent acts differently. A terminal state means that the Q-value does not need to be updated and the information associated to this state-action pair is therefore directly inserted to the HashMap Q as shown below:

```
if (isTerminal()) {
            Q.put(new Pair<>(sPrime, noneAction), rPrime);
        }
```

A terminal state has no action assigned to it and therefore the JointAction noneAction is inserted here. The method $isTerminal()$ is used and what it does is basically checking if the walker has currently fallen or it is out of bounds. If this is the case the walker is to be reset and this is therefore considered a terminal state.

```
    private boolean isTerminal() {
        if (mode == 0) {
            return Simulation.walker.hasFallen() ||
                !Simulation.walker.isInSight(); // Falling is a terminal state in
                mode 0
        }
        return false;
    }
```

The optimal future Q-value

As explained in the equation for updating Q-values, the maximum future Q-value must be calculated. This is done by the method $maxAPrime()$ which is shown below:

```
private double maxAPrime(State sPrime) {
        double max = Double.NEGATIVE_INFINITY;
        if (sPrime.getActions().size() == 0) {
            // a terminal state
            max = Q.get(new Pair<State, JointAction>(sPrime, noneAction));
        } else {
            for (JointAction aPrime : sPrime.getActions()) {
                Double Q_sPrimeAPrime = Q.get(new Pair<State,
                    JointAction>(sPrime, aPrime));
                if (null != Q_sPrimeAPrime && Q_sPrimeAPrime > max) {
                    max = Q_sPrimeAPrime;
```

```
                }
            }
        }
        if (max == Double.NEGATIVE_INFINITY) {
            // Assign 0 as the mimics Q being initialized to 0 up front.
            max = 0.0;
        }
        return max;
    }
```

This method takes a state as a parameter and returns a double called max which is the maximum expected Q-value. There is an if-statement for determining first of all if this state is terminal. If this is true then the same approach as previously explained is used. If this is not the case then all the possible actions in the given state are iterated through a for-each loop.

For every state-action pair the following is checked:

```
if (null != Q_sPrimeAPrime && Q_sPrimeAPrime > max)
```

If $Q(s', a')$ equals $null$, this indicates that this state-action pair is previously unexplored. Additionally we check if $Q(s', a')$ is larger than max. If this is true max is set to $Q(s', a')$. When the iteration of the loop is done max is the highest Q-value for s' and the value is returned.

The execute()-method

One of the most important methods in the *Agent*-class is the *execute*()-method. This is a public method which returns a JointAction. The purpose of this method is to return the jointAction, which is the result of the optimal policy calculated in the $argMaxAPrime$-method. Seeing as the other methods have been described, the *execute*()-method will be explained and elaborated upon line by line.

```
State sPrime = Simulation.walker.getState();
double rPrime = Simulation.walker.reward();
```

One could say that the agent "backtracks" and therefore $sPrime$ is set to the current state, since this is the outcome of the last time the *execute*()-method ran. $rPrime$, the associated reward for the previous state-action pair is set accordingly. One of the jobs for this method is to update the Q-value before calculating the optimal policy. The basecase is that the current state $sPrime$ is a terminal state and if this is the case the state is paired with a non-action and put into the HashMap containing the Q-values.

```
if (isTerminal(sPrime)) {Q.put(new Pair<>(sPrime, noneAction), rPrime);}
```

After this check is done the Q-value for the current state-action pair is stored as the double Qsa:

```
Double Qsa = Q.get(sa);
          if (Qsa == null) {Qsa = 0.0;}
```

In cases where Qsa is null, which would happen if the current state-action pair is unexplored, Qsa is set to 0.0. Following this, the double r is set to the walkers current reward:

```
r = Simulation.walker.reward();
```

This uses the method in BipedBody, which returns a double that is the reward. After assigning r and Qsa we are then able to update the Q-value. This is done using the update equation for Q-Learning:

$$Q(a,s) \leftarrow Q(a,s) + \alpha(R(s) + \gamma max_{a'} Q(a',s') - Q(a,s)) \tag{4.4}$$

The equation above can be seen implemented as code below:

```
Q.put(sa, Qsa + alpha * (r + gamma * maxAPrime(sPrime) - Qsa));
```

In cases where the current state is terminal, no action should be returned:

```
if (isTerminal(sPrime)) {
          s = null;
          a = null;
          r = null;
      } else {
          this.s = sPrime;
          this.a = argmaxAPrime(sPrime);
          this.r = rPrime;
      }

      if (a != null) {
          Main.gui.update(a);
      }
      return a;
```

If $sPrime$ ia not terminal, JointAction a is found using the $argmaxAPrime$-method, the GUI is updated with the current action and JointAction a is returned to the loop in the main method, ready to be executed by the walker.

c.2 Conclusion

Q-learning is a useful approach for model-free reinforcement learning. As opposed to Markov Decision Processes its utility is not connected to states but to state-action pairs, which are denoted $Q(s,a)$. We have implemented Q-learning in our program in the $Agent$-class, which is based on an implementation example from AIMA. This class has methods for updating Q-values and returning a $JointAction$ object based on the optimal policy.

Chapter 5

Program Flow

Below, in the following chapter, we will go over the *requirements* for the program and the overall structure, as well as the implementation of these. This will include the various *packages* utilized, a rundown of the *main-method* and its interaction with all relevant classes, as well as an explanation of the *graphical user interface*.

a Requirements

In this section we will explain what we see as the requirements for the program. Firstly we will deal with the general requirements for the program and will then be going into the more specific requirements of the program structure.

a.1 Program

The purpose of this program is to create a physics simulation which can act as an environment for a Q-learning algorithm involving a two-dimensional biped walker. Since we have included different reward modes in order to show the possibilities and short-comings of the learning algorithm, the user must have the ability to select a reward mode before the simulation is started. This can be done by having a dialogue window pop up, when the program is run, where the user can select a mode. After choosing the desired reward mode, the simulation should start running in a new window, while there's a GUI that should contain controls for and information on the simulation. These should, without noticeable latency, be in sync with the simulation window that is running.

a.2 Program Structure

Since we are not experienced programmers, the development process of this program has not only been a learning process in implementing and working with AI, but also in working with general programming and software development. We knew this from the beginning and therefore aspired to have a program structure, which enabled expandability and modularity.

Abstraction, modularity and interaction between classes

One of the main requirements, if we are to have a program which is expandable, is abstraction. Abstraction, in computer science, is about hiding irrelevant details and focusing on properties rather than the inner-workings of each class. We would also like to strive for *high-cohesion*, where all data in a class is conceptually connected to that class and *low-coupling*, where the classes, are able to function independently and only pass data between classes when it's necessary for the responsibilities of classes. Standard good practice would be to encapsulate everything, where all data is only passed through methods. While we do strive for some encapsulation such methods can quickly add numerous lines of codes to a programming language which is already verbose. Therefore we strive for encapsulation, but there are cases where this does not have highest priority in order to make the program less verbose and more easily understandable.

b Implementation

b.1 Packages

The program consists of three packages - **QLearning** , **Rendering-dyn4j** and **sample**. The program classes are located in these three different packages in an effort to make the program flow more logical. Since the program consists of three distinctive main parts, the packages are created in an attempt to underline this. The classes will be listed below and the public methods will be explained. We have chosen only to elaborate on the public methods here because the main purposes of this chapter is to demonstrate and analyse the interactions and the flow within the program. In addition to this, he methods used from the example at the dyn4j-website, are not included here since no major changes have been made to them.

QLearning Package

Agent Class

public JointAction execute()
This method is where the Q-values are updated and a JointAction is returned.

JointAction Class

public JointAction (RevoluteJoint joint)
Constructor returning an action for making the joint relaxed.

public JointAction (RevoluteJoint joint, int i)
Constructor for an action to move.

public JointAction ()
Constructor for an none operation.

public void doAction ()
Method for executing an action.

State Class

public State(BipedBody walker)
Constructor for making new state objects based on the BipedBody.

public static void fillActions)
Method for filling the static ArrayList actions, which contains the JointActions available to all states

Rendering-dyn4j Package

BipedBody Class

public BipedBody
Constructor where the biped walker is created

public void setJoint(RevoluteJoint joint, int x)
Method for setting manipulate joint. The first input is the joint to be manipulated and the second decides how it is to be manipulated.

public void relaxJoint(RevoluteJoint joint)
Method is for setting the respective joint relaxed.

public boolean hasFallen()
Method for detecting if the BipedBody has collided with the floor, returns a boolean.

public double reward()
Method for returning rewards given the selected mode.

public void resetPosition
This is used for reseting the biped walker to its initial position.

public boolean isInSight
Method checking if the biped walker is inside the rendered frame.

CollisionDetector Class

public boolean collision(Body body, Body body1)
Method used for checking collision between two objects of type Body.

GameObject Class
From dyn4j example, not modified. Used for drawing purposes.

Graphics2DRenderer Class
From dyn4j example, not modified. Used for rendering purposes.

Simulation Class
From dyn4j example. Used for simulation purposes.

ThreadSync Class
Class for synchronising threads.

sample Package

Generation Class

public Generation(int generationNumber, double accumulatedReward)
Constructor for *Generation*-objects, which are used in the GUI table.

GUI Class

public GUI (Simulation world)
Constructor for the Graphic User Interface, where the graphic elements and action listeneres are created.

public void update (int Nsa, double Q)
Method for updating Nsa and number of Q-values.

public void update ()
Method for updating generation number.

public void update (JointAction action)
Method for updating the current executed action.

HighScoreTable Class
Class for creating a table of Generations.

MainClass

public static void main(String[args)]
The Main method of the program.

public static void learn()
This method is called in the main method and contains the main learning loop for the agent.

StartDialog Class
This class is where the start dialog window is created.

c Static instantiation of certain classes

We have chosen to use static instances of classes in cases, where we wanted to ensure that there is only a single instance of a certain class.

```
public static Simulation simulation;
public static GUI gui;
public static Agent agent;
```

This is, as seen in the code above, the case for the *Simulation*-, *GUI*- and *Agent*-class. Making these objects static enabled us to access their methods and variables throughout the program without having to pass the objects as arguments in methods or constructors.

This helped in reducing the complexity of the program, e.g. in the *Agent*-class where the agent always acts based on data from the *same* instance of the *BipedBody*, which is a static object instantiated in the *Simulation*-class. While securing that there is only a single instance of each object is one of the advantages of a static approach, this is also one of the short-comings.
If this program was to be developed further there could be certain advantages in taking an approach that is not necessarily based on single instances of a lot the classes that are static in our approach. The advantages to a more dynamic approach could for example be that one could have several simulations running simultaneously on multiple threads or several BipedBodies in one simulation in an effort to make *Agent* learn faster. This, however, has not been a focus-point on the development of this particular program.

d *main*- and *learn*-method

This section will explain the *main*- and *learn*-method. This will focus on the *Main*-class and the interaction between and instantiation of classes with elaboration on choices regarding static or dynamic instantiation of classes. Lastly this will lead up to a discussion on our approach on this matter and the advantages and disadvantages connected to this.

d.1 Main learning loop

When the program is run this is done from the *main*-method. As seen below the main method simply creates a new *StartDialog* object, where the user can choose mode and after the dialog is disposed the *learn*()-method is called.

```
public static void main(String[] args) {
       StartDialog dialog = new StartDialog();
       learn();}
```

The learn method is the central method to the program, where the different classes come in to play. This method has been commented in the code, but we will go it through it line-by-line nonetheless in order to ensure the readers understanding of this method. This method starts by doing a number of method-calls including instantiation of *simulation*, *agent* and *gui*. After this is done, the main-loop of the method is executed:

```
while (true) {// Loops as long as program is running
           accumulatedReward = 0;
           double t = 0;
           boolean isTerminal = false;
```

The *while*(*true*)-test is not elegant, but it gets the job done, it simply loops the entire while-loop until the program is exited by the user. After this three variables are created. The first is *accumulatedReward*, which is a double and is used in mode 0. This represents the total reward which each generation has been able to accumulate after each ended generation. This is used in the GUI. The variable t is a time-counter, which is used to set a limit on the frequency of actions returned by the agent. Finally isTerminal is a boolean, which is also used in in mode 0. This is set to true, if the BipedBody ends up in a terminal state. After this a new *while*-loop, that runs as long as isTerminal equals

false, is created. Therefore this loop is only ever broken in mode 0, since the other modes never reach a terminal state.

```
while (!isTerminal) {
            if (!Simulation.walker.isInSight()) { // Reset if out of sight
                isTerminal = true;
            }
```

In the code above, there is an *if*-statement checking if the *walker.isInSight*()-method returns true. This calls a method in the *BipedBody*-class, which returns false if the *BipedBody* is outside of the screen. This is done in order to make sure the walker is visible at all times. For the next test the value of t is tested:

```
            if (t > 400000) {
                // Observe and execute
                JointAction action = agent.execute();
                if (action != null) {
                    synchronized (ThreadSync.lock) {
                        action.doAction();
                    }
                } else {// If null is returned, agent is at a terminal state
                    isTerminal = true;
                }
                t = 0; // Reset time to zero
            }
            t += simulation.getElapsedTime(); // Increment time
        }
```

In the first versions of the program, we had issues with *agent.execute*() being called at too high a frequency. This meant that the agent kept analysing the current state and returning actions, even if the current state had yet to be changed. For each round in the nested while-loop, t is incremented by the amount of time that has passed in the simulation. Only if $t > 400000$ *agent.execute*() is called and t is then set to zero, which causes the "waiting" period to start over. The number 400000 was adjusted after some tests and this seemed like an appropriate rate, since *agent.execute*() is now executed several times a second as opposed to thousands of times per second.

Whenever t is larger than 400000, the following code is executed:

```
JointAction action = agent.execute();
                if (action != null) {
                    synchronized (ThreadSync.lock) {
                        action.doAction();
                    }
                } else {// If null is returned, agent is at a terminal state
                    isTerminal = true;
                }
                t = 0; // Reset time to zero
```

First of a new JointAction *action* is set to the JointAction returned by *agent.execute*(). This methods returns the JointAction from the optimal policy and this method only

returns null if the current state is a terminal state. If reward mode is 0, null would indicate that the walker has fallen or is out of bounds. If this is the case *isTerminal* is set to true, which would cause the outer loop to break and the walker to be reset.
If the action returned is not null, this action is then the optimal policy. The JointAction is then performed by calling the *doAction*()-method.

Resetting the walker

As explained, terminal states only exist in reward mode 0. If *isTerminal* is true the loop is broken and the following is executed:

```
if (isTerminal) {
              updateGuiTable();
              Simulation.walker.resetPosition();
}
```

This simply updates the GUI table with the generation that has just ended. After this *resetPosition*() is called from the walker, which resets the walker to its initial position. After being reset to this initial position the state is no longer terminal and the *agent.execute*() while be called once again after $t > 40000$.

d.2 Threads

One of the requirements for the program was to have a GUI, which enabled the user to control and see information from the simulation- and learning environment. The GUI should be updated automatically. When implementing methods for altering the simulation speed, we quickly ran into threading issues. The issue here seemed to be that the GUI and the frame containing the simulation ran on different threads, which caused issues with a lack of synchronization between these threads. With help from the developer of dyn4j, William Bittle, we were able to create a solution to these issues. These solutions will be presented below.

ThreadSync-class

Our solution to this was creating a class named *ThreadSync*, which contains a *lock*-object. This object is built-in the Java API and has the ability to lock threads. There are more elegant solutions to the threading issues we were facing, but this fix made sure no exceptions were thrown when manipulating simulation speed and helps avoiding deadlocks. The implementation simply works by synchronizing interactions with the simulation with the *ThreadSync*-object as seen below:

```
synchronized (ThreadSync.lock) {
    this.world.update(elapsedTime, Integer.MAX_VALUE);
}
```

The code above is from the *Simulation*-class and handles updating the simulation. By making this method synchronized with *ThreadSync.lock*, this method isn't run until it is in-sync with the *lock*-object. The same idea is put into practice with all *actionListeners*

in the GUI, as it can be seen in the example below, where the action listener for the simulation-speed slider in the GUI is synchronized:

```
simSpeedSlider.addChangeListener(e -> {
        // Slider for changing simulation speed
        synchronized (ThreadSync.lock) {
            Main.simulation.setSimulationSpeed(simSpeedSlider.getValue());
            simSpeed.setText(Main.simulation.getSimulationSpeed() + " x
                Speed");
        }
    });
```

Issues

The method explained above has worked in our tests with a single exception: there is a *JTable* in the GUI, which contains info on each generation and the accumulated reward for this generation, as shown below:

Generation #	Total reward
156	11,500
69	2,583.439
46	2,515.921
119	1,592.926
107	153.299
41	75.094
7	-279.95
139	-332.315
172	-389.139
118	-411.535

Figure 5.1: JTable in GUI

When new rows are added to the *JTable*, the rows are automatically sorted. This is done by a method in the *JTable*-class. This method is *not* synced to the *ThreadSync.lock*-object, since this would require overriding of the *JTable*-methods. With this method not synced, the *JTable* sometimes throws an exception. Since this doesn't interfere with the simulation, we estimated that this bug did not have a drastic effect on the functionality of the program and therefore we have left this minor issue unresolved for now.

e Graphical User Interface

The Graphical User Interface (GUI) is divided into two parts. There is a start-dialog for choosing which mode to run and one for controls and information during simulation. They are created using the Swing toolkit for Java.

e.1 Start window

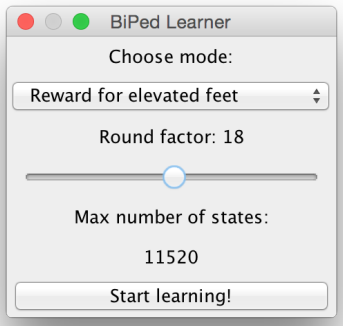


Figure 5.2: Start Dialogue Window

The start dialog has three main components. There is a *JComboBox* used for making a drop-down menu, where the user can choose a reward mode. Besides this, the user can also choose the rounding factor using a *JSlider*-component, that goes from 5 to 30. The rounding factor has been explained before, but in a few words this factor determines how many states the the agent operates with. The maximum theoretical number of states available is calculated and then presented to the user, using the component *JLabel*. Finally there's a start-button *JButton* for starting the simulation.

e.2 Control window

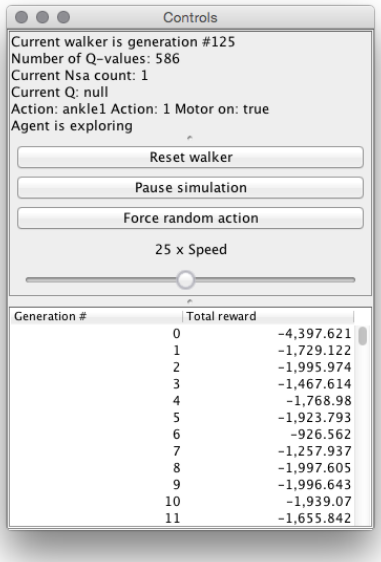


Figure 5.3: Control Window

As shown above, the control window is divided in to three parts. The first part is where information about the agent and the learning process is shown. The second part is for controlling and adjusting the simulation and the third part is to follow the development of the walker.

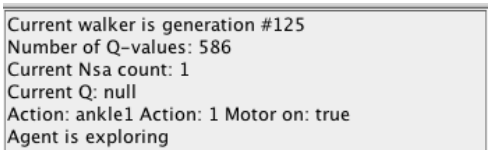


Figure 5.4: Control Window - First part

All the elements of the first part are *JLabels* that are updated throughout the learning process. They will be explained here, one by one. There is *Generationcounter* to keep track of how many generations of the walker have that been made. A new generation is made every time the walker falls or out of bounds. There is Q-value counter so the user is able to see how many Q-values the agent has learned all in all. There is also a counter for counting how many times the current Q-value has been updated. This value reflects how explorative the agent is and it gives an idea of how the agent takes the already known Q-values into consideration. There is a label showing the current Q-value and a label showing which action is being executed currently. Finally there is a label showing if the agent is learning or exploring. "Agent is exploring" is shown when the agent is in a state, doing an action, that it hasn't done before. "Agent is learning" is shown when the agent has performed the current state-action before.

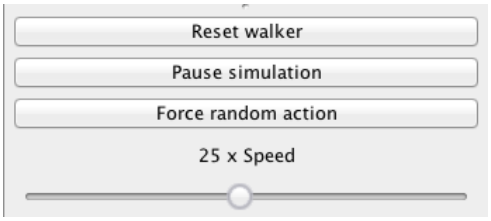
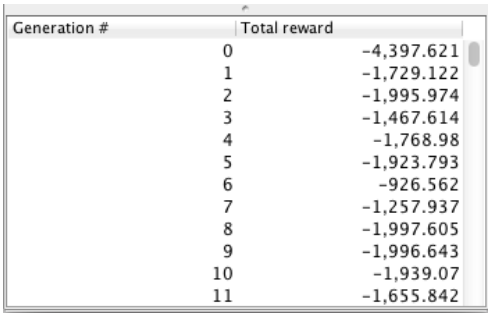


Figure 5.5: Control Window- Second part

The control panel has 3 buttons and a slider. There is a button for resetting the walker, a button for pausing the simulation and there is a button that forces the walker to do a random action, in case the user wants to break the followed policy and make the walker do an action that is not the one return by the *execute*() method.



Generation #	Total reward
0	-4,397.621
1	-1,729.122
2	-1,995.974
3	-1,467.614
4	-1,768.98
5	-1,923.793
6	-926.562
7	-1,257.937
8	-1,997.605
9	-1,996.643
10	-1,939.07
11	-1,655.842

Figure 5.6: Control Window- Third part

Finally there is a table showing the generations and their total accumulated reward. This table is only shown in the reward mode 0 (walking forward), because this is the only mode operating with generations.

f Conclusion

Considerations concerning the program structure and overall program flow have been made, based on the requirements for the three main parts of the program.

We have made the decision to have some methods *static* in order to access data between classes.

In order to avoid thread issues we have chosen to have a *ThreadSync* class, but there are still minor issues relating to the *JTable*. The GUI is made using Swing and is composed of two windows; one start dialog window is shown, in order fpr the user choose the reward mode and one for showing information about the learning process, in order to control and adjust the simulation and to show the development of the walker through generations.

Chapter 6

Testing

In the following chapter we will describe several different test-cases and experiments and the results of these. This will lead up to a discussion and conclusion on our experiences on using Q-learning, including suggestion on improvements for the program.

a Procedure

Throughout the development of this program, we have been testing and debugging in order to improve performance and experiment with parametrisation and optimisation of the learning algorithm and the program in general. We have not at all reached a point, where the biped is able to walk in a conventional way, even after longer simulation rounds. However, we have been able to see that the *agent*) is able to learn and over time choose actions, which have a higher utility.

a.1 Reward modes

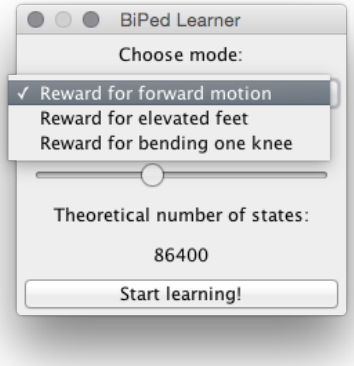


Figure 6.1: Initial options for reward mode

As seen in picture 6.1, this drop down menu is shown to the user when the program is run. As mentioned earlier, it makes it possible to choose different test-cases where the

rewards and the simulation environment vary. This makes it possible to choose from a number of different modes, which all have different complexities and parameters set and assists in further investigating the strengths and short-comings of the agent.

Method for testing

For testing we ran each case independently and ran experiments in relation to these. For each time *agent.execute*() was run, we recorded the reward for the walker and are therefore able to create graphs and compare these to each other. The number of times we allowed *agent.execute*() to run, was dependent on the kind of tests that we wanted to do. We then ran tests for the same mode a number of times, where the parameters were changed for the *agent* in order to shed light on the effect of these adjustments on the capabilities of the bipedal walker.

b Test 1 - Bending knee

This test-case can be seen as the most simple, as the reward is solely dependent on the angle of the walker's knee. The idea behind this case is to determine whether the agents ability to translate the experience it gains, based on the reward, into a actions that converge with our expectations of the effects from said reward. Furthermore, in this case, $hasFallen()$ is not a terminal state and walls are created, so the BipedBody is never *out of bounds*. The reward defined in this mode is as follows:

```
reward = Math.toDegrees(Simulation.walker.knee2.getJointAngle());
```

Here, the walker is given reinforcement based on the angle and the more the knee is bent, the higher a reward is returned. For all tests in this mode, $Rplus$ was set 150, since preliminary testing showed that with the specified reward, this value was approximately the maximum possible reward attainable for the walker in this test-case. In this test we only tried different parameters for learning rate α and decay rate γ.

We did two tests where we ran *agent.execute*() 20,000 times and the results can be seen in the following graphs:

Test 1

First test was done with $\alpha = 0.1$, which is quite a low learning rate:

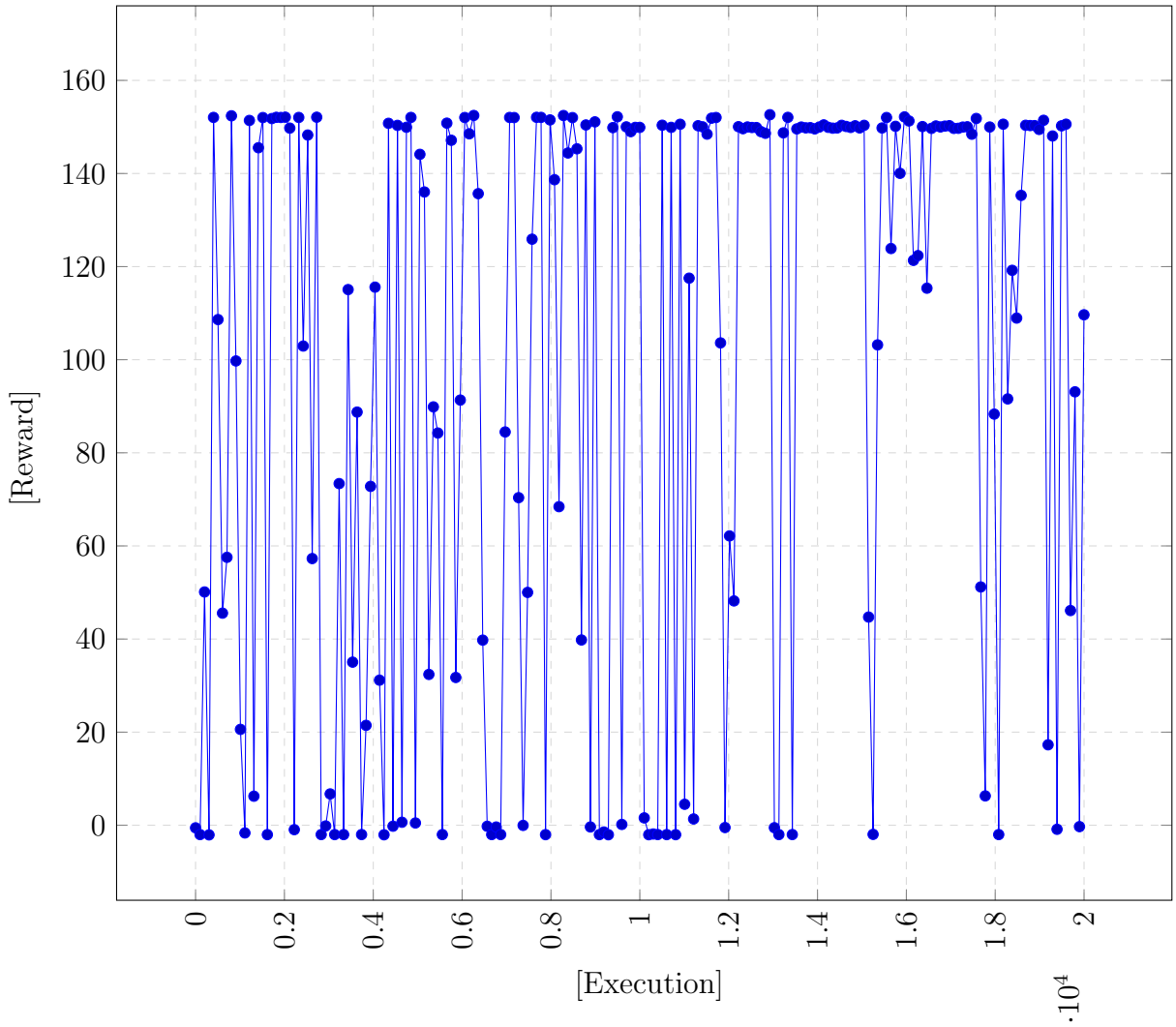


Figure 6.2: Rplus= 150 alpha = 0.1 gamma = 0.1 Rounding factor = 25

As it can be seen there is a lot of exploration going on and the agent does not stabilize at a point, but instead keeps exploring throughout the 20,000 executions. If the program was running for a longer time, there might more success in finding a stable reward. This can be supported by the following the graph. Here $\alpha = 0.9$:

Test 2

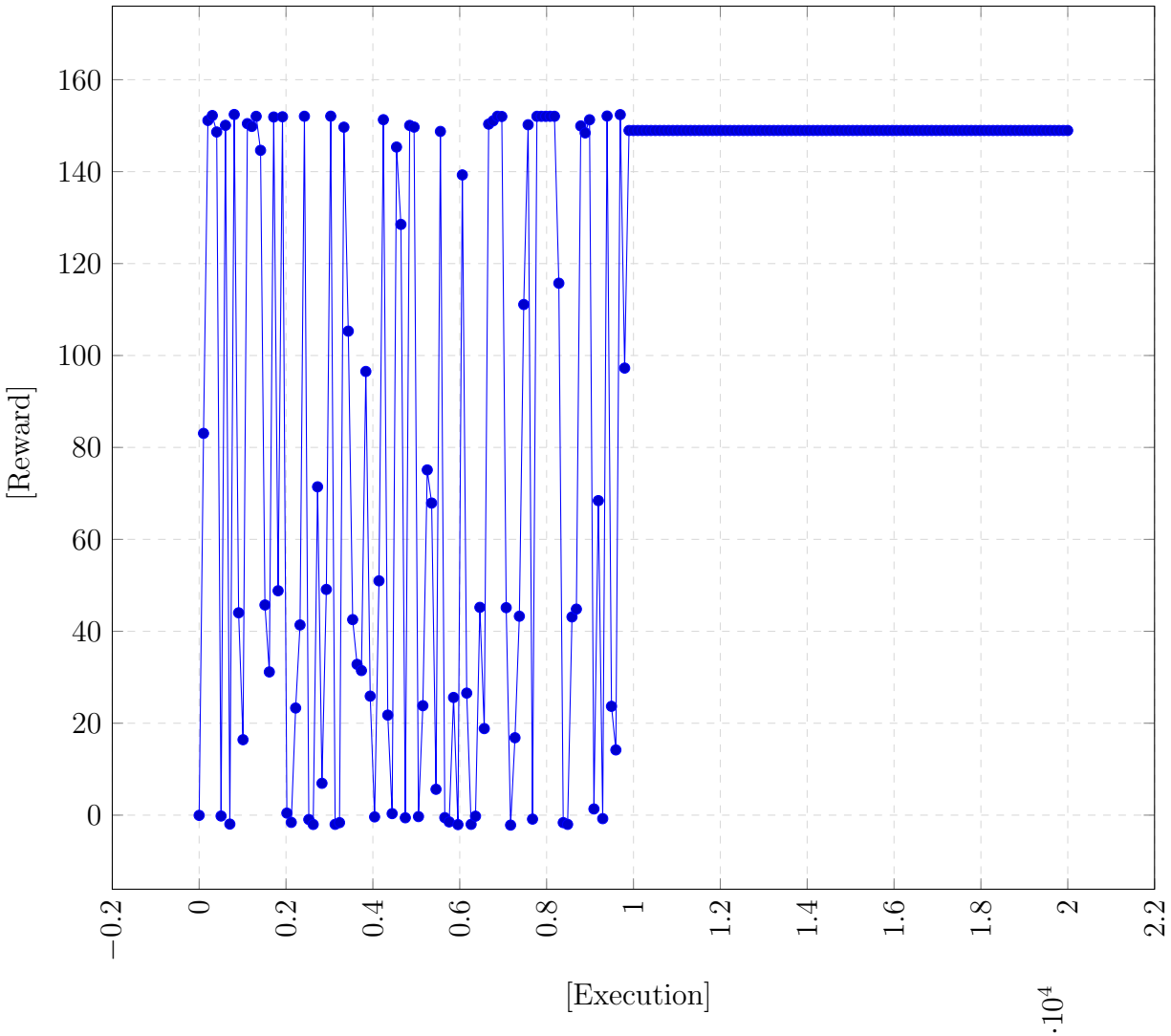


Figure 6.3: Rplus=150 alpha = 0.9 gamma = 0.1 Rounding factor = 25

As it can be seen above, the agent continues exploring until it has executed approximately 10,000 actions and then starts stabilizing around a reward, which is only a fraction lower than what seems to be the highest reward explored. This was the posture in which the walker stopped doing actions:

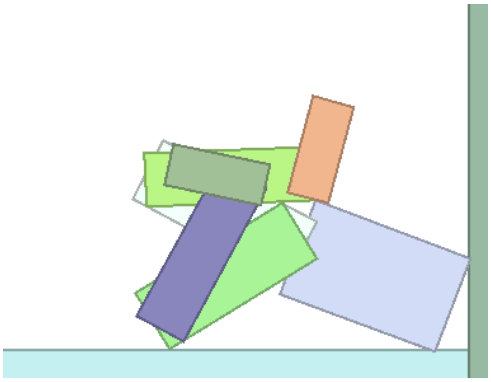


Figure 6.4: Final position in tests for with reward for knees bent

c Test 2 - Elevated feet

In the next test-case, the reward is for keeping the walkers' feet as high as possible. This test has a bit more complexity than the test before, since there is more than one action to be done to achieve a high reward. The walker also needs to figure out how high the feet can be considering it needs to be balanced.

Rewarding was done as shown below:

```
reward = 1500 + ((Simulation.walker.foot2.getWorldCenter().y+
    Simulation.walker.foot1.getWorldCenter().y) * 1000);
if(!feetOnTheGround()){reward+=1000;}
```

In accordance with this reward configuration, the height in which the walker positions its feet along the *y-axis*, determines the reward. This should encourage the walker to lift its feet as high as possible. In addition to this, the *if-statement* is in place to further encourage the walker to keep the feet off the floor. This means that, in relation to our expectations for the effects of this *if-statement*, will result in the walker being on the floor, pushing its feet up in the air. Therefore, $hasFallen()$-method is not terminal and to prevent it from going out of bounds on either side of its starting point, while on the floor, walls are created on each edge of the scene.

The following sections will focus on the results of the three individual tests that were conducted in this case. It should be noted that, unlike in the previous case with the *bended knee* reward, we chose to run *agent.execute* 50,000 times instead of 20,000 to allow more time for the agent to learn.

The rounding factor is also set to a higher value, for minimizing the number of states. The optimistic reward *Rplus* has been set to 600. Like in the previous test-case, this was based on preliminary tests within the mode. The learning rate value has been set 1 in all the three tests, since we could conclude from the test above - bending knee test - that a higher learning rate meant a lot in the learning process, due to the large amount of states. Therefore the following test have been made to emphasize the significance of the decay rate. The decay rate is the only variable that is changed in each test, starting with a low

value going to a high value. A quick recap of how decay rate is considered: If $\gamma = 1$ 1 the agent weighs future rewards as it weighs current rewards, if the decay factor is set to a lowers value the agent sees future rewards as less important.

Test 1

In this first test the decay rate was set to 0.1, a very low value, and the graph below shows clearly that the agent is not capable of stabilizing .

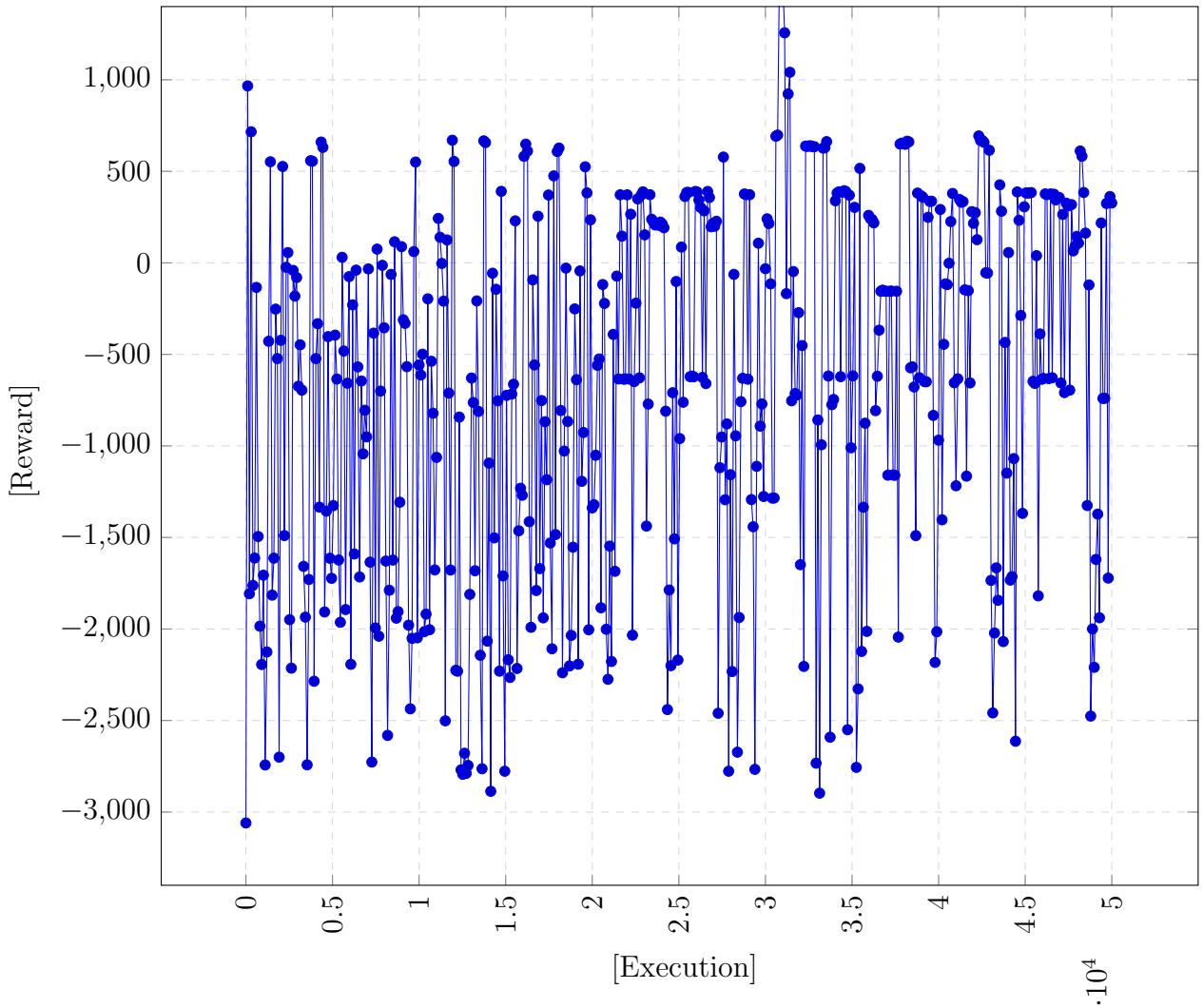


Figure 6.5: **Rplus= 600; alpha = 1 gamma = 0.1, Rounding factor = 30**

As it can be observed from the graph above the agent is very explorative. Considering the decay rate and the agents task of elevating its feet, we can, to some extent, conclude that the agent gives higher utility to instant rewards, as opposed to rewards in the far future. After some iterations the agent has probably learned how to lift the feet, but the problem here relies on keeping balance.

Test 2

In the second test that was run, we set $\gamma = 0.5$. The graph shows that this made a difference on the walkers behaviour and decision making.

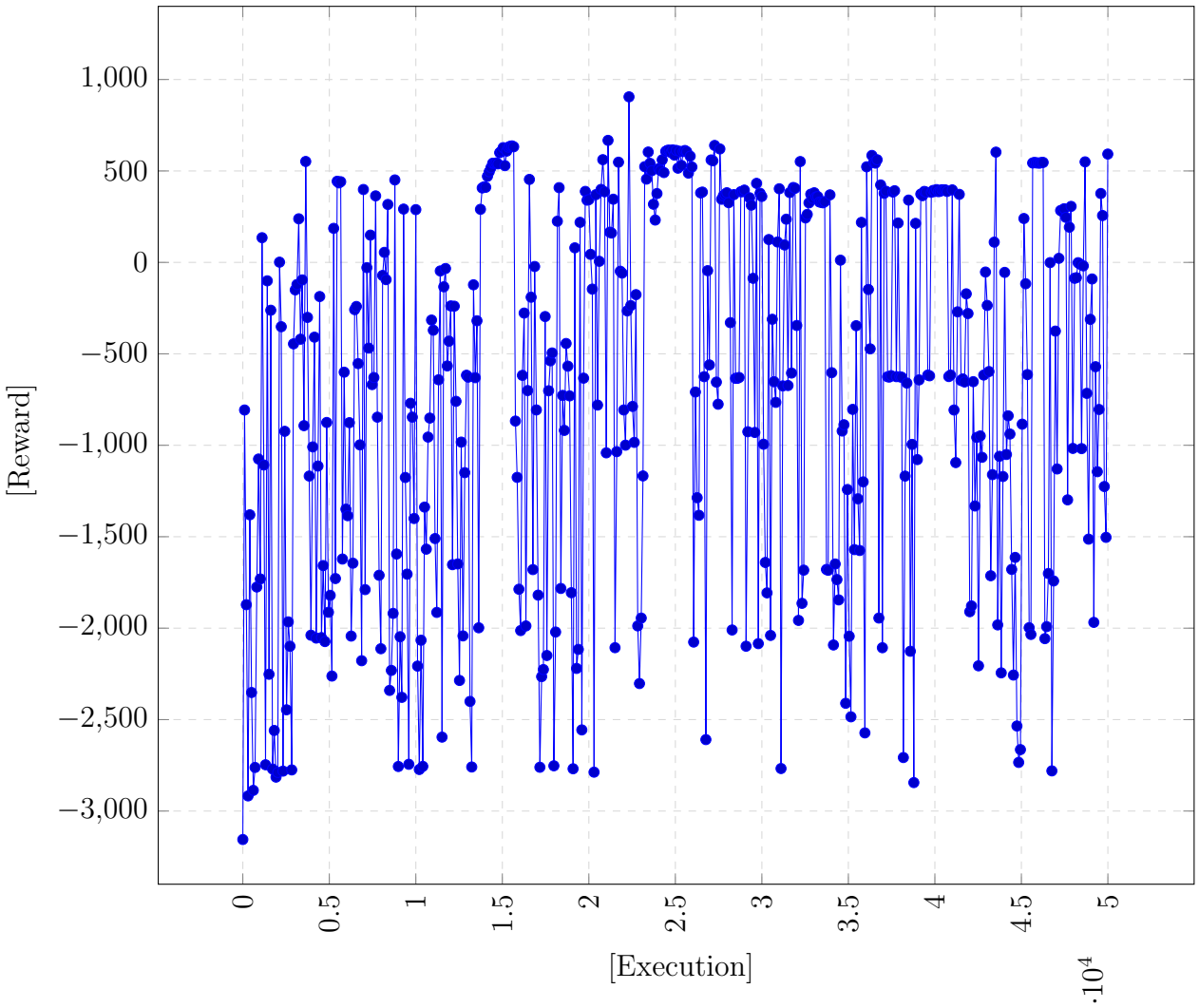


Figure 6.6: **rPlus= 600; alpha = 1 gamma = 0.5, Rounding factor = 30**

The graph above shows that the walker is a slightly more stable. There seems to be more executions with a reward around 500, as opposed to the previous test. This test showed an improvement in the learning process, but would the agent be able to learn how to stand in a balanced position with its two feet elevated if the decay rate was even higher?

Test 3

In the third test we have set the decay rate to its maximum $\gamma = 1$, and based on the test before we hoped the agent would learn to keep its feet elevated. The graph below shows

evidently that the agent is in fact able to stabilize its reward.

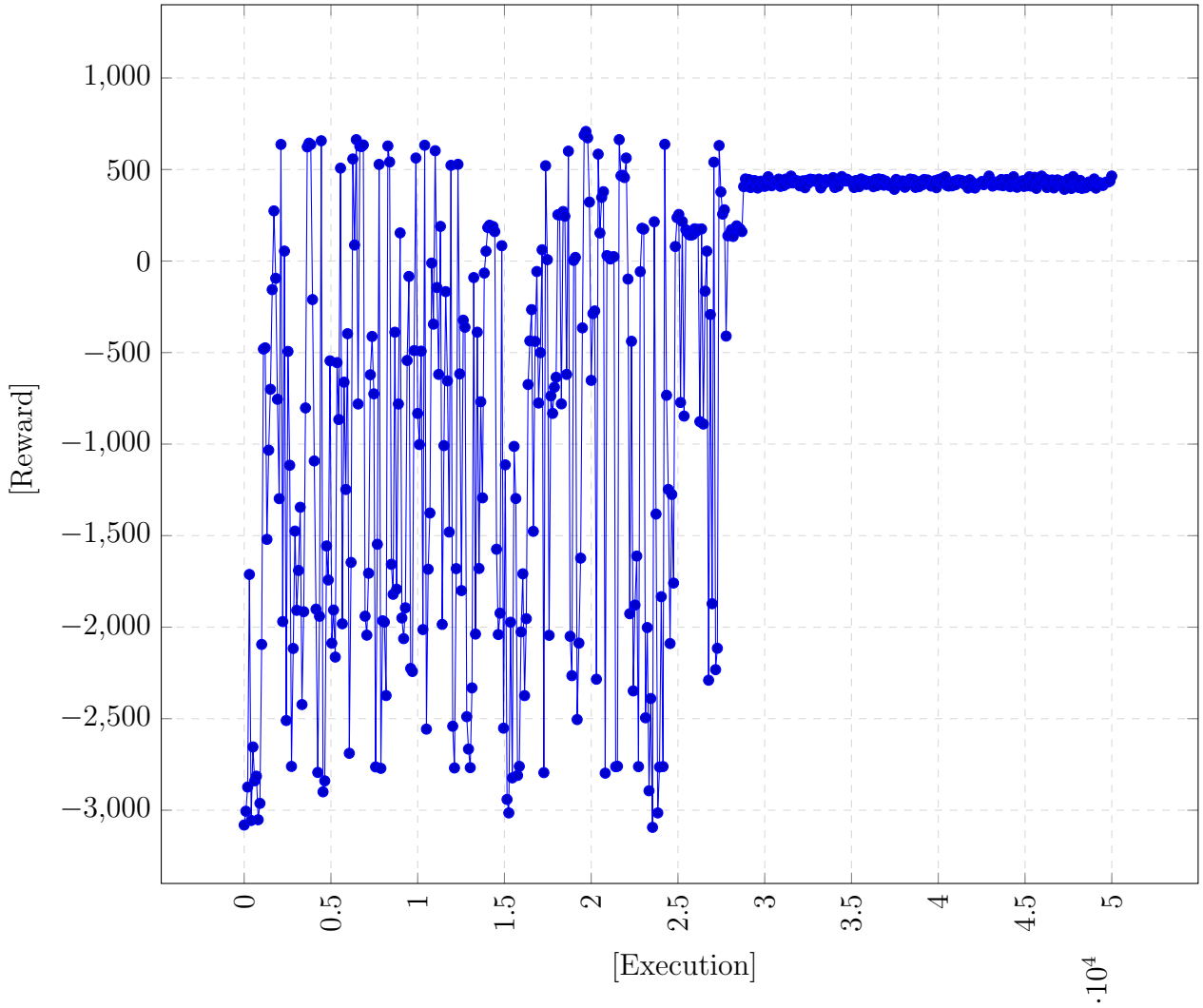


Figure 6.7: **Rplus= 600; alpha = 1 gamma = 1, Rounding factor = 30**

After some initial exploring, the agent finally learns a good policy for keeping its feet elevated at around 27,000 performed actions. It learns which actions, given a certain state give the best reward. The position the walker ended up at, is the one shown in the picture below.

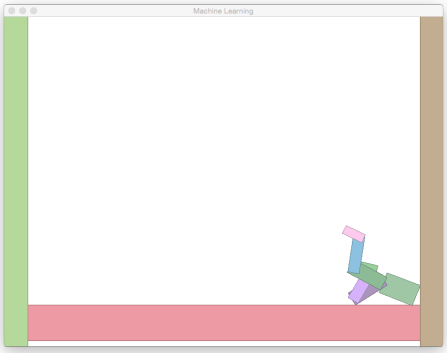


Figure 6.8: Final position for elevated feet in reward mode 1

This gives us indication that the decay rate is important for the learning algorithm. Weighing rewards in the far future as high as current rewards has an impact on the agent. However this might not the case with other reward modes and parametrisation.

Now that we have tried to get the agent to learn two minor complex behaviours, we will test the learning algorithm on a more complex behaviour: to walk.

d Test 3 - Forward motion

We did some preliminary tests prior to deciding on a final testing procedure for reward mode 0. In this mode the walker receives positive reinforcement for moving to the right and negative reward for moving to the left. In addition to this the walker receives a large negative reward for falling and a large negative reward for not moving. This was done in order to encourage the agent to take action and avoid a *lazy walker* agent. This is explained further in the discussion chapter. All in all the code for the *BipedBody.reward*()-method for mode 0 is as follows:

```
if ((Simulation.walker.foot2.getChangeInPosition().x +
    Simulation.walker.foot1.getChangeInPosition().x) > 0) {
                reward = ((Simulation.walker.foot2.getChangeInPosition().x +
                    Simulation.walker.foot1.getChangeInPosition().x) * 5000);
            }
            if ((Simulation.walker.foot2.getChangeInPosition().x +
                Simulation.walker.foot1.getChangeInPosition().x) < 0) {
                reward = ((Simulation.walker.foot2.getChangeInPosition().x +
                    Simulation.walker.foot1.getChangeInPosition().x) * -1000);
            }
            if (Simulation.walker.hasFallen()) {reward = -1000;}

            if((Simulation.walker.foot2.getChangeInPosition().x +
                Simulation.walker.foot1.getChangeInPosition().x) == 0){reward
                = - 1000;}
```

The method $getChangeInPosition().x$ is a method for the *Body*-class in the dyn4j-library. This method returns how the body has moved on the x-axis since last simulation step and thereby gives the ability to use a bodys velocity as a variable.

Since the desired behaviour from the agent is more complex in this mode, we initially decided to run these tests in a manner, where we recorded the accumulated reward per generation for 100,000 generations as opposed to 20,000 or 50,000 executed actions. These tests ran for more than four hours but at around 70,000 generations they reached a point, where the performance was so slow that a $agent.execute()$ would take several seconds to compute. The reasons for the these issues with performance are discussed in the discussion chapter. As a result of these limitations, we decided to run tests for 70,000 generations. We did these tests with an unchanged $reward()$-method, where the only parametrisation was learning rate α and decay rate γ in order to see how these affected the performance of the agent.

Test 1

In the first test we set $\alpha = 0.5$ and $\gamma = 0.2$. This was done in attempt to encourage the agent to explore while setting a very low impact of future rewards. The 70,000 generations can be seen in the following graph:

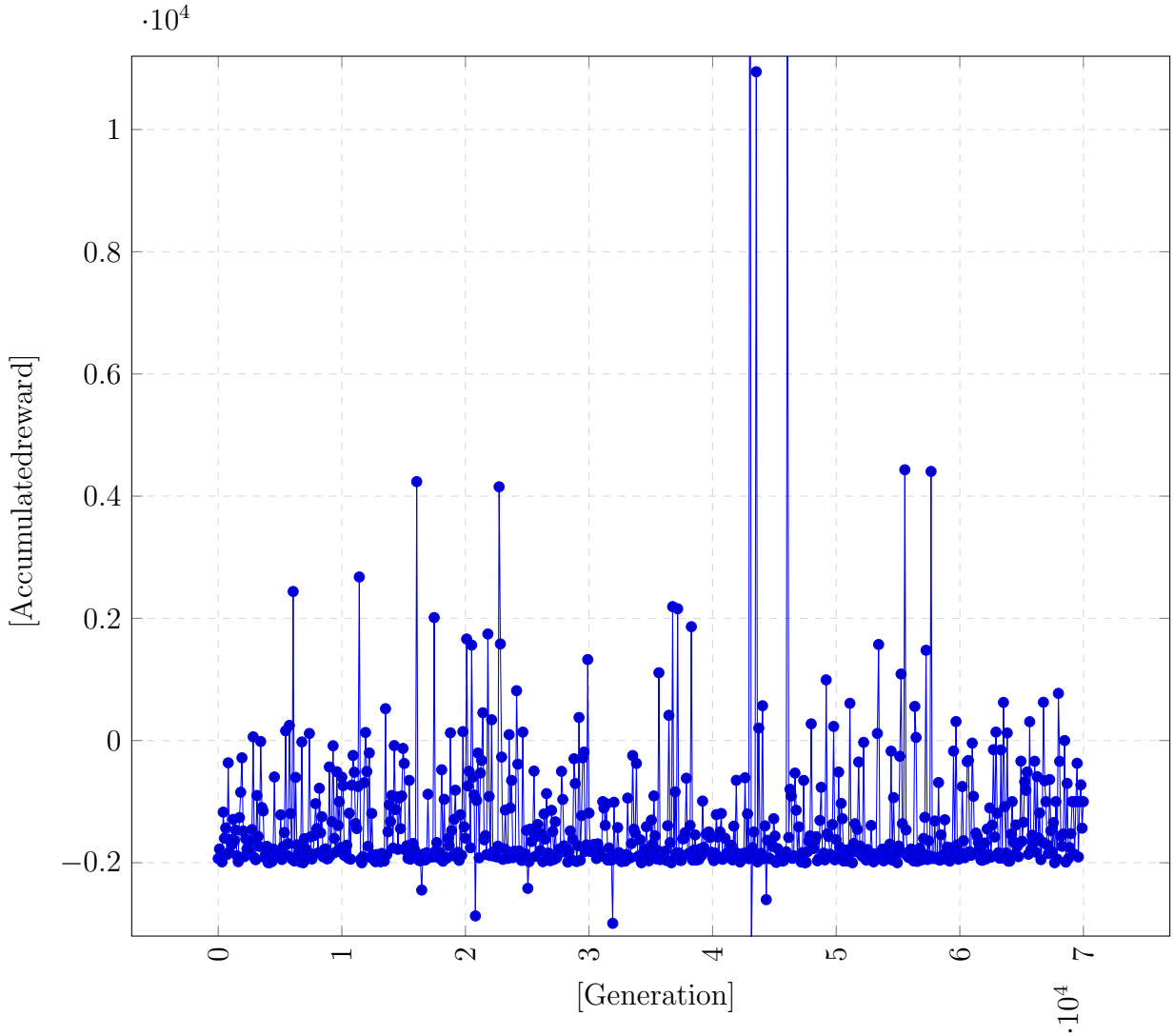


Figure 6.9: Mode 0, $\alpha = 0.5$ and $\gamma = 0.2$ running for 70,000 generations

As it can be seen above, there seems to be no consistent improvement during the 70,000 generations. The reward returned seems to vary quite a lot, which could either indicate that the agent is unable to find a satisfactory pattern in actions or simply explores too much. While there are generations, where the agent is able to have a higher accumulated reward this seems to be accidental, since it is unable to repeat this pattern in the subsequent generations.

Test 2

In the second test we therefore tried running with a high learning rate with $\alpha = 1.0$ and and an increased decay rate $\gamma = 0.5$. This was done in an attempt to have the agent learn more from its experiences and weigh future estimated utility higher in the decision making. Since the walker received negative reinforcement the hope here was to, over time, stimulate a more cautious behaviour for the agent where it would try to avoid falling, as

this gave negative reinforcement. The accumulated reward per generation for the 70,000 generations went as shown in the graph below:

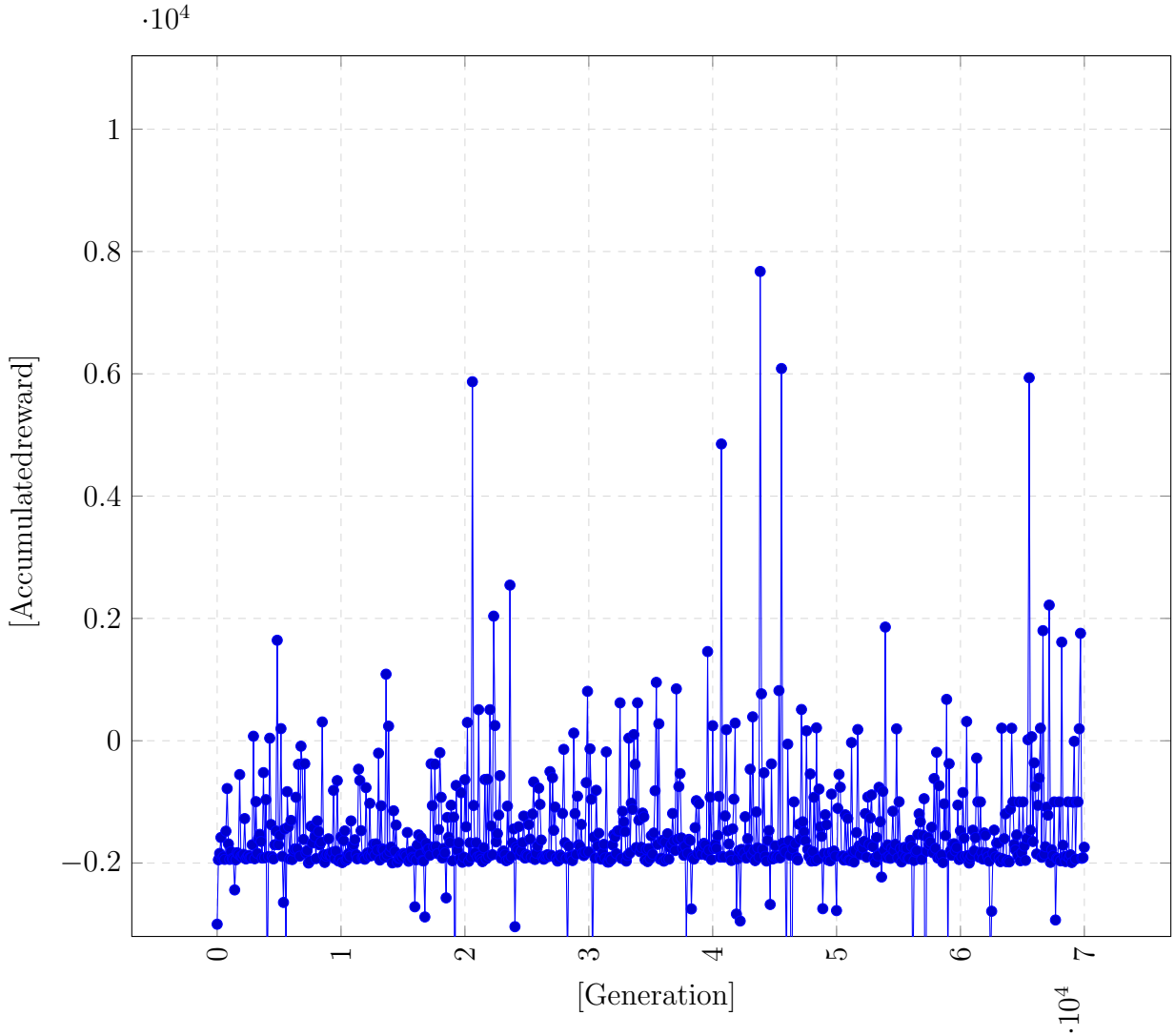


Figure 6.10: Mode 0, $\alpha = 1.0$ and $\gamma = 0.5$ running for 70,000 actions

It would require quite a trained eye to spot the difference this graph and the one from the first test. There is a slightly higher tendency to have an accumulated reward larger than 0, but apart from that, there is not noticeable improvement in the performance.

Test 3

In the third final test we tried increasing the decay rate even further to $\gamma = 0.8$, while keeping the learning rate $\alpha = 1.0$. In the test above the high learning rate did not seem to have a negative influence on the agent.

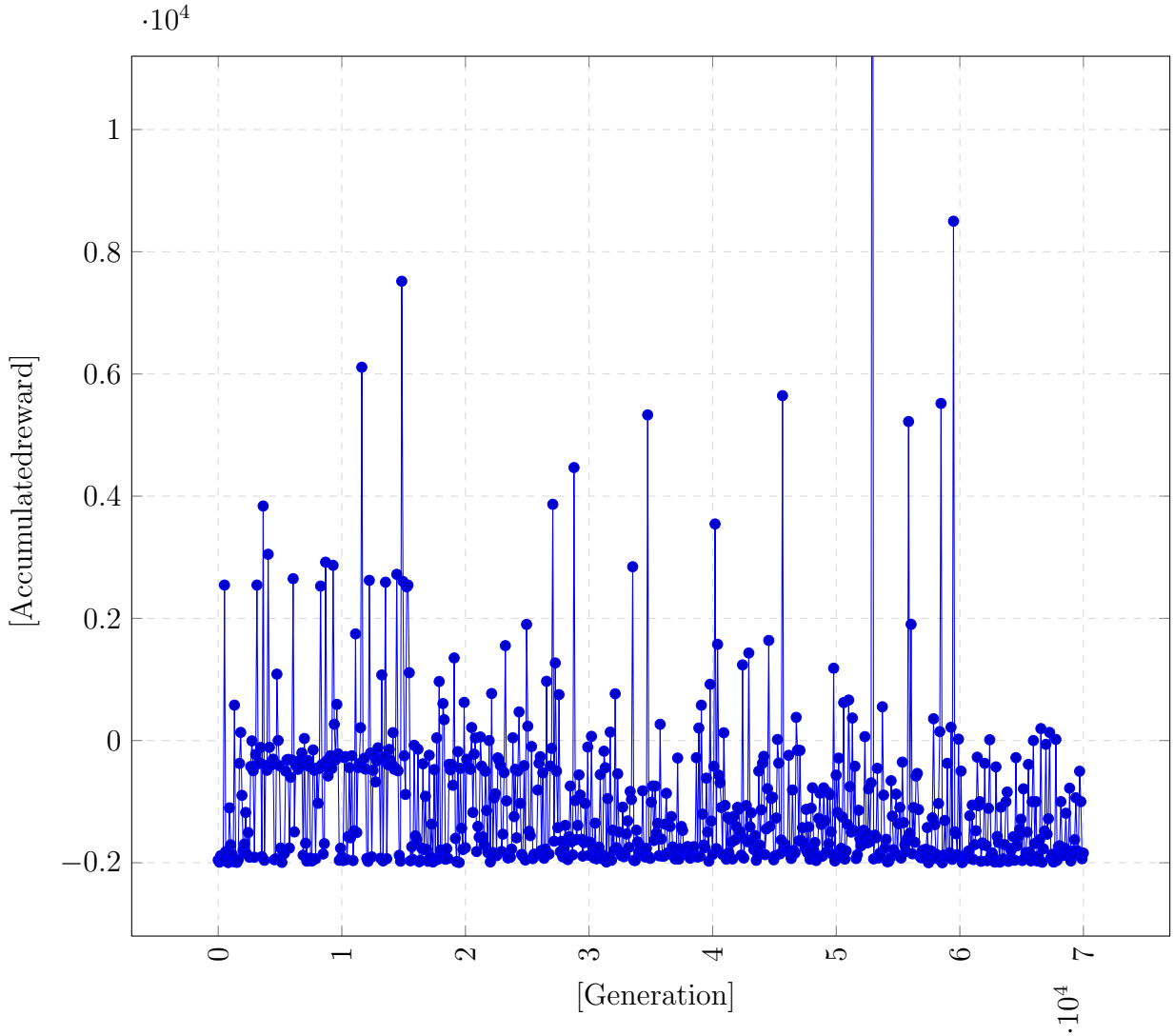


Figure 6.11: Mode 0, $\alpha = 1.0$ and $\gamma = 0.8$ running for 70,000 actions

In this test the walker still was not at all able to start walking or even take several successive steps, but there was some improvement compared to the first and second test. In the third test the accumulated reward per. generation is still quite low, but there is an noticeably larger number of generations that have an accumulated reward $> 0.2{*}10^4$. This could be due to a more cautious behaviour, where the agent becomes more conservative given that a future negative reinforcement has a larger impact with a decay rate of $\gamma = 0.8$ as opposed to $\gamma = 0.5$ or $\gamma = 0.2$.

Final thoughts on forward motion

While it does seems as though there is improvement this has not really been noticeable while doing the tests themselves, since it only came into attention when looking at the data recorded from the test. The third test could however indicate that there would be an idea in doing further testing with an even higher decay rate, since this seemed to have a

positive reaction. All in all the agent is far from being able to take successive steps, which could indicate that some parts of the learning algorithm are flawed. A better parametrisation of the agents parameters could yield interesting results.

Another thought could be that 70,000 generations is simply too few for our learning algorithm to find a successful pattern of actions resulting forward movement. We have, however, had some performance issues, which would have made it very time-consuming to try 500,000 generations or more. These performance issues are discussed in the discussion chapter.

e Conclusion

We did tests for the three types of reward modes. The first two, bending of the knee and elevation of the feet, showed that the agent is capable of learning which state-action pairs have high utility and even manages to stay at a stable level of reward-per-action.
Throughout the testing, it became more or less clear that the complexity of tasks available for the agent had a significant effect on its ability to learn. Parametrisation of the learning rate α to a higher value, enabled the agent to stabilize faster than at lower values. Experimentation with the decay rate γ showed that, weighing future rewards higher had a positive effect on learning.
When testing the mode pertaining to forward motion, the agent was not able to find a successful pattern of state-action pairs. When looking closely at the data provided from the test, there seemed to be some variation when parametrising. We were, however, not able to explore this to its fullest extent due to performance limits.

Chapter 7

Discussion

The main goal of the project was to develop an algorithm that made it possible for a biped walker to learn how to walk without having any kind of prior knowledge. To some extent we were able to develop an algorithm that the agent can use for learning. This is demonstrated in the testing chapter, where we show that the agent is able to learn how to bend its knees or elevate its feet in unison with the associated reward. Although we can conclude that learning is done to some extent, walking is not something the agent has learned yet, based on the tests we have made.

In the following chapter we will discuss how and to what extent we were able to achieve the goal we set out the reach, what could have been made different and present some reflections on how to solve the problems at hand.

a Complexity and a large number of States

It is important for a state to be quite accurate and well-defined, when working with MDP's. A state needs to be well defined and precise, so that it can be represented, in the best possible way. But, if no kind of function approximation was used, the algorithm would have an infinite number of states, since even the smallest variation in the features that define a state, would cause the given state to be considered unexplored by the learning agent. When working with a relatively complex environment, such as the one in our physics simulation, function approximation is therefore essential if the Q-learning algorithm is supposed to start learning within a reasonable amount of time.

We tackled this problem by rounding the features that define a state with an adjustable rounding factor. With the rounding factor set to the minimum value allowed by the program, the number of states is at $282,268,800$, which is nonetheless an enormous amount of states.

a.1 Approaches to function approximation

Our method for function approximation is rather rough, since it dictates that precision, in defining the angles for each joint in a state, is equally important for all joints, since

they are all rounded by the same factor. One could imagine that precision, or a larger possible interval, would be more important to the hips than the ankles, since the angle of the hips have an influence on the position of the ankles and not vice-versa.

Other approaches to function approximation could be less boolean-focused approaches for deciding if a state is known or unknown. If, for example, an unknown state was similar to an already-known state in most ways, the Q-values of this new state could be affected and weighed by the Q-values of the known similar state. The more the two states were alike, the less the agent would be encouraged to do exploration from scratch and instead have some notion of the expected utility of state-action pairs in the new state.

The weighing and function for deciding the expected utility of state-actions pairs in this new state, which had some similarities to an already-known state, could perhaps be learned through another learning algorithm. This learning algorithm would then, over time, need to learn how to determine which of the features in different states that are important when considering whether a new state is completely new and should be explored from scratch, or if there might be some valid guesses as to what the expected utility of state-action pairs might be.

An implementation of such an algorithm would, in theory, mean that the agent no longer considers states as independent. Instead there could be a number of inputs, which would be the walkers sensors, without any rounding factor, and an output which was the expected utility of performing each possible action. The agent would then, through its learning process, learn functions that determines how the input values were expected to influence the output.

In our case, function approximation was, as explained in the beginning of this section, important in order for the agent to be able to define an optimal policy. We found however that setting the appropriate rounding factor was difficult, since this parametrisation involves a fine balance between precision and complexity. With no rounding factor at all, the features of a walker in a state would be precise and therefore the learning rate could be set to a high value, since the agent would weigh the expected reward of the outcome of a state-action pair higher. On the other hand, a high rounding factor would lead the agent to explore the same state more frequently and one could imagine that this would also play a role in determining the expected utility of a state-action pair, since the agent would spend more time updating Q-values instead of exploring.

b Defining actions

Q-learning dictates that utility is not connected to states alone, but to state-action pairs. We have set the number of actions possible for each state to a constant, which is 24 actions - 4 for each joint. The total number of Q-values, with a rounding factor set to 5, can then be calculated as:

$$282,268,800 * 24 = 6,774,451,200 \tag{7.1}$$

This number, however, is the theoretical maximum number of Q-values and it would be unlikely to think that the *agent* would explore them all. For this to happen, the agent would have to always find the updated Q-value having smaller utility than the optimistic estimate *Rplus* and in addition to this, the agent would, maybe by chance, have to find itself in each state enough times to try each action.

We can thereby conclude that the complexity of the agents learning process is not only determined by the number of states, but also by the available actions in each state. In our program the *State*-class has a static list, which contains all possible actions. This means that the same actions are possible in every state.

As mentioned in the section about Markov Decision Processes, this was done as a conceptual idea, since we wanted the agent to learn which actions had outcomes and which actions that did not. We thought it was interesting to keep a lot the information about the walkers body hidden for the agent, since it would then have to learn only from the consequences of its actions. As we progressed further into the development process, this might have turned out to be a rather naive idea, given the difficulties we have faced throughout the development and testing.

b.1 A different way to define actions

While such a method has not been implemented, due to time limitations, there could be a way to easily lower the number of actions available in each state. This could e.g. be done by checking if a joint angle is at its maximum or minimum angle, as allowed by the joint. If this was the case, trying (and failing) to rotate the joint further in that direction should then not be considered a possible action for that state. This approach could also be used if motorOn is false for a joint, as then setting the joints motor off should not be an action.

In the current version of the program there might be an issue with updating a Q-value for state-action pair, where the action is of the type explained above. This would mean that the agent wrongfully learns that there is a high expected utility for doing an action, which in reality does nothing.

Another way of determining an action and the associated expected utility could be through use of some kind of mirroring- or symmetry-method concerning both states and actions. An example: if the agent learns that moving the right knee gives high expected utility if the left foot is at a certain position, wouldn't the same be the case if it was switched and the agent moved its left knee with the right foot at that same position? It is not an entirely simple algorithm to develop, but if it was implemented successfully it would be able to explore state-action pairs significantly faster.

c Performance issues

All this discussion, of optimising actions and minimising the number of state-actions pairs, stems from issues experienced when running the simulation for a prolonged amount of time. Among these issues were unresponsiveness in the program, latency and in the run speed. We are not sure what the reasons behind these issues are, but we suspect that they are related to a lack of computing power or an inefficient program, maybe even both. Since there seemed to be a correlation in the decrease of run speed and the number state-action pairs, the bottleneck for the entire programs' computation could be in the HashMap that contains the Q-values for the state-action pairs.

Searching a HashMap could, in the worst cases, have a execution time of $O(N)$, where N is the number of entries in the HashMap. Because of the large number of states, N would then be increasing and hereby make the program run slower and slower. We have run tests for 70,000 generations, which took 4.5 hours and ended up using more than 4 GB of RAM, which could indicate that this might be the case. A way of solving this problem could possibly be to use a more sophisticated $hashCode$-method, though we haven't made enough performance tests to know whether or not this is the case.

c.1 Testing and simulation

The problems with computing power might have been an issue, when it comes to evaluating the algorithms ability to teach a agent to walk. We don't know for sure, if the bipedal body would start walking after certain number of generations, since we quite simply have not been able to make it simulate that far, due to said lack of computing power. At maximum load, we have been able to simulate at 150x simulation speed while retaining acceptable responsiveness. If we were to fully test the capabilities of the learning algorithm, this might have had to be done for a prolonged time at a much higher simulation speed.

We could also have experimented with executing the learning algorithm without doing graphical rendering, which might have made it faster to compute. This would require a functionality, where the user was able to turn rendering off. This would require that the simulation was put in a separate thread from the GUI in order for the program to remain responsive, while it was simulating in the background. Dyn4j has some limits on multithreading, so this might have been difficult to achieve using that particular physics library.

c.2 Parametrisation

To deal with the above-mentioned limitations, we often set quite a high learning rate for the learning agent. This was done so the agent would weigh the updated Q-value higher, since there was such a high number of state-action pairs. In addition to this, the final implementation of the exploration function was not prioritised highly, since we figured that the agent already spent a large amount time exploring every action in every state.

This could be one of the reasons why the biped walker has yet to start successfully walking, since it might not use the necessary amount of time exploring and very quickly starts exploiting using a greedy approach, where it always takes the action with the highest expected utility instead of exploring. Walking involves making several decisions that are sub-optimal and therefore would require more exploration.

This could also be the one of the reasons why the learning algorithm is quite successful in reward mode 1 (elevated feet) and 2 (bended knee), where the behaviour does not have the necessity for taking sub-optimal actions. In taking sub-optimal actions, the decay rate γ also plays a big role, since it values future rewards as opposed to instant reward. One could therefore assume that walking would require much more dependence on future rewards and as such, need a higher decay rate when updating Q-values. During the development, we found it difficult to figure out and test what decay rate was optimal for a certain behaviour and a lot of the resulting parametrisation was consequently done through experimentation.

This was also the case for giving rewards, where the rewards for elevated feet and bended knee were pretty self-explanatory while the reward for forward motion was far more complex. We ended up rewarding the walker for moving its feet in the right direction and the higher the speed, the higher a reward was returned from the $reward()$-method. The reason for this was to motivate the agent to move its feet, since we figured this plays the largest role in forward motion. However, we tried not to restrict the agent too much by e.g. rewarding the agent for keeping the body upright, since we wanted the agent to find its own optimal manner of moving forward. We did, on the other hand, give negative reinforcement to the agent when it moved in the wrong direction and when it fell. Ideally, we thought that the agent would learn which state-action pairs that lead to falling and would then slowly conclude that these state-action did not have a high utility. Throughout development we changed this approach and started dictating that wanted behaviour from the agent to a higher degree.

We had a rather peculiar issue with an agent that learned that *laziness* pays off. After a certain number of iterations the agent learned that not doing anything from the initial position was the optimal policy because of the negative reward for going in the wrong direction or falling. This was solved by giving negative reinforcement for not moving in order to encourage the walker to attempt forward movement in all states.

d Final thoughts on the process

One of our initial motivations for working with a simulation of a biped walker, when exploring the concept of AI, was that we thought it would be fun to work with a *"clumsy"* 2D-model, where, even if it failed completely, it would hold some comic value to the project. When working with such a physics simulation we, the developers, are not sure of the optimal style of forward motion and therefore have not had an initial goal for the precise movement scheme of the biped walker. Working with a simulation and having to optimize a learning algorithm based on a 2D-rendering, has also proved to be very

difficult, since a lot of the decisions have been made on interpretations of what happened on the screen as opposed being able to analyse data through numbers.

The concept of learning solely through reinforcement is interesting, since the agent has no idea of the concept of walking, forward movement or the physics environment, but only learns from the outcome of its actions. While this at first was a charming idea, we quickly became aware of the complexity of this concept and knowing this, would have changed the initial workings of the learning algorithm to try and accommodate this.

As a learning process the development of this walker has been extremely rewarding. None of the group members had previously had any real experience with working with a medium-sized object-oriented program and absolutely no experience in working with physics modelling or developing and nurturing an artificial intelligence. While the concept of the program might have turned out to be too ambitious given the our programming skill-level, this has not only been a hindrance but also a motivating factor.

e Last-minute changes

During the final hours of writing, correcting and compiling we have found an error in the code, which might have been the cause of a lot of the performance and learning related issues which we have been facing. While we have not had time to do extensive testing or rewrite any chapters, we will, in the following section, describe the source of this performance related issue and how we managed to make a solution to this problem

e.1 HashCode()-method in *State*-class

The hashCode method in Java returns an integer based on the instantiation of a class' location in memory. We did originally override this, since the state was not supposed to be a specific state located in memory, but instead be an approximate state as described in the section on function approximation. During the final testing we found that the size of the Q hashMap in the *Agent*-class, was able to increase in size and become larger than the theoretical maximum of Q-values. This was calculated on the basis that the theoretical maximum of Q-values for state-action pairs should never be larger than the product of the theoretical maximum number of states and the number of actions available.

This overly large size of the HashMap could indicate that the Q HashMap ends up containing duplicates, which could increase the length of the learning process for the agent, since the amount of exploration would be vastly increased. To locate the problem we tried changing the overridden *hashCode*()-method in the *State*-class to the method seen below:

```
@Override
   public int hashCode() {return 0;}
```

Since all states now have a hashcode-method returning 0, the result is that every time $Q.put(key, value)$ is executed it is now forced to run the *equals*()-method for the *State*-class. Since this method ensures *true* is only returned if the state tested is an approximate state and false in all other cases:

```
    @Override
    public boolean equals(Object o) {
        // equals method is found when collision are found when searching the
            HashMap Q in Agent-class
        State s = (State) o;

        //Checks degree compared to world
        if (Math.round(Math.toDegrees(this.worldAngle) / roundFactor) !=
            Math.round(Math.toDegrees(s.worldAngle) / roundFactor)) {
            return false; // return false if angles do not match
        }

        // Checks degrees of each joints
        for (int i = 0; i < this.jointAngles.size(); i++) {
            if (Math.round(Math.toDegrees(s.jointAngles.get(i) / roundFactor))
                != Math.round(Math.toDegrees(this.jointAngles.get(i) /
```

```
                roundFactor))) {
                return false;// return false if angles do not match
            }
        }
        return true;
    }
```

As a result of *hashCode*() returning 0, the code above is executed for all cases, since a collision in the HashMap is bound to happen, given that all states have the same hashCode. Due to time constraints we have not been able find a more elegant solution to this issue, but it seems to solve the issue of an ever-increasing number of state-action pairs.

e.2 Result of correction in the *State*-class

As mentioned, we have not been able to do any extensive testing on this work-around, since this is only hours before deadline. The fix does however reduce the number of state-action pairs greatly, while retaining the same level of precision for each state. To demonstrate this we ran 1,000 generations in reward mode 0 and these are the screendumps of the results:

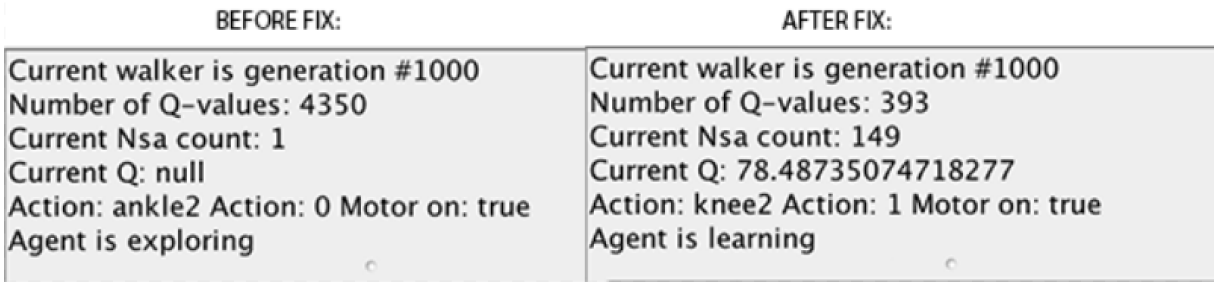


Figure 7.1: GUI screenshot from before and after the fix

These two tests were done with a rounding factor of 15 and as it can be seen above, the fix does reduce the number of Q-values greatly. This gives the agent a significantly higher number off iterations for updates of Q-values, as it is seen in the Nsa count.

It does feel frustrating to find such an issue in the code at this point in time, since it might've been able to change a lot if found earlier. We have, however, chosen to include the unsophisticated fix seen above, since one of the major issues for the learning agent is the enormous amount of states, which forces it to explore a lot. By doing this simple fix we can lower this number greatly, but might have worsened the performance in doing so. Note that since this is a last-minute change, we have included this fix in the source code and the program provided, but haven't had time include these new changes in the discussion and other chapters.

Chapter 8

Conclusion

We were interested in researching and understanding the main concepts of reinforcement learning. The idea behind developing an algorithm that made a 2D biped walker able to learn how to walk through experience, without some kind of prior knowledge, seemed interesting and relevant for the general understanding of reinforcement learning. Therefore we based this project on the following research question:

How can a bipedal body learn forward movement, within a simulated 2D physics environment, using a Q-learning algorithm?

Working with Q-learning involves solving Markov Decisions Processes, where states and actions need to be well defined in order for the agent to utilize the experience it receives from its environment. They need to represent the actual state and action that the agent is currently in. This leads to a problem when working with the biped walker, because there is a infinite amount of states that the Q-learning algorithm needs to handle. Function approximation can be used to handle this problem. Considering the manner we define states, using the position of each joint angle, we approach function approximation by using a round factor, thereby reducing the number of states.

When using a Q-learning algorithm, an agent is required to be implemented. This agent represents the element that learns through iterations and accumulates knowledge in the form of updated Q-values. To update Q-values, states and actions need to be given as information to the agent. This happens in our program by using objects of type State and JointAction. States takes the biped walker as input and returns the approximate state it is in, using the above-mentioned function approximation. JointActions affect the joints in the biped walker and are the same for all states.

To be able to build a biped walker and see it perform, a body and a simulation environment needed to be implemented. In this project, we use the physics simulation library dyn4j for this purpose. In addition, we have based our implementation on an example taken from dyn4j website, making some changes to meet the project needs. A GUI is implemented to make it possible to see the development of the learning agent based on values and not only on the 2D biped walker itself. The GUI also enables manipulation of the simulation speed. This is done to speed the learning process since learning in this

environment is a time-consuming task.

To test the learning algorithm we have made various tests. In these tests, parameters associated to the Q-learning algorithm were set to different values as well as the rounding factor. Considering the complexity of walking, some less complex tests were made before forward motion was attempted. They show that the agent was able to learn when the task was to bend its knee and lifting its feet. Learning rate and decay rate showed to significantly influence the way the agent learned. A high learning sped up the learning. Although this was the case in the less complex tests, to some degree the same did not apply for the task of forward motion. Here, the agent was not able to find a pattern for walking; this can be due to the complexity of the task or the way we parameterised the variables for the agent. In addition to this, the time consuming element of this task can have had an influence on the obtained results. Considering that the agent was able to learn the less complex tasks and the small amount of improvement during the test for forward motion, we can presume that if we ran the test for a longer time the agent might be able to learn how to acquire a pattern in state-action pairs, which would lead to forward motion eventually.

Chapter 9

Bibliography

AIMA Github. *https://github.com/aima-java*. Visited on May 19 2015.

dyn4j website. *http://www.dyn4j.org*. Visited on May 19 2015.

Genetic Algorithm Walkers. *http://rednuht.org/genetic_walkers/*. Visited on May 10 2015.

Learn About Java Technology. *http://java.com/en/about/* . Visited on May 19 2015.

Lee, Mark (2005). 6.5 Q-Learning: Off-Policy TD Control. *http://webdocs.cs.ualberta.ca/ sutton/book/ebook/node65.html*

Murphy, Daniel (2014).JBox2D: A Java Physics Engine *http://www.jbox2d.org* . Visited on May 19 2015.

Russel, S. and Norvig, P. (1995) Artificial Intelligence A modern Approach. Second Edition. Chapter. 1, 17 ,21. New Jersey: Pearson Education.

```java
package sample;

import javax.swing.*;

import QLearning.JointAction;
import Rendering_dyn4j.Simulation;
import Rendering_dyn4j.ThreadSync;

import java.awt.*;

/*
This class contains the components and method for the graphical user interface.
All actions listeners are synchronized to ThreadSync. Refer to the written report for more details on this.
 */

public class GUI {
    public static HighScoreTable highScoreTable = new HighScoreTable(); // JTable containing information for each generation
    private static JLabel generationLabel; //Label for no. of generation
    private static JLabel statesLabel; // Label for total number of states
    private static JLabel currentNsa; // Label for state-action frequency for current state-action pair
    private static JLabel currentQ; // Label for Q-value of current state-action pair
    private static JLabel agentStatus; // Label indicating if agent is exploring or learning
    private static JLabel currentAction; // Label for current action

    public GUI(Simulation world)  {
        JFrame gui = new JFrame();
        gui.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        gui.setSize(350, 500);
        gui.setLocation(820, 0);
        gui.setTitle("Controls");

        //Panel for information about current walker
        JPanel currentBipedPanel = new JPanel();
        currentBipedPanel.setLayout(new GridLayout(0, 1));
        generationLabel = new JLabel("Current walker is generation #" + Main.generation);
        statesLabel = new JLabel("Number of Q-values: " + 0);
        currentNsa = new JLabel("Current Nsa count: " + 0);
        currentQ = new JLabel("Current Q: " + 0);
        agentStatus = new JLabel("Agent is exploring");
        currentAction = new JLabel("Action: ");

        // add components
        currentBipedPanel.add(generationLabel);
        currentBipedPanel.add(statesLabel);
        currentBipedPanel.add(currentNsa);
        currentBipedPanel.add(currentQ);
        currentBipedPanel.add(currentAction);
        currentBipedPanel.add(agentStatus);
```

```
49
50        //Panel for controls
51        JPanel control = new JPanel();
52        JButton resetButton = new JButton("Reset walker"); // Button for
    resetting walker to initial position
53        control.setLayout(new GridLayout(0, 1));
54        JToggleButton pauseButton = new JToggleButton("Pause simulation");
    // Button for pausing simulation
55        JLabel simSpeed = new JLabel(Main.simulation.getSimulationSpeed() + "
    x Speed", SwingConstants.CENTER);
56        JSlider simSpeedSlider = new JSlider(1, 50, 1); // Slider for simulation
    speed
57        JButton randomAction = new JButton("Force random action");
58
59        // Add Components to panel
60        control.add(resetButton);
61        control.add(pauseButton);
62        control.add(randomAction);
63        control.add(simSpeed);
64        control.add(simSpeedSlider);
65
66        // Panels for highscore table
67        JScrollPane highScorePane = new JScrollPane(highScoreTable.getTable())
    ;
68
69        // Panels are added to two vertical split panes
70        JSplitPane splitPane1 = new JSplitPane(JSplitPane.VERTICAL_SPLIT,
    currentBipedPanel, control);
71        JSplitPane splitPane2 = new JSplitPane(JSplitPane.VERTICAL_SPLIT,
    splitPane1, highScorePane);
72
73        // Panels are added to frame
74        // highScorePane is only added if the reward mode is "0"
75        if (Main.mode == 0 || Main.mode == 3 || Main.mode == 4) {
76           gui.add(splitPane2);
77        } else {
78           gui.setSize(350, 250);
79           gui.add(splitPane1);
80        }
81        gui.setVisible(true);
82
83        // Actions listeners
84        simSpeedSlider.addChangeListener(e -> {
85           // Slider for changing simulation speed
86           synchronized (ThreadSync.lock) {
87                 Main.simulation.setSimulationSpeed(simSpeedSlider.getValue());
88                 simSpeed.setText(Main.simulation.getSimulationSpeed() + " x
    Speed");
89           }
90        });
91
92        resetButton.addActionListener(e1 -> {
93           // Action listener for resetting position of walker
94           synchronized (ThreadSync.lock) {
95              Main.generation++;
```

```
 96              Simulation.walker.resetPosition();
 97              update();
 98            }
 99        });
100
101        pauseButton.addChangeListener(e1 -> {
102            //Action Listener for pausing or resuming simulation
103            synchronized (ThreadSync.lock) {
104                if (Main.simulation.getSimulationSpeed() != 0) {
105                    Main.simulation.setSimulationSpeed(0);
106                } else {
107                    Main.simulation.setSimulationSpeed(simSpeedSlider.getValue());
108                }
109            }
110        });
111
112        randomAction.addActionListener(e -> {
113            // In cases where the bi-ped is to lazy to explore, one can force a
    random action by using this button
114            JointAction random = Simulation.walker.getState().getRandomAction()
    ;
115            random.doAction();
116            update(random);
117        });
118
119    }
120
121    public void update(int Nsa, double Q) {
122        // Method for updating gui with current Nsa count and Q-value
123        synchronized (ThreadSync.lock) {
124            currentNsa.setText("Current Nsa count: " + Nsa);
125            statesLabel.setText("Number of Q-values: " + Main.agent.getQsize()
    );
126            if (Q == 0.0) {
127                currentQ.setText("Current Q: " + null);
128                agentStatus.setText("Agent is exploring");
129            } else {
130                currentQ.setText("Current Q: " + Q);
131                agentStatus.setText("Agent is learning");
132            }
133        }
134    }
135
136    public void update() {
137        // Overloaded method for updating gui labels
138        synchronized (ThreadSync.lock) {
139            generationLabel.setText("Current walker is generation #" + Main.
    generation);
140        }
141    }
142
143    public void update(JointAction action) {
144        // Overloaded method for updating gui with label for current action
145        synchronized (ThreadSync.lock) {
146            currentAction.setText("Action: " + action.toString());
```

```
147         }
148     }
149 }
150
```

```java
package sample;

import QLearning.*;
import Rendering_dyn4j.Simulation;
import Rendering_dyn4j.ThreadSync;

/*
This is the Main class for the BiPed learner.
 */

public class Main {
    public static int generation = 0;
    public static Simulation simulation;
    public static GUI gui;
    public static Agent agent;
    public static double accumulatedReward;
    public static State initState;
    public static JointAction initAction;
    public static int mode = 0;

    public static void main(String[] args) {
        StartDialog dialog = new StartDialog(); // Starting dialog window
        learn();
    }

    public static void learn() {
        // Setup
        simulation = new Simulation();
        gui = new GUI(simulation);

        State.fillActions(); //Add actions to static ArrayList in State

        //First action
        initAction = Simulation.walker.getState().getRandomAction();
        initAction.doAction();

        agent = new Agent(mode); // Agent is created based on chosen reward mode
        initState = Simulation.walker.getState();

        while (true) {// Loops as long as program is running
            accumulatedReward = 0;
            double t = 0;
            boolean isTerminal = false;

            while (!isTerminal) {
                if (!Simulation.walker.isInSight()) { // Reset if out of sight
                    isTerminal = true;
                }
                if (t > 400000) {
                    // Observe and execute
                    JointAction action = agent.execute();
                    if (action != null) {
                        synchronized (ThreadSync.lock) {
                            action.doAction();
```

```
55                  }
56              } else {// If null is returned, agent is at a terminal state
57                  isTerminal = true;
58              }
59              t = 0; // Reset time to zero
60          }
61          t += simulation.getElapsedTime(); // Increment time
62      }
63      // When loop is breaked, gui is updated and walker is reset to initial
    position
64      if (isTerminal) {
65          updateGuiTable();
66          Simulation.walker.resetPosition();
67      }
68    }
69  }
70
71  private static void updateGuiTable() {
72    synchronized (ThreadSync.lock) {
73      gui.highScoreTable.add(new Generation(generation,
    accumulatedReward));
74      generation++;
75      gui.update();
76    }
77  }
78 }
79
80
81
82
```

```java
package sample;

/*
This class is for creating generation objects that saves information. These are shown in the table in the GUI.
 */

public class Generation  {
    double accumulatedReward;
    int generationNumber;
    public Generation(int generationNumber, double accumulatedReward) {
        // This constructor takes information from each ended iteration of biped walkers
        this.generationNumber = generationNumber;
        this.accumulatedReward = accumulatedReward;
    }
}
```

```java
package sample;

import QLearning.State;
import javax.swing.*;
import java.awt.*;

/*
This class is for the opening dialog window which is instantiated at runtime.
In this dialog the user can choose mode and adjust rounding factor.
 */


public class StartDialog {
    private JDialog dialog = new JDialog();
    String[] rewardModes = {"Reward for forward motion", "Reward for elevated feet", "Reward for bending one knee"};

    public StartDialog() {
        JPanel startPanel = new JPanel();
        startPanel.setLayout(new GridLayout(0, 1));
        JLabel chooseModeLabel = new JLabel("Choose mode:", SwingConstants.CENTER);
        JComboBox modeComboBox = new JComboBox(rewardModes);
        JLabel setRoundFactor = new JLabel("Round factor: " + State.roundFactor,SwingConstants.CENTER);
        JSlider roundFactorSlider = new JSlider(5, 30, 15);
        JLabel noOfStatesLabel = new JLabel("Max number of states:", SwingConstants.CENTER);
        JLabel noOfStatesNumber = new JLabel(State.getMaxNumberOfStates() + "",SwingConstants.CENTER);
        JButton runSimulationButton = new JButton("Start learning!");

        modeComboBox.setSelectedIndex(1); // mode 1 is default, since this is more impressive than mode 0
        modeComboBox.updateUI();

        roundFactorSlider.addChangeListener(e1 ->
        {
            State.roundFactor = roundFactorSlider.getValue();
            setRoundFactor.setText("Round factor: " + State.roundFactor);
            noOfStatesNumber.setText(State.getMaxNumberOfStates() + "");
        });

        runSimulationButton.addActionListener(e -> {
            Main.mode = modeComboBox.getSelectedIndex();
            dialog.dispose();
        });

        // Add to startPanel
        startPanel.add(chooseModeLabel);
        startPanel.add(modeComboBox);
        startPanel.add(setRoundFactor);
        startPanel.add(roundFactorSlider);
        startPanel.add(noOfStatesLabel);
        startPanel.add(noOfStatesNumber);
```

```
50        startPanel.add(runSimulationButton);
51
52        dialog.setDefaultCloseOperation(JDialog.DISPOSE_ON_CLOSE);
53        dialog.setModal(true);
54        dialog.add(startPanel);
55        dialog.pack();
56        dialog.setLocation(400, 200);
57
58        dialog.setTitle("BiPed Learner");
59        dialog.setVisible(true);
60    }
61
62 }
63
```

```java
package sample;

import Rendering_dyn4j.ThreadSync;
import javax.swing.*;
import javax.swing.table.*;

/*
This class contains a list of each generation of walkers. It is only added to the gui if the mode is set to reward for forward motion.
The class contains a JTable and a method for adding a new generation of walkers to the table after each iteration.
 */

public class HighScoreTable {
    private Object[][] data;
    private Object[] columns = {"Generation #", "Total reward"}; // TODO FJERN NUMBER OF NEW STATES
    private DefaultTableModel model = new DefaultTableModel(data,columns) {
        @Override // This method is overriden to ensure correction sorting in the table
        public Class getColumnClass(int column) {
            switch (column) {
                case 0:
                    return Integer.class;
                case 1:
                    return Double.class;
                default:
                    return String.class;
            }
        }
    };
    private JTable table = new JTable(model);
    public HighScoreTable() {
        table.setAutoCreateRowSorter(true); // Table is automatically sorted
    }

    public void add(Generation g) {
        synchronized (ThreadSync.lock) { // The method for adding rows is synchronized to the threadsync object
            model.addRow(new Object[]{new Integer(g.generationNumber), new Double(g.accumulatedReward)});
        }
    }
    public JTable getTable() {
        return table;
    }


}
```

```java
package QLearning;

import Rendering_dyn4j.Simulation;
import aima.core.util.FrequencyCounter;
import aima.core.util.datastructure.Pair;
import sample.Main;

import java.util.Collections;
import java.util.HashMap;
import java.util.Map;

/*
The Agent class is the learning agent. The code seen here is based largely on the one QLearningAgent included in the AIMA
implementations of algorithms. This can be found on the AIMA github (https://github.com/aima-java/aima-java). The purpose of
this class is contain Q-values, update these and around an action connected to the optimal policy for a given state.
 */


public class Agent {
    private JointAction noneAction = new JointAction(); // A "none"-action
    private double alpha; // Learning rate
    private double gamma; // Decay rate
    private double Rplus; // Optimistic reward prediction
    private int mode; // Reward mode for the walker
    private State s = null; // S (previous State)
    private JointAction a = null; // A (previous action)
    private Double r = null; // Reward

    private int Ne = 1; // Variable used in exploration function
    private FrequencyCounter<Pair<State, JointAction>> Nsa = new FrequencyCounter<>();
    private Map<Pair<State, JointAction>, Double> Q = new HashMap<>();

    public Agent(int mode) { // Constructor with mode as input argument
        this.mode = mode;
        switch (mode) { // Value for Rplus, gamma and alpha depend on mode.
            case 0:
                this.Rplus = 250;
                this.gamma = 0.2; // Lots of reliance of future reward - decay rate
                this.alpha = 0.5; // Large number of states = high learning rate - learning rate
                break;
            case 1:
                this.Rplus = 500;
                this.gamma = 0.2;
                this.alpha = 1;
                break;
            case 2:
                this.Rplus = 150;
                this.gamma = 0.1;
                this.alpha = 0.1;
                this.Ne = 1;
```

```
51              break;
52          }
53          this.s = Simulation.walker.getState(); // State is set to initial state for
    walker
54          this.a = Main.initAction; // Initial action
55      }
56
57      public JointAction execute() {
58          // This method returns a JointAction from the optimal policy
59          State sPrime = Simulation.walker.getState();
60          double rPrime = Simulation.walker.reward();
61
62          // if terminal
63          if (isTerminal()) {
64              Q.put(new Pair<>(sPrime, noneAction), rPrime);
65          }
66          // If State s not null
67          if (s != null) {
68              Pair<State, JointAction> sa = new Pair<>(s, a);
69              Nsa.incrementFor(sa);// Increment frequencies
70
71              // Get Q-value
72              Double Qsa = Q.get(sa);
73              if (Qsa == null) {
74                  Qsa = 0.0;
75              }
76
77              Main.gui.update(Nsa.getCount(sa), Qsa); // Update gui with info for
    current actions
78
79              r = Simulation.walker.reward();
80              Q.put(sa, Qsa + alpha * (r + gamma * maxAPrime(sPrime) - Qsa));
81          }
82
83          if (isTerminal()) { // if terminal, station, action and reward are set
    accordingly
84              s = null;
85              a = null;
86              r = null;
87          } else {
88              this.s = sPrime;
89              this.a = argmaxAPrime(sPrime);
90              this.r = rPrime;
91          }
92
93          if (a != null) {
94              Main.gui.update(a); // GUI is updated with action
95          }
96          return a; // JointAction is returned
97      }
98
99      private JointAction argmaxAPrime(State sPrime) {
100         JointAction a = null;
101         Collections.shuffle(sPrime.getActions()); // Shuffle list for random
    actions
```

```
102       double max = Double.NEGATIVE_INFINITY;
103       for (JointAction aPrime : sPrime.getActions()) {
104          Pair<State, JointAction> sPrimeAPrime = new Pair<State, JointAction
    >(sPrime, aPrime);
105          double explorationValue = f(Q.get(sPrimeAPrime), Nsa.getCount(
    sPrimeAPrime));
106          if (explorationValue > max) {
107             max = explorationValue;
108             a = aPrime;
109          }
110       }
111       return a; // returns a with highest expected utility
112    }
113
114    private boolean isTerminal() {
115       // Falling is only a terminal state in mode 0
116       if (mode == 0) {
117          return Simulation.walker.hasFallen() || !Simulation.walker.isInSight();
118       }
119       return false;
120    }
121
122    protected double f(Double u, int n) {
123       // A Simple definition of f(u, n):
124       if (null == u || n < Ne) {
125          return Rplus;
126       }
127       return u;
128    }
129
130    private double maxAPrime(State sPrime) {
131       double max = Double.NEGATIVE_INFINITY;
132       if (sPrime.getActions().size() == 0) {
133          // a terminal state
134          max = Q.get(new Pair<State, JointAction>(sPrime, noneAction));
135       } else {
136          for (JointAction aPrime : sPrime.getActions()) {
137             Double Q_sPrimeAPrime = Q.get(new Pair<State, JointAction>(
    sPrime, aPrime));
138             if (null != Q_sPrimeAPrime && Q_sPrimeAPrime > max) {
139                max = Q_sPrimeAPrime;
140             }
141          }
142       }
143       if (max == Double.NEGATIVE_INFINITY) {
144          // Assign 0 as the Q if no max has been found
145          max = 0.0;
146       }
147       return max;
148    }
149
150    public int getQsize() {
151       return Q.size(); // Size of the Q HashMap
152    }
153 }
```

```java
package QLearning;

import Rendering_dyn4j.BipedBody;
import Rendering_dyn4j.Simulation;
import org.dyn4j.dynamics.joint.RevoluteJoint;

import java.util.*;

/*
This class is for states. It takes BipedBody as an input and creates an approximate state from the values of the BipedBody joints.
These variables are then rounded in order to create an approximate state.
 */

public class State {
    private int worldAngle; // Torso angle relative to World
    private ArrayList<Integer> jointAngles = new ArrayList<>(); // Angles of joints
    private static ArrayList<JointAction> actions = new ArrayList<>(); // List containing possible actions.
    public static int roundFactor = 15; // Round factor - the higher the number the lower precision in states

    public State(BipedBody walker) {
        // Constructor for a new state with the walker as input.
        // The jointangles from the walker are rounded and then added to the state in order to make the state approximate of the "actual" state
        for (RevoluteJoint j : walker.joints) {
            jointAngles.add((int) (Math.round(Math.toDegrees(j.getJointAngle()) / roundFactor)));
        }
        this.worldAngle = (int) (Math.round(Math.toDegrees(walker.getRelativeAngle()) / roundFactor));
    }

    public static void fillActions() {
        // This methods creates actions for all joints.
        for (RevoluteJoint joint : BipedBody.joints) {
            actions.add(new JointAction(joint)); // New relaxed action (!motorOn)
            for (int i = -1; i <= 1; i++) { // Loop that creates three actions for increase, decrease and "lock" joint
                actions.add(new JointAction(joint, i));
            }
        }
    }

    @Override
    public int hashCode() {
        // LAST-MINUTE CHANGES:
        // Originally the hashcode was returned from this states toString, which contains the angles of the state
        // It now returns and 0, which is an expensive solution, but it seems to optimize performance.
        // We have this explained more in-depth in the section "Last-minute changes" in the written report.
```

```
45
46  //       return toString().hashCode();
47
48      return 0;
49    }
50
51    @Override
52    public String toString() {
53      // Tostring methods returns a string containing the rounded angles of
    the state
54      String s = "";
55      for (double angle : jointAngles) {
56        s = s + Math.round(Math.toDegrees(angle) / roundFactor);
57      }
58      return s;
59    }
60
61    @Override
62    public boolean equals(Object o) {
63      // equals method is found when collision are found when searching the
    HashMap Q in Agent-class
64      State s = (State) o;
65
66      //Checks degree compared to world
67      if (Math.round(Math.toDegrees(this.worldAngle) / roundFactor) != Math.
    round(Math.toDegrees(s.worldAngle) / roundFactor)) {
68        return false; // return false if angles do not match
69      }
70
71      // Checks degrees of each joints
72      for (int i = 0; i < this.jointAngles.size(); i++) {
73        if (Math.round(Math.toDegrees(s.jointAngles.get(i) / roundFactor)) !=
    Math.round(Math.toDegrees(this.jointAngles.get(i) / roundFactor))) {
74          return false;// return false if angles do not match
75        }
76      }
77      return true;
78    }
79
80    public JointAction getRandomAction() {
81      //Returns a random action
82      return Simulation.walker.getState().actions.get((int) (Math.random() *
    actions.size()));
83    }
84
85    public ArrayList<JointAction> getActions() {
86      return actions;
87    }
88
89    public static long getMaxNumberOfStates() {
90      // The numbers for these intervals are found by looking at the set upper
    - and lower limit in each joint
91      int hipInterval = Math.round(55) / roundFactor;
92      int kneeInterval = Math.round(150) / roundFactor;
93      int ankleInterval = Math.round(30) / roundFactor;
```

```
 94        int relativeAngle = Math.round(360) / roundFactor;
 95
 96        // These numbers are multiplied and the product is returned
 97        return (hipInterval*hipInterval*kneeInterval*kneeInterval*ankleInterval*
    ankleInterval*relativeAngle);
 98     }
 99  }
100
```

```java
package QLearning;

import Rendering_dyn4j.Simulation;
import Rendering_dyn4j.ThreadSync;
import org.dyn4j.dynamics.joint.RevoluteJoint;

/*
This class represents an action. It has three different constructors for different types of actions
and method for executing this action.
 */

public class JointAction {
    RevoluteJoint joint; // Joint used in action
    boolean motorOn; // is motor on or is joint relaxed in this action
    int a; // int indicating negative (-1), locked (0) or positive motor input (1)
    boolean noOp; // Boolean indicating a none-action

    public JointAction(RevoluteJoint joint) {
        // Constructor for a relaxed joint action
        this.joint = joint;
        this.motorOn = false;
    }

    public JointAction(RevoluteJoint joint, int i) {
        // Constructor for an action
        this.joint = joint;
        this.a = i; // Integer indicating a negative, zero or positive motor speed
        this.motorOn = true;
    }

    public JointAction() {
        // Third constructor for none operation (noOp) used in terminal states
        this.noOp = true;
    }

    public void doAction() {
        // this method is called in order to execute an action
        synchronized (ThreadSync.lock) {
            if (noOp) {;}
            if (this.motorOn) {
                Simulation.walker.setJoint(this.joint, this.a);
            } else {
                Simulation.walker.relaxJoint(this.joint);
            }
        }
    }

    @Override
    public String toString() {
        // Overridden tostring
        if (noOp) {
            return "noOp";
        } else {
            return joint.getUserData() + " Action: " + a + " Motor on: " +
```

```
54 motorOn;
55       }
56    }
57
58    public boolean isNoOp() {
59       // isNoOp indicates if this is a none operation
60       return noOp;
61    }
62 }
63
```

```java
package Rendering_dyn4j;

import QLearning.State;
import org.dyn4j.collision.CategoryFilter;
import org.dyn4j.dynamics.BodyFixture;
import org.dyn4j.dynamics.World;
import org.dyn4j.dynamics.joint.RevoluteJoint;
import org.dyn4j.geometry.*;
import sample.Main;

import java.util.ArrayList;

/*
This class contains the limbs and joints of the biped body. It also contains method for getting a state from the current posture,
returning the current reward, resetting the walker to its intial position and moving joints.
 */


public class BipedBody {

    World world; // The world object, which the body is added to

    // Limbs
    GameObject torso;
    GameObject upperLeg1;
    GameObject upperLeg2;
    GameObject lowerLeg1;
    GameObject lowerLeg2;
    GameObject foot1;
    GameObject foot2;
    ArrayList<GameObject> limbs = new ArrayList<>();

    // Joints
    public static ArrayList<RevoluteJoint> joints = new ArrayList<>();
    RevoluteJoint hip1;
    RevoluteJoint hip2;
    RevoluteJoint knee1;
    RevoluteJoint knee2;
    RevoluteJoint ankle1;
    RevoluteJoint ankle2;

    // Torque and jointspeed for movement of joints
    Double maxHipTorque = 250.0;
    Double maxKneeTorque = 250.0;
    Double maxAnkleTorque = 150.0;
    Double jointSpeed = 60.0;

    // Categories (for allowing overlap of leg 1 and leg 2)
    CategoryFilter f1 = new CategoryFilter(1, 1);
    CategoryFilter f2 = new CategoryFilter(2, 2);

    // Constructor
    public BipedBody() {
```

```
54        this.world = Simulation.world;
55
56        // Torso
57        torso = new GameObject();
58        torso.setUserData("torso"); // User data set to allow for more readable
      toString() in JointAction
59        {
60           Convex c = Geometry.createRectangle(0.6, 1.0);
61           BodyFixture bf = new BodyFixture(c);
62           torso.addFixture(bf);
63           torso.translate(0, 0);
64           torso.setMass(Mass.Type.NORMAL);
65        }
66        world.addBody(torso);
67
68        // Leg 1
69        // Upper leg
70        upperLeg1 = new GameObject();
71        upperLeg1.setUserData("upper leg 1");
72        {// Fixture4
73           Convex c = Geometry.createRectangle(0.4, 1.05);
74           BodyFixture bf = new BodyFixture(c);
75           bf.setFilter(f1);
76           upperLeg1.addFixture(bf);
77           upperLeg1.translate(0, -1.0);
78           upperLeg1.setMass(Mass.Type.NORMAL);
79
80        }
81        world.addBody(upperLeg1);
82
83        // Lower leg
84        lowerLeg1 = new GameObject();
85        lowerLeg1.setUserData("lower leg 1");
86        {
87           Convex c = Geometry.createRectangle(0.32, 1.05);
88           BodyFixture bf = new BodyFixture(c);
89           bf.setFilter(f1);
90           lowerLeg1.addFixture(bf);
91           lowerLeg1.translate(0, -1.8);
92           lowerLeg1.setMass(Mass.Type.NORMAL);
93        }
94        world.addBody(lowerLeg1);
95
96        // Foot
97        foot1 = new GameObject();
98        foot1.setUserData("foot 1");
99        {// Fixture4
100          Convex c = Geometry.createRectangle(0.6, 0.25);
101          BodyFixture bf = new BodyFixture(c);
102          bf.setFilter(f1);
103          foot1.addFixture(bf);
104          foot1.translate(0.15, -2.3);
105          foot1.setMass(Mass.Type.NORMAL);
106
107       }
```

```
108         world.addBody(foot1);
109 
110         // Joints
111         // Hip
112         hip1 = new RevoluteJoint(torso, upperLeg1, new Vector2(0.0, -.6));
113         hip1.setLimitEnabled(true);
114         hip1.setLimits(Math.toRadians(-50.0), Math.toRadians(5.0));
115         hip1.setReferenceAngle(Math.toRadians(0.0));
116         hip1.setMotorEnabled(true);
117         hip1.setMaximumMotorTorque(maxHipTorque);
118         hip1.setCollisionAllowed(false);
119         hip1.setUserData("hip1");
120         world.addJoint(hip1);
121 
122         // Knee
123         knee1 = new RevoluteJoint(upperLeg1, lowerLeg1, new Vector2(0.0, -1.
    4));
124         knee1.setLimitEnabled(true);
125         knee1.setLimits(Math.toRadians(0.0), Math.toRadians(150.0));
126         knee1.setReferenceAngle(Math.toRadians(0.0));
127         knee1.setMotorEnabled(true);
128         knee1.setMaximumMotorTorque(maxKneeTorque);
129         knee1.setCollisionAllowed(false);
130         knee1.setUserData("knee1");
131         world.addJoint(knee1);
132 
133         // Ankle
134         ankle1 = new RevoluteJoint(lowerLeg1, foot1, new Vector2(0, -2.3));
135         ankle1.setLimitEnabled(true);
136         ankle1.setLimits(Math.toRadians(-15.0), Math.toRadians(15.0));
137         ankle1.setReferenceAngle(Math.toRadians(0.0));
138         ankle1.setMotorEnabled(true);
139         ankle1.setMaximumMotorTorque(maxAnkleTorque);
140         ankle1.setCollisionAllowed(false);
141         ankle1.setUserData("ankle1");
142         world.addJoint(ankle1);
143 
144         // Leg 2
145         // Upper leg
146         upperLeg2 = new GameObject();
147         {
148            Convex c = Geometry.createRectangle(0.4, 1.05);
149            BodyFixture bf = new BodyFixture(c);
150            bf.setFilter(f2);
151            upperLeg2.addFixture(bf);
152            upperLeg2.translate(0, -1.0);
153            upperLeg2.setMass(Mass.Type.NORMAL);
154            upperLeg2.setUserData("upper leg 2");
155 
156         }
157         world.addBody(upperLeg2);
158 
159         // Lower leg
160         lowerLeg2 = new GameObject();
161         {
```

```
162         Convex c = Geometry.createRectangle(0.32, 1.05);
163         BodyFixture bf = new BodyFixture(c);
164         bf.setFilter(f2);
165         lowerLeg2.addFixture(bf);
166         lowerLeg2.translate(0, -1.8);
167         lowerLeg2.setMass(Mass.Type.NORMAL);
168         lowerLeg2.setUserData("lower leg 2");
169       }
170       world.addBody(lowerLeg2);
171
172       // Foot
173       foot2 = new GameObject();
174       {
175         Convex c = Geometry.createRectangle(0.6, 0.25);
176         BodyFixture bf = new BodyFixture(c);
177         bf.setFilter(f2);
178         foot2.addFixture(bf);
179         foot2.translate(0.15, -2.3);
180         foot2.setMass(Mass.Type.NORMAL);
181         foot2.setUserData("foot2");
182
183       }
184       world.addBody(foot2);
185
186       // Joints
187       // Hip
188       hip2 = new RevoluteJoint(torso, upperLeg2, new Vector2(0.0, -.6));
189       hip2.setLimitEnabled(true);
190       hip2.setLimits(Math.toRadians(-50.0), Math.toRadians(5.0));
191       hip2.setReferenceAngle(Math.toRadians(0.0));
192       hip2.setMotorEnabled(true);
193       hip2.setMotorSpeed(Math.toRadians(0.0));
194       hip2.setMaximumMotorTorque(maxHipTorque);
195       hip2.setCollisionAllowed(false);
196       hip2.setUserData("hip2");
197       world.addJoint(hip2);
198
199       // Knee
200       knee2 = new RevoluteJoint(upperLeg2, lowerLeg2, new Vector2(0.0, -1.
      4));
201       knee2.setLimitEnabled(true);
202       knee2.setLimits(Math.toRadians(0.0), Math.toRadians(150.0));
203       knee2.setReferenceAngle(Math.toRadians(0.0));
204       knee2.setMotorEnabled(true);
205       knee2.setMotorSpeed(Math.toRadians(0.0));
206       knee2.setMaximumMotorTorque(maxKneeTorque);
207       knee2.setCollisionAllowed(false);
208       knee2.setUserData("knee2");
209       world.addJoint(knee2);
210
211       // Ankle
212       ankle2 = new RevoluteJoint(lowerLeg2, foot2, new Vector2(0, -2.3));
213       ankle2.setLimitEnabled(true);
214       ankle2.setLimits(Math.toRadians(-15.0), Math.toRadians(15.0));
215       ankle2.setReferenceAngle(Math.toRadians(0.0));
```

```
216         ankle2.setMotorEnabled(true);
217         ankle2.setMotorSpeed(Math.toRadians(0.0));
218         ankle2.setMaximumMotorTorque(maxAnkleTorque);
219         ankle2.setCollisionAllowed(false);
220         ankle2.setUserData("ankle2");
221         world.addJoint(ankle2);
222
223         // Add joints to list
224         joints.add(hip1);
225         joints.add(hip2);
226         joints.add(knee1);
227         joints.add(knee2);
228         joints.add(ankle1);
229         joints.add(ankle2);
230
231         // Add limbs to arraylist
232         limbs.add(torso);
233         limbs.add(upperLeg1);
234         limbs.add(upperLeg2);
235         limbs.add(lowerLeg1);
236         limbs.add(lowerLeg2);
237         limbs.add(foot1);
238         limbs.add(foot2);
239
240     }
241
242     public void setJoint(RevoluteJoint joint, int x) {
243         // setMotorSpeed for joint depending on x
244         if (!joint.isMotorEnabled()) {
245             joint.setMotorEnabled(true);
246         }
247         switch (x) {
248             case -1:
249                 joint.setMotorSpeed(Math.toRadians(jointSpeed));
250                 break;
251             case 0:
252                 joint.setMotorSpeed(0); // joint is "locked"
253                 break;
254             case 1:
255                 joint.setMotorSpeed(Math.toRadians(-jointSpeed));
256                 break;
257         }
258     }
259
260     public void relaxJoint(RevoluteJoint joint) {
261         // Motor is turned of on joint
262         if (joint.isMotorEnabled()) {
263             joint.setMotorEnabled(false);
264         }
265     }
266
267     public State getState() {
268         // Creates a new State object from posture and returns this state
269         return new State(this);
270     }
```

```
271
272    public boolean hasFallen() {
273        // Returns true if torso, upperleg1 or upperleg2 collides with Simulation
       .floor
274        return (CollisionDetector.cl.collision(Simulation.walker.torso, Simulation
       .floor)) ||
275                (CollisionDetector.cl.collision(Simulation.walker.upperLeg1,
       Simulation.floor)) ||
276                (CollisionDetector.cl.collision(Simulation.walker.upperLeg2,
       Simulation.floor));
277    }
278
279    private boolean feetOnTheGround() {
280        return (CollisionDetector.cl.collision(Simulation.walker.foot1, Simulation
       .floor) || (CollisionDetector.cl.collision(Simulation.walker.foot2, Simulation.
       floor)));
281    }
282
283    public double reward() {
284        // This checks values for limbs and returns an appropriate.
285        // The reward is different depending on the reward mode set at runtime
286        // The idea behind the different values returned is described in the Q-
       Learning chapter
287        double reward = 0;
288        switch (Main.mode) {
289            case 0:
290                if ((Simulation.walker.foot2.getChangeInPosition().x + Simulation.
       walker.foot1.getChangeInPosition().x) > 0) {
291                    reward = ((Simulation.walker.foot2.getChangeInPosition().x +
       Simulation.walker.foot1.getChangeInPosition().x) * 5000);
292                }
293                if ((Simulation.walker.foot2.getChangeInPosition().x + Simulation.
       walker.foot1.getChangeInPosition().x) < 0) {
294                    reward = ((Simulation.walker.foot2.getChangeInPosition().x +
       Simulation.walker.foot1.getChangeInPosition().x) * -1000);
295                }
296                if (Simulation.walker.hasFallen()) {reward = -1000;}
297
298                if((Simulation.walker.foot2.getChangeInPosition().x + Simulation.
       walker.foot1.getChangeInPosition().x) == 0){reward = - 1000;}
299                break;
300            case 1:
301                reward = 1500 + ((Simulation.walker.foot2.getWorldCenter().y +
       Simulation.walker.foot1.getWorldCenter().y) * 1000);
302                if (!feetOnTheGround()) {
303                    reward += 1000;
304                }
305                break;
306            case 2:
307                reward = Math.toDegrees(Simulation.walker.knee2.getJointAngle());
308                break;
309        }
310
311        Main.accumulatedReward += reward; // Main.accumulatedReward
       incremented for use in the GUI.
```

```
312
313         return reward; // Reward returned
314     }
315
316
317     public double getRelativeAngle() {
318         // This method returns the angle the body and a straight horizontal
    vector
319         return (new Vector2(hip2.getAnchor1(), torso.getWorldCenter()).
    getAngleBetween(new Vector2(1, 0)));
320     }
321
322     public void resetPosition() {// Method for resetting position of all limbs
323
324         // Rotates all limbs back to initial position
325         torso.setTransform(Transform.IDENTITY);
326         upperLeg1.setTransform(Transform.IDENTITY);
327         upperLeg2.setTransform(Transform.IDENTITY);
328         lowerLeg1.setTransform(Transform.IDENTITY);
329         lowerLeg2.setTransform(Transform.IDENTITY);
330         foot1.setTransform(Transform.IDENTITY);
331         foot2.setTransform(Transform.IDENTITY);
332
333         // Set bodyparts to initial positions
334         torso.translate(0, 0);
335         upperLeg1.translate(0, -1.0);
336         upperLeg2.translate(0, -1.0);
337         lowerLeg1.translate(0, -1.8);
338         lowerLeg2.translate(0, -1.8);
339         foot1.translate(0.15, -2.3);
340         foot2.translate(0.15, -2.3);
341
342         // Clears force - this makes sure that no force is transferred when reset
     is performed
343         for (GameObject limb : limbs) {
344             limb.clearForce();
345             limb.clearTorque();
346             limb.clearAccumulatedTorque();
347             limb.clearAccumulatedForce();
348             limb.setAngularVelocity(0.0);
349             limb.setLinearVelocity(0.0, 0.0);
350         }
351     }
352
353     public boolean isInSight() {
354         // Checks if walkers torso is roughly inside the rendering frame
355         return (Simulation.walker.torso.getWorldCenter().x > -5.5 &&
    Simulation.walker.torso.getWorldCenter().x < 5.5);
356     }
357 }
358
```

```java
package Rendering_dyn4j;

import java.awt.Color;
import java.awt.geom.AffineTransform;
import org.dyn4j.dynamics.Body;
import org.dyn4j.dynamics.BodyFixture;
import org.dyn4j.geometry.Convex;

public class GameObject extends Body {
	/**
	 * The color of the object
	 */
	protected Color color;

	/**
	 * Default constructor.
	 */
	public GameObject() {
	  // randomly generate the color
	  this.color = new Color(
			(float) Math.random() * 0.5f + 0.5f,
			(float) Math.random() * 0.5f + 0.5f,
			(float) Math.random() * 0.5f + 0.5f);
	}

	/**
	 * Draws the body.
	 * <p>
	 * Only coded for polygons and circles.
	 *
	 * @param g the graphics object to render to
	 */
	public void render(java.awt.Graphics2D g) {
	  // save the original transform
	  AffineTransform ot = g.getTransform();

	  // transform the coordinate system from world coordinates to local coordinates
	  AffineTransform lt = new AffineTransform();
	  lt.translate(this.transform.getTranslationX() * Simulation.SCALE, this.transform.getTranslationY() * Simulation.SCALE);
	  lt.rotate(this.transform.getRotation());

	  // apply the transform
	  g.transform(lt);

	  // loop over all the body fixtures for this body
	  for (BodyFixture fixture : this.fixtures) {
	    // get the shape on the fixture
	    Convex convex = fixture.getShape();
	    Graphics2DRenderer.render(g, convex, Simulation.SCALE, color);
	  }

	  // set the original transform
	  g.setTransform(ot);
```

```
54        }
55    }
```

```
1  /*
2   * Copyright (c) 2010-2014 William Bittle  http://www.dyn4j.org/
3   * All rights reserved.
4   *
5   * Redistribution and use in source and binary forms, with or without
    modification, are permitted
6   * provided that the following conditions are met:
7   *
8   *   * Redistributions of source code must retain the above copyright notice,
    this list of conditions
9   *     and the following disclaimer.
10  *   * Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions
11  *     and the following disclaimer in the documentation and/or other materials
     provided with the
12  *     distribution.
13  *   * Neither the name of dyn4j nor the names of its contributors may be used
    to endorse or
14  *     promote products derived from this software without specific prior
    written permission.
15  *
16  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
    CONTRIBUTORS "AS IS" AND ANY EXPRESS OR
17  * IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
    WARRANTIES OF MERCHANTABILITY AND
18  * FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL
    THE COPYRIGHT OWNER OR
19  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
    SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
20  * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
    SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
21  * DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
    ON ANY THEORY OF LIABILITY, WHETHER
22  * IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
    OTHERWISE) ARISING IN ANY WAY OUT
23  * OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
    SUCH DAMAGE.
24  */
25
26  // TODO kommenter gameLoop og initializeWorld
27
28  package Rendering_dyn4j;
29
30  import java.awt.Canvas;
31  import java.awt.Color;
32  import java.awt.Dimension;
33  import java.awt.Toolkit;
34  import java.awt.event.*;
35  import java.awt.geom.AffineTransform;
36  import java.awt.image.BufferStrategy;
37
38  import javax.swing.JFrame;
39
40  import org.dyn4j.dynamics.BodyFixture;
41  import org.dyn4j.dynamics.World;
```

```
42 import org.dyn4j.geometry.Geometry;
43 import org.dyn4j.geometry.Mass;
44 import org.dyn4j.geometry.Rectangle;
45 import sample.*;
46
47 /**
48  * Class used to show a simple example of using the dyn4j project using
49  * Java2D for rendering.
50  * <p>
51  * This class can be used as a starting point for projects.
52  *
53  * @author William Bittle
54  * @version 3.1.5
55  * @since 3.0.0
56  */
57 public class Simulation extends JFrame {
58
59     public static BipedBody walker;
60     public static GameObject floor;
61
62     /**
63      * The serial version id
64      */
65     private static final long serialVersionUID = 56637602931448826 35L;
66
67     /**
68      * The scale 65 pixels per meter
69      */
70     public static final double SCALE = 65.0;
71
72     /**
73      * The conversion factor from nano to base
74      */
75     public static final double NANO_TO_BASE = 1.0e9;
76
77     /**i
78      * Custom Body class to add drawing functionality.
79      *
80      * @author William Bittle
81      * @version 3.0.2
82      * @since 3.0.0
83      */
84
85     /**
86      * The canvas to draw to
87      */
88     protected Canvas canvas;
89
90     /**
91      * The dynamics engine
92      */
93     public static World world;
94
95     /**
96      * Wether the example is stopped or not
```

```
 97      */
 98     protected boolean stopped;
 99
100     /**
101      * The time stamp for the last iteration
102      */
103     protected long last;
104
105     public int getSimulationSpeed() {
106        return simulationSpeed;
107     }
108
109     public void setSimulationSpeed(int simulationSpeed) {
110        this.simulationSpeed = simulationSpeed;
111     }
112
113     private int simulationSpeed = 1;
114
115     /**
116      * Default constructor for the window
117      */
118     public Simulation() {
119        // setup the JFrame
120        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
121
122        // add a window listener
123        this.addWindowListener(new WindowAdapter() {
124           /* (non-Javadoc)
125            * @see java.awt.event.WindowAdapter#windowClosing(java.awt.event.
    WindowEvent)
126            */
127           @Override
128           public void windowClosing(WindowEvent e) {
129              // before we stop the JVM stop the example
130              stop();
131              super.windowClosing(e);
132           }
133        });
134
135        // create the size of the window
136        Dimension size = new Dimension(800, 600);
137
138        // create a canvas to paint to
139        this.canvas = new Canvas();
140        this.canvas.setPreferredSize(size);
141        this.canvas.setMinimumSize(size);
142        this.canvas.setMaximumSize(size);
143
144        // add the canvas to the JFrame
145        this.add(this.canvas);
146
147        // make the JFrame not resizable
148        // (this way I dont have to worry about resize events)
149        this.setResizable(false);
150
```

```
151         // size everything
152         this.pack();
153 
154         // make sure we are not stopped
155         this.stopped = false;
156 
157         // setup the world
158         this.initializeWorld();
159 
160         this.setTitle("Machine Learning");
161 
162         // show it
163         this.setVisible(true);
164 
165         // start it
166         this.start();
167     }
168 
169     /**
170      * Creates game objects and adds them to the world.
171      * <p>
172      * Basically the same shapes from the Shapes test in
173      * the TestBed.
174      */
175     private void initializeWorld() {
176         // create the world
177         this.world = new World();
178 
179         // create the floor
180         floor = new GameObject();
181         BodyFixture floorFixture = new BodyFixture(Geometry.createRectangle(
    15.0, 1.0));
182         floorFixture.setFriction(1000.0);
183         floor.addFixture(floorFixture);
184         floor.setMass(Mass.Type.INFINITE);
185 
186         // move the floor down a bit
187         floor.translate(0.0, -2.95);
188         this.world.addBody(floor);
189 
190         if (Main.mode == 1 || Main.mode == 2) {
191             // Add walls, if needed by mode
192             Rectangle wall1Rect = new Rectangle(1.0, 15.0);
193             GameObject wall1 = new GameObject();
194             BodyFixture wall1Fixture = new BodyFixture(Geometry.
    createRectangle(1.0, 15.0));
195             wall1.addFixture(wall1Fixture);
196             wall1.setMass(Mass.Type.INFINITE);
197             wall1.translate(-6, 0);
198 
199             GameObject wall2 = new GameObject();
200             BodyFixture wall2Fixture = new BodyFixture(Geometry.
    createRectangle(1.0, 15.0));
201             wall2.addFixture(wall2Fixture);
202             wall2.setMass(Mass.Type.INFINITE);
```

```
203         wall2.translate(6, 0);
204
205         world.addBody(wall1);
206         world.addBody(wall2);
207       }
208
209       walker = new BipedBody();
210     }
211
212     /**
213      * Start active rendering the example.
214      * <p>
215      * This should be called after the JFrame has been shown.
216      */
217     public void start() {
218       // initialize the last update time
219       this.last = System.nanoTime();
220       // don't allow AWT to paint the canvas since we are
221       this.canvas.setIgnoreRepaint(true);
222       // enable double buffering (the JFrame has to be
223       // visible before this can be done)
224       this.canvas.createBufferStrategy(2);
225       // run a separate thread to do active rendering
226       // because we don't want to do it on the EDT
227       Thread thread = new Thread() {
228         public void run() {
229           // perform an infinite loop stopped
230           // render as fast as possible
231           while (!isStopped()) {
232             gameLoop();
233             // you could add a Thread.yield(); or
234             // Thread.sleep(long) here to give the
235             // CPU some breathing room
236           }
237         }
238       };
239       // set the game loop thread to a daemon thread so that
240       // it cannot stop the JVM from exiting
241       thread.setDaemon(true);
242       // start the game loop
243       thread.start();
244
245
246     }
247
248
249     /**
250      * The method calling the necessary methods to update
251      * the game, graphics, and poll for input.
252      */
253     protected void gameLoop() {
254       // get the graphics object to render to
255       java.awt.Graphics2D g = (java.awt.Graphics2D) this.canvas.
    getBufferStrategy().getDrawGraphics();
256
```

```
257         // before we render everything im going to flip the y axis and move the
258         // origin to the center (instead of it being in the top left corner)
259         AffineTransform yFlip = AffineTransform.getScaleInstance(1, -1);
260         AffineTransform move = AffineTransform.getTranslateInstance(400, -
        300);
261 
262         g.transform(yFlip);
263         g.transform(move);
264 
265         // now (0, 0) is in the center of the screen with the positive x axis
266         // pointing right and the positive y axis pointing up
267 
268         // render anything about the Example (will render the World objects)
269         this.render(g);
270 
271         // dispose of the graphics object
272         g.dispose();
273 
274         // blit/flip the buffer
275         BufferStrategy strategy = this.canvas.getBufferStrategy();
276         if (!strategy.contentsLost()) {
277            strategy.show();
278         }
279 
280         // Sync the display on some systems.
281         // (on Linux, this fixes event queue problems)
282         Toolkit.getDefaultToolkit().sync();
283 
284         // update the World
285 
286         // get the current time
287         long time = System.nanoTime();
288         // get the elapsed time from the last iteration
289         long diff = time - this.last;
290         // set the last time
291         this.last = time;
292         // convert from nanoseconds to seconds
293         double elapsedTime = ((double) diff / NANO_TO_BASE);
294 
295         // Multiply with simulation speed
296         elapsedTime = elapsedTime * simulationSpeed;
297         // update the world with the elapsed time
298 
299         // Sync to threadsync
300         synchronized (ThreadSync.lock) {
301            this.world.update(elapsedTime, Integer.MAX_VALUE);
302         }
303      }
304 
305      public double getElapsedTime() {
306         return world.getAccumulatedTime() * simulationSpeed;
307      }
308 
309 
310      /**
```

```
311      * Renders the example.
312      *
313      * @param g the graphics object to render to
314      */
315     protected void render(java.awt.Graphics2D g) {
316         // lets draw over everything with a white background
317         g.setColor(Color.WHITE);
318         g.fillRect(-400, -300, 800, 600);
319 
320         // lets move the view up some
321         g.translate(0.0, -1.0 * SCALE);
322 
323         // draw all the objects in the world
324         for (int i = 0; i < this.world.getBodyCount(); i++) {
325             // get the object
326             GameObject go = (GameObject) this.world.getBody(i);
327             // draw the object
328             go.render(g);
329         }
330 
331 
332     }
333 
334     /**
335      * Stops the example.
336      */
337     public synchronized void stop() {
338         this.stopped = true;
339     }
340 
341     /**
342      * Returns true if the example is stopped.
343      *
344      * @return boolean true if stopped
345      */
346     public synchronized boolean isStopped() {
347         return this.stopped;
348     }
349 
350     public void step(int i) {
351         this.world.step(i);
352     }
353 
354     public double getStepFrequency() {
355         return this.world.getSettings().getStepFrequency();
356     }
357 }
358 
```

```
1  package Rendering_dyn4j;
2
3  /*
4  This class is ensuring sync in threads between GUI and dyn4j simulation. This
   was done with help from Willaim at the
5  dyn4j forum.
6   */
7
8  public class ThreadSync {
9     // A new object lock is created making it possible for other methods to
   sync to this
10    public static final Object lock = new Object();
11 }
12
```

```java
package Rendering_dyn4j;

import org.dyn4j.collision.broadphase.DynamicAABBTree;
import org.dyn4j.collision.manifold.Manifold;
import org.dyn4j.collision.narrowphase.Penetration;
import org.dyn4j.dynamics.Body;
import org.dyn4j.dynamics.BodyFixture;
import org.dyn4j.dynamics.CollisionListener;
import org.dyn4j.dynamics.contact.ContactConstraint;

/*
This class contains methods for collision detection. This is done using the CollisionListener in dyn4j.
 */

public class CollisionDetector {
   public static CollisionListener cl = new CollisionListener() {
      DynamicAABBTree dynamicAABBTree = new DynamicAABBTree();
      @Override
      public boolean collision(Body body, Body body1) {
         synchronized (ThreadSync.lock) {
            // Bodies are added to the broadphase detector
            dynamicAABBTree.add(body);
            dynamicAABBTree.add(body1);
            return dynamicAABBTree.detect(body, body1); // Returns boolean for collision detected
         }
      }

      // Below are two overridden but unused methods
      @Override
      public boolean collision(Body body, BodyFixture bodyFixture, Body body1, BodyFixture bodyFixture1, Penetration penetration) {
         return false;
      }

      @Override
      public boolean collision(Body body, BodyFixture bodyFixture, Body body1, BodyFixture bodyFixture1, Manifold manifold) {
         return false;
      }

      @Override
      public boolean collision(ContactConstraint contactConstraint) {
         return false;
      }
   };
}

```

```
1  /*
2   * Copyright (c) 2010-2014 William Bittle  http://www.dyn4j.org/
3   * All rights reserved.
4   *
5   * Redistribution and use in source and binary forms, with or without
    modification, are permitted
6   * provided that the following conditions are met:
7   *
8   *   * Redistributions of source code must retain the above copyright notice,
    this list of conditions
9   *     and the following disclaimer.
10  *   * Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions
11  *     and the following disclaimer in the documentation and/or other materials
     provided with the
12  *     distribution.
13  *   * Neither the name of dyn4j nor the names of its contributors may be used
    to endorse or
14  *     promote products derived from this software without specific prior
    written permission.
15  *
16  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
    CONTRIBUTORS "AS IS" AND ANY EXPRESS OR
17  * IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
    WARRANTIES OF MERCHANTABILITY AND
18  * FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL
    THE COPYRIGHT OWNER OR
19  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
    SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
20  * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
    SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
21  * DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
    ON ANY THEORY OF LIABILITY, WHETHER
22  * IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
    OTHERWISE) ARISING IN ANY WAY OUT
23  * OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
    SUCH DAMAGE.
24  */
25 package Rendering_dyn4j;
26
27 import java.awt.Color;
28 import java.awt.Graphics2D;
29 import java.awt.geom.AffineTransform;
30 import java.awt.geom.Arc2D;
31 import java.awt.geom.Ellipse2D;
32 import java.awt.geom.Line2D;
33 import java.awt.geom.Path2D;
34
35 import org.dyn4j.geometry.Capsule;
36 import org.dyn4j.geometry.Circle;
37 import org.dyn4j.geometry.Ellipse;
38 import org.dyn4j.geometry.HalfEllipse;
39 import org.dyn4j.geometry.Polygon;
40 import org.dyn4j.geometry.Segment;
41 import org.dyn4j.geometry.Shape;
```

```
42 import org.dyn4j.geometry.Slice;
43 import org.dyn4j.geometry.Vector2;
44
45 /**
46  * Graphics2D renderer for dyn4j shape types.
47  * @author William Bittle
48  * @version 3.1.7
49  * @since 3.1.5
50  */
51 public final class Graphics2DRenderer {
52    /**
53     * Renders the given shape to the given graphics context using the given
    scale and color.
54     * @param g the graphics context
55     * @param shape the shape to render
56     * @param scale the scale to render the shape (pixels per dyn4j unit (
    typically meter))
57     * @param color the color
58     */
59    public static final void render(Graphics2D g, Shape shape, double scale,
    Color color) {
60       // no-op
61       if (shape == null) return;
62
63       // just default the color
64       if (color == null) color = Color.ORANGE;
65
66       if (shape instanceof Circle) {
67          Graphics2DRenderer.render(g, (Circle)shape, scale, color);
68       } else if (shape instanceof Polygon) {
69          Graphics2DRenderer.render(g, (Polygon)shape, scale, color);
70       } else if (shape instanceof Segment) {
71          Graphics2DRenderer.render(g, (Segment)shape, scale, color);
72       } else if (shape instanceof Capsule) {
73          Graphics2DRenderer.render(g, (Capsule)shape, scale, color);
74       } else if (shape instanceof Ellipse) {
75          Graphics2DRenderer.render(g, (Ellipse)shape, scale, color);
76       } else if (shape instanceof Slice) {
77          Graphics2DRenderer.render(g, (Slice)shape, scale, color);
78       } else if (shape instanceof HalfEllipse) {
79          Graphics2DRenderer.render(g, (HalfEllipse)shape, scale, color);
80       } else {
81          // unknown shape
82       }
83    }
84
85    /**
86     * Renders the given {@link Circle} to the given graphics context using the
    given scale and color.
87     * @param g the graphics context
88     * @param circle the circle to render
89     * @param scale the scale to render the shape (pixels per dyn4j unit (
    typically meter))
90     * @param color the color
91     */
```

```
 92    public static final void render(Graphics2D g, Circle circle, double scale,
       Color color) {
 93        double radius = circle.getRadius();
 94        Vector2 center = circle.getCenter();
 95
 96        double radius2 = 2.0 * radius;
 97        Ellipse2D.Double c = new Ellipse2D.Double(
 98          (center.x - radius) * scale,
 99          (center.y - radius) * scale,
100          radius2 * scale,
101          radius2 * scale);
102
103        // fill the shape
104        g.setColor(color);
105        g.fill(c);
106        // draw the outline
107        g.setColor(getOutlineColor(color));
108        g.draw(c);
109
110        // draw a line so that rotation is visible
111        Line2D.Double l = new Line2D.Double(
112              center.x * scale,
113              center.y * scale,
114              (center.x + radius) * scale,
115              center.y * scale);
116        g.draw(l);
117    }
118
119    /**
120     * Renders the given {@link Polygon} to the given graphics context using
       the given scale and color.
121     * @param g the graphics context
122     * @param polygon the polygon to render
123     * @param scale the scale to render the shape (pixels per dyn4j unit (
       typically meter))
124     * @param color the color
125     */
126    public static final void render(Graphics2D g, Polygon polygon, double
       scale, Color color) {
127        Vector2[] vertices = polygon.getVertices();
128        int l = vertices.length;
129
130        // create the awt polygon
131        Path2D.Double p = new Path2D.Double();
132        p.moveTo(vertices[0].x * scale, vertices[0].y * scale);
133        for (int i = 1; i < l; i++) {
134          p.lineTo(vertices[i].x * scale, vertices[i].y * scale);
135        }
136        p.closePath();
137
138        // fill the shape
139        g.setColor(color);
140        g.fill(p);
141        // draw the outline
142        g.setColor(getOutlineColor(color));
```

```
143         g.draw(p);
144     }
145 
146     /**
147      * Renders the given {@link Segment} to the given graphics context using the given scale and color.
148      * @param g the graphics context
149      * @param segment the segment to render
150      * @param scale the scale to render the shape (pixels per dyn4j unit (typically meter))
151      * @param color the color
152      */
153     public static final void render(Graphics2D g, Segment segment, double scale, Color color) {
154         Vector2[] vertices = segment.getVertices();
155 
156         Line2D.Double l = new Line2D.Double(
157             vertices[0].x * scale,
158             vertices[0].y * scale,
159             vertices[1].x * scale,
160             vertices[1].y * scale);
161 
162         // draw the outline
163         g.setColor(getOutlineColor(color));
164         g.draw(l);
165     }
166 
167     /**
168      * Renders the given {@link Capsule} to the given graphics context using the given scale and color.
169      * @param g the graphics context
170      * @param capsule the capsule to render
171      * @param scale the scale to render the shape (pixels per dyn4j unit (typically meter))
172      * @param color the color
173      */
174     public static final void render(Graphics2D g, Capsule capsule, double scale, Color color) {
175         // get the local rotation and translation
176         double rotation = capsule.getRotation();
177         Vector2 center = capsule.getCenter();
178 
179         // save the old transform
180         AffineTransform oTransform = g.getTransform();
181         // translate and rotate
182         g.translate(center.x * scale, center.y * scale);
183         g.rotate(rotation);
184 
185         double width = capsule.getLength();
186         double radius = capsule.getCapRadius();
187         double radius2 = radius * 2.0;
188 
189         Arc2D.Double arcL = new Arc2D.Double(
190                 -(width * 0.5) * scale,
191                 -radius * scale,
```

```
192             radius2 * scale,
193             radius2 * scale,
194             90.0,
195             180.0,
196             Arc2D.OPEN);
197         Arc2D.Double arcR = new Arc2D.Double(
198             (width * 0.5 - radius2) * scale,
199             -radius * scale,
200             radius2 * scale,
201             radius2 * scale,
202             -90.0,
203             180.0,
204             Arc2D.OPEN);
205 
206         // connect the shapes
207         Path2D.Double path = new Path2D.Double();
208         path.append(arcL, true);
209         path.append(new Line2D.Double(arcL.getEndPoint(), arcR.getStartPoint()
    ), true);
210         path.append(arcR, true);
211         path.append(new Line2D.Double(arcR.getEndPoint(), arcL.getStartPoint()
    ), true);
212 
213         // set the color
214         g.setColor(color);
215         // fill the shape
216         g.fill(path);
217         // set the color
218         g.setColor(getOutlineColor(color));
219         // draw the shape
220         g.draw(path);
221 
222         // re-instate the old transform
223         g.setTransform(oTransform);
224     }
225 
226     /**
227      * Renders the given {@link Ellipse} to the given graphics context using
    the given scale and color.
228      * @param g the graphics context
229      * @param ellipse the ellipse to render
230      * @param scale the scale to render the shape (pixels per dyn4j unit (
    typically meter))
231      * @param color the color
232      */
233     public static final void render(Graphics2D g, Ellipse ellipse, double scale,
    Color color) {
234         // get the local rotation and translation
235         double rotation = ellipse.getRotation();
236         Vector2 center = ellipse.getCenter();
237 
238         // save the old transform
239         AffineTransform oTransform = g.getTransform();
240         g.translate(center.x * scale, center.y * scale);
241         g.rotate(rotation);
```

```
242
243         double width = ellipse.getWidth();
244         double height = ellipse.getHeight();
245         Ellipse2D.Double c = new Ellipse2D.Double(
246             (-width * 0.5) * scale,
247             (-height * 0.5) * scale,
248             width * scale,
249             height * scale);
250
251         // fill the shape
252         g.setColor(color);
253         g.fill(c);
254         // draw the outline
255         g.setColor(getOutlineColor(color));
256         g.draw(c);
257
258         // re-instate the old transform
259         g.setTransform(oTransform);
260     }
261
262     /**
263      * Renders the given {@link Slice} to the given graphics context using the
    given scale and color.
264      * @param g the graphics context
265      * @param slice the slice to render
266      * @param scale the scale to render the shape (pixels per dyn4j unit (
    typically meter))
267      * @param color the color
268      */
269     public static final void render(Graphics2D g, Slice slice, double scale,
    Color color) {
270         double radius = slice.getSliceRadius();
271         double theta2 = slice.getTheta() * 0.5;
272
273         // get the local rotation and translation
274         double rotation = slice.getRotation();
275         Vector2 circleCenter = slice.getCircleCenter();
276
277         // save the old transform
278         AffineTransform oTransform = g.getTransform();
279         // translate and rotate
280         g.translate(circleCenter.x * scale, circleCenter.y * scale);
281         g.rotate(rotation);
282
283         // to draw the arc, java2d wants the top left x,y
284         // as if you were drawing a circle
285         Arc2D a = new Arc2D.Double(-radius * scale,
286                 -radius * scale,
287                 2.0 * radius * scale,
288                 2.0 * radius * scale,
289                 -Math.toDegrees(theta2),
290                 Math.toDegrees(2.0 * theta2),
291                 Arc2D.PIE);
292
293         // fill the shape
```

```
294         g.setColor(color);
295         g.fill(a);
296         // draw the outline
297         g.setColor(getOutlineColor(color));
298         g.draw(a);
299 
300         // re-instate the old transform
301         g.setTransform(oTransform);
302     }
303 
304     /**
305      * Renders the given {@link HalfEllipse} to the given graphics context using the given scale and color.
306      * @param g the graphics context
307      * @param halfEllipse the halfEllipse to render
308      * @param scale the scale to render the shape (pixels per dyn4j unit (typically meter))
309      * @param color the color
310      */
311     public static final void render(Graphics2D g, HalfEllipse halfEllipse, double scale, Color color) {
312         double width = halfEllipse.getWidth();
313         double height = halfEllipse.getHeight();
314 
315         // get the local rotation and translation
316         double rotation = halfEllipse.getRotation();
317         Vector2 center = halfEllipse.getEllipseCenter();
318 
319         // save the old transform
320         AffineTransform oTransform = g.getTransform();
321         // translate and rotate
322         g.translate(center.x * scale, center.y * scale);
323         g.rotate(rotation);
324 
325         // to draw the arc, java2d wants the top left x,y
326         // as if you were drawing a circle
327         Arc2D a = new Arc2D.Double(
328                 (-width * 0.5) * scale,
329                 -height * scale,
330                 width * scale,
331                 height * 2.0 * scale,
332                 0,
333                 -180.0,
334                 Arc2D.PIE);
335 
336         // fill the shape
337         g.setColor(color);
338         g.fill(a);
339         // draw the outline
340         g.setColor(getOutlineColor(color));
341         g.draw(a);
342 
343         // re-instate the old transform
344         g.setTransform(oTransform);
345     }
```

```
346
347     /**
348      * Returns the outline color for the given color.
349      * @param color the fill color
350      * @return Color
351      */
352     private static final Color getOutlineColor(Color color) {
353         Color oc = color.darker();
354         return new Color(oc.getRed(), oc.getGreen(), oc.getBlue(), color.
    getAlpha());
355     }
356 }
357
```