Design and Optimization of Control Primitives for Simulated Characters

by

Shuo Shen

Dual B. S. in Computer Science, Zhejiang University and Simon Fraser University, 2011

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

(Computer Science)

The University Of British Columbia

(Vancouver)

December 2013

Abstract

Physics-based character motion has the potential of achieving realistic motions without laborious work from artists and without needing to use motion capture data. It has potential applications in film, games and humanoid robotics. However, designing a controller for physics motions is a difficult task. It requires expertise in software engineering and understanding of control methods. Researchers typically develop their own dedicated software framework and invent their own sets of control rules to control physics-based characters. This creates an impediment to the non-expert who wants to create interesting motions and others who want to share and revise motions. In this thesis, we demonstrate that a set of motion primitives that have been developed in recent years constitute effective building blocks for authoring physics-based character motions. These motion primitives are made accessible using an expressive and flexible motion scripting language. The motion language allows a motion designer to create controllers in a text file that can be loaded at runtime. This is intended to simplify motion design, debugging, understanding and sharing. We use this framework to create several interesting 2D planar motions. An optimization framework is integrated that allows the hand-designed motion controller to be optimized for more interesting behaviors, such as a fast prone-to-standing motion.

We also develop a state-action compatibility model for adaping controllers to new situations. The state-action compatibility model maintains a hypervolume of compatible states ("situations") and actions (controllers). It allows queries for compatible actions given a state.

Preface

The CMA-ES algorithm described in Chapter 5 was originally developed by N. Hansen and A. Ostermeier [19], and further explained in [1, 3]. In Chapter 6, the query algorithm is based on the diversity optimization algorithm developed by S. Agrawal, S. Shen and M. van de Panne [7]. Several figures and part of texts from this thesis are copyright and are reused in this thesis by permission. Figure 1.1 is recreated from [6]. Figure 5.1 is adapted from Wikepedia [1]. Figures with the phrase "used with permission" in the caption are used with permission from the authors of the cited papers. The rest of the work is original and it was developed by the author Shuo Shen who discussed with Dr. Michiel van de Panne.

Table of Contents

List of Tables

List of Figures

Acknowledgments

This thesis could not have been written without my supervisor Michiel van de Panne, who encouraged and guided me through my entire academic program. I would like to express my sincerest gratitude to Michiel, for his patience, inspiration and resourcefulness. Thank you!

I would also like to thank the second reader of my thesis, Ian Mitchell, for providing valuable feedback on the writing of this thesis.

Many thanks to my friends and colleagues, Brian Xu, Chuan Zhu, Matt Dirks, C. Albert Thompson, Sarwar Alam, Lan Wei, Shu Yang, Yufeng Zhu, Lei Xiao and Andrew Smith for helping out with my study and thesis writing. That means a lot to me.

Lastly, a heartfelt thanks goes out to my parents and my girlfriend Lina Miao, for all their love and support.

Chapter 1

Introduction

Traditional computer animation relies on keyframing. In this technique, animators first draw characters in a number of key frames at relatively large time intervals and fill in the rest of frames using "in-between" frames to get a full animation. The work of finding appropriate key frames and filling in the rest of the frames is artistic and laborious. The keyframing technique is extended for use in kinematics-based character animation techniques by using splines to define the values of the degrees of freedom as a function of time. These interpolate the keyframes and thus fully specify the in-between frames. This process automates some of the artists' work but it is difficult for this technique to maintain plausible physics, such as motion due to gravity and contact forces.

Physics-based character animation, on the other hand, models character motion from first principles by using internal forces and physics-based simulation. Consequently the physical plausibility of the motions comes as a byproduct of the simulation. With simulation, less input is potentially required from human experts to author a physically-plausible character motion. Compared to kinematics-based animation, physics-based character animation can be extended more easily by changing a character's dimensions and weights or adjusting its environment. It has seen use in film and video-game industry, although to date this has most often been used for "passive ragdoll" simulations. It remains challenging to design the control solutions that are needed to emulate the muscle-based motor skills that drive human and animal motions in reality.

Figure 1.1: Big Dog robot developed by Boston Dynamics

Robotics research has also seen considerable ongoing efforts towards modeling motor skills for human and animal-like movements that can be instantiated on robots. Locomotion skills have been developed for humanoid robots, and quadrupeds such as the Big Dog robot [37] (Figure 1.1) and the Wildcat robot [9].

One general approach to developing motor skills is to develop fundamental building blocks of motions, or *motion primitives*. Motion primitives are rules that compute torques and can be composed in parallel or sequentially to build up control strategies for interesting robotic movements. However, there exist a multitude of challenges in developing motion primitives: what is a proper set of motion primitives, and when should they be invoked? The problem space consists of high dimensional states and actions and can contain discontinuities caused by contacts. Also, human movement is usually *underactuated*. Due to these reasons, seemingly simple problems are very difficult to solve and still remain open today. For example, it is still unclear how to a develop a robust rising controller, i.e., a controller that enables a humanoid character to stand up from various situations.

There have been a number of demonstrated successes in learning motor controls in physics-based character animation. Muscle models have been developed to simulate joint behaviors that have produced natural locomotions. These models range from simple proportional-derivative (PD) control to detailed simulations with biological actuators. Motor learning helps robots and physics-based characters improve on movements over time by trial-and-errors on the gathered feedbacks. Locomotion controllers have been developed successfully for humanoid and

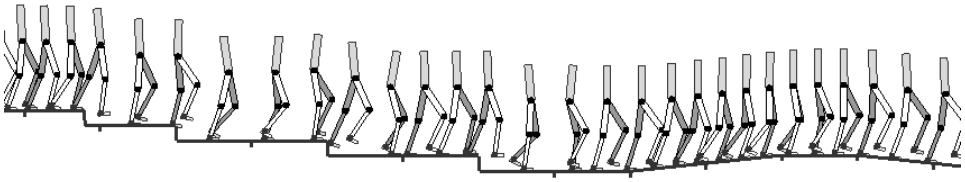


Figure 1.2: Walking gait from SIMBICON [41]. Used with permission from van de Panne.

animal-like characters, e.g., walking (Figure 1.2), running, rolling, galloping, etc. Among these, walking is one of the most heavily studied and the most successful skills. Physics-based characters are able to walk on different slopes at various velocities. They are robust to perturbations such as pushes and can adapt to different ground models including sloped terrains and terrains with steps.

1.1 Goals

A primary motivation for our work arises from the current inefficiency of research in controller development for physics-based character animation. Much ongoing effort has been devoted to authoring motions for rigid-body humanoid or animal characters. Researchers have developed a variety of ways of implementing motion controls using conceptually similar control primitives. However, it is difficult for them to share motions with each other. There is an inevitable demand for a unified tool that allows researchers to design motions using a well-known set of motion primitives, and to share and improve on other people's work with a small amount of effort. Well defined languages have significant impacts on other fields that greatly benefit from end-user development and design, such as Renderman [38] for rendering shaders and OpenFab [34] for 3D printing. A principle goal of our work is to develop a motion authoring framework that exposes a good set of primitives using a simple controller language.

Another motivation of the work in this thesis arises from the need to produce diverse and adaptive physics-based motions. Motions that work only in one fixed style or in the same situation, e.g. jump in same height or walking on a fixed-sloped terrain, are not that interesting. The potential of being able to generate a particular motion task in different styles and in different situations is part of the

promises of physics-based motion synthesis techniques. We propose a method for modelling the space of all possible feasible actions for a given situation, using a novel *state-action compatibility* model.

1.2 Contributions

The main contributions of this thesis are as follows:

- We create a flexible, scriptable implementation of the motion primitives that have recently been developed. This potentially allows non-expert users to author a wide range of motions without needing to develop or edit source code.
- We create several motions using a simple motion-scripting language, including kip-up, sit-to-stand, walk and prone-to-stand.
- We develop an approach that allows motion scripts to be parameterized and optimized. This is demonstrated for an optimization of a kip-up and a prone-to-stand motion for agility.
- We develop a new method for modelling the subspace of motions that accomplish a given task, which we refer to as the state-action compatibility model.

1.3 Overview

The remainder of this thesis is organized in the following structure. The components developed from Chapter 3 to Chapter 5 are visualized in Figure 1.3.

- **Chapter 2** reviews the related work in physics-based character animation.
- **Chapter 3** introduces motion primitives and the related control strategies that will be used by the scriptable framework.
- **Chapter 4** describes the scripting language and the development of several motions using this language.

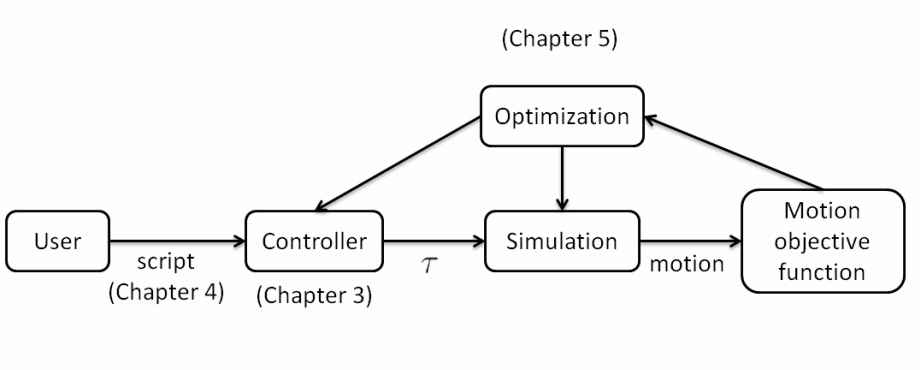


Figure 1.3: The overall architecture of the system (Chapters 3-5)

- **Chapter 5** describes the parameterization and optimization of scriptable motions.
- **Chapter 6** introduces the state-action compatibility model and presents related results.
- **Chapter 7** concludes and discusses directions of future work.

Chapter 2

Related Work

Physics-based character animations can be achieved by using "inverse dynamics" methods or using controllers based on finite state machines along with forward dynamics simulation. Most research on physics-based character animation falls into one of these two categories of methods. In this chapter, we will discuss the work from these two categories that are similar in spirit to ours, with a focus on recently developed methods. A complete survey on physics-based character animation is outside the scope of this thesis. Such a survey can be found in [16].

2.1 Inverse Dynamics Based Methods

Inverse dynamics methods solve the equation of motion (EOM) for torques at every time step in order to achieve desired kinematic properties, normally the desired joint accelerations. This then produces a trajectory that is physically plausible. If desired, the torques can also then be fed into physics simulation in order to regenerate a trajectory. This is fully consistent with frictional effects, collisions, and other limits that are not directly captured by desired accelerations. This method can be viewed as planning the motion one time step at a time. Lee et al. [25] design controllers that regulate desired joint accelerations based on reference motion and balance requirements, and use inverse dynamics to produce motions that conform to captured motion sequences. Macchietto et al. [27] optimizes the desired joint accelerations in order to satisfy several balance feedback laws. They use in-

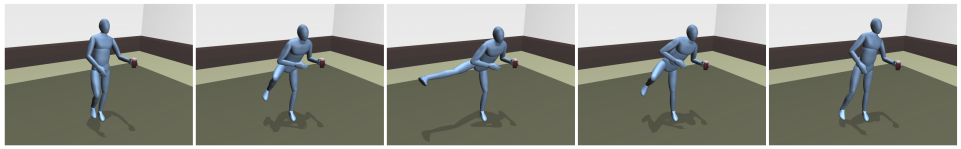


Figure 2.1: A side kick motion produced using inverse dynamics approach [27]. Used with permission from Macchietto.

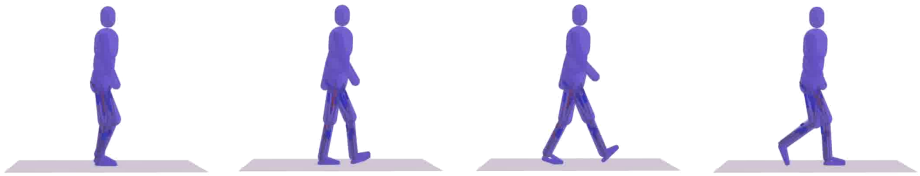


Figure 2.2: A walk motion produced by trajectory optimization [29]. Used with permission from Wang.

verse dynamics to compute torques that help bipedal and single-legged characters maintain balance while doing various in-place movements, such as the kick motion illustrated in Figure 2.1.

Trajectory optimization techniques, also known as space-time constraints [39] are similar to inverse dynamics methods, but they plan an optimal motion over a larger time horizon. This approach finds an entire motion sequence and torque sequence using optimization. The produced trajectory is physics plausible because it is required to satisfy the EOM constraints. The trajectory optimization technique allows a user to specify a set of additional constraints for the motion, e.g. a certain pose must be achieved at a certain time. It further allows the specification of an objective function, e.g. one that rewards the motion for being energy efficient. An optimization then solves for a motion trajectory that satisfies the constraints while minimizing the objective. Mordatch et al. [28, 29] use this method to generate various human motions including walk. The walk motion is shown in Figure 2.2. The trajectory optimization technique can be seen as a variation of the inverse dynamics method, with the difference that it plans once for all the steps in the entire motion. The optimization of an entire motion is typically an offline process.

Online trajectory optimization over a fixed finite horizon, also known as Model

Predictive Control (MPC), works in a similar fashion as trajectory optimization. However, instead of planning the motion once, this method re-plans at each time step, and only over a shorter time window, typically 0.5-1.5 seconds for motions involving human dynamics. This method lies somewhere between inverse dynamics methods and trajectory optimization methods. It is a tradeoff between the required computations and the time span of the motion planning window. It can be used as an online method but can produce motions slower than realtime. This method is used by Tassa et al. [32] to synthesize rise-up motions at a rate 7x slower than realtime. Al Borno et al. [8] use an offline implementation of this method to develop a wider range of human motions with simple task-specific objectives.

2.2 Finite State Machine (FSM) Based Controllers

FSM-based controllers compute torques based on control goals, and feed the torques into a forward dynamics simulator. With the development of forward simulation engines, FSM-based methods do not require the direct knowledge of equations of motion [16] and allow interactive control of the motions [24, 42]. FSM based controllers have been used to produce physics-based character animations over the last two decades [31, 33]. Variations of FSM based controllers are applied in robotics [15, 30].

Several components are used for FSM based controllers, such as PD controls for pose tracking in joint angle space, inverse kinematics for controlling end effector position in Cartesian space, virtual forces applied via a Jacobian transpose and foot placement for balance, etc. Coros et al. [12] use PD controls, inverse kinematics, virtual forces, gravity compensation and inverted pendulum models to create generalized biped walking controllers for a variety of character morphologies in various environments (Figure 2.3). These are the fundamental components of this thesis work.

A considerable amount of work has focused on locomotion skills. Yin et al. [41] develop robust locomotion skills including walking and running for biped characters using an FSM-based controller with simple feedback mechanisms. Kwon et al. [22] use an inverse pendulum model to regulate balance for running steps. Coros et al. [13] create quadrupedal locomotions using a gait control graph and

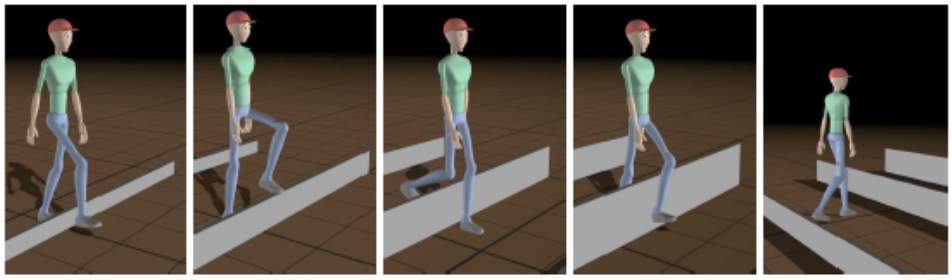


Figure 2.3: Biped walker stepping over a sequence of obstacles [12]. Used with permission from van de Panne.

various virtual model control primitives. Specific control goals are allowed for walking controllers such as speed and direction control [11, 23]. Other work have focused on different types of non-locomotions. Jain et al. [21] demonstrate controllers for object dodging and balancing through the use of external support. Faloutsos et al. [14] develop controllers for various rising up motions such as prone-to-stand, supine-to-stand, kip-up, etc. Ha et al. [18] develop falling and landing controllers for a human character.

FSM-based controllers can be refined with offline optimization. Wang et al. [35] extend the work of [41] and optimize the controller for various objectives including energy and style. This creates a more natural walk controller. Geijtenbeek et al. [17] use offline optimization to produce motions that track motion capture data. Optimization can also be used to discover new motion styles that are hard to develop manually. Agrawal et al. [7] used diverse optimization techniques to synthesize a range of motions in the same family but with noticeably different styles.

Our work uses the FSM-based approach. It is closest in spirit to [7, 12, 41]. While most previous work on FSM-based methods focus on a family of closely related motion controllers, our work produces a wide range of different motions, including kip-up, walk, sit-to-stand, prone-to-stand.

Chapter 3

Motion Primitives

In this chapter we describe the control strategy used in this thesis. It belongs to the class of finite state machine controls schemes used in [12, 41], i.e., we use phase-based control scheme to author the motions of physics-based characters. Each phase carries out a portion of the motion, e.g. the support phase of a walk cycle. The phase-based control structure is illustrated in Figure 3.1. A phase remains active until some interesting event has happened, and it then transitions to the next phase. Scheduled timeouts or new contacts with the environment typically result in a transition to a new phase.

Within each phase, several motion control primitives can be used in combination to achieve the desired goal. Our system integrates several motion control primitives that were demonstrated to work well in [12, 17, 18]. Figure 3.2 gives an overview of the control structure and motion primitives used within each phase. The remainder of this chapter will cover all the motion primitives used in this thesis as well as the phase transition model. The primitives collectively produce a net torque for each joint. The torque is given by

$$\tau = \tau_{PD} + \tau_{VF} + \tau_{GC} + \tau_{QB} \tag{3.1}$$

where *VF* represents the desired virtual force, *GC* represents the gravity compensation force, *QB* represents the quiescent balance.

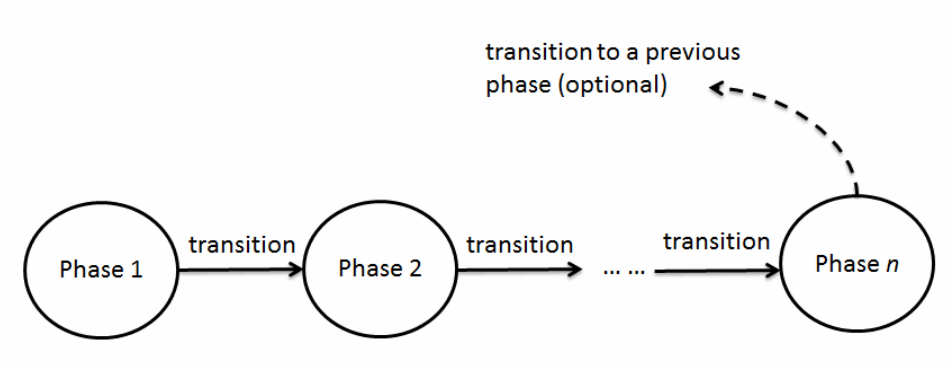


Figure 3.1: Phase diagram

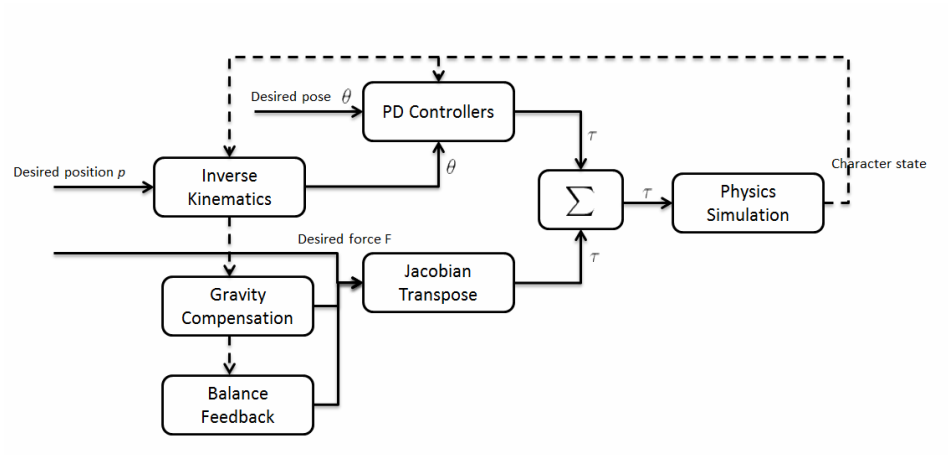


Figure 3.2: Motion primitives and high level controller structure

3.1 PD Control on Joints

Proportional-derivative (PD) controls are a basic method to track desired joint angle trajectories. At every simulation step, a PD controller computes the torque for each joint based on the desired joint angle θ_d, the current joint angle θ and the joint's angular velocity $\dot{\theta}$:

$$\tau_{PD} = k_p(\theta_d - \theta) - k_d\dot{\theta} \tag{3.2}$$

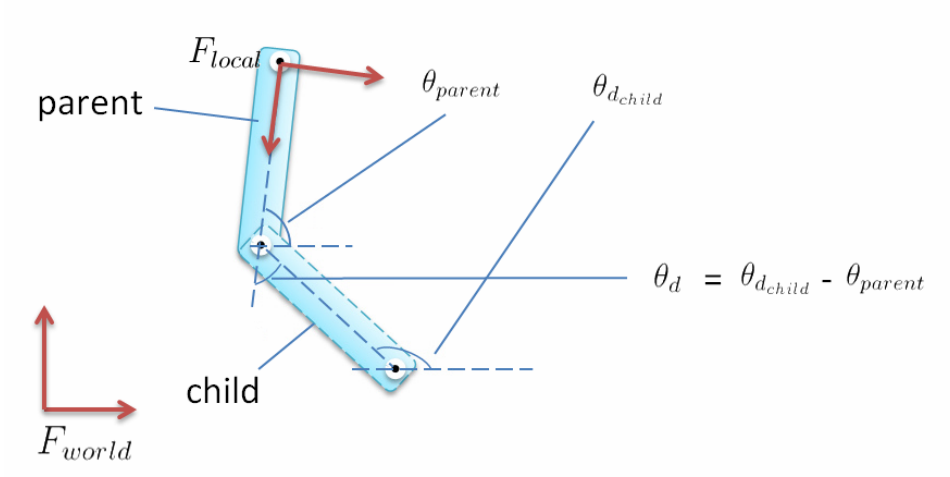


Figure 3.3: PD control tracking a child's angle in world coordinate frame. The desired angle of the child $\theta_{d_{child}}$ in world coordinate frame is converted to the desired local joint angle θ_d in local coordinate frame.

where k_p and k_d are proportional gain and derivative gain parameters that govern the responsiveness and stiffness of the joint motions. We allow for different k_p and k_d gains for each joint. Most joints use low-gain PD controllers in order to achieve compliant motions that are less stiff and therefore more natural.

PD controls are used to track the desired angle of joints in either the local joint coordinate frame or in the world coordinate frame. In the case of tracking in the world coordinate frame, a desired angle of the joint's child or parent can be specified in the world coordinate frame. This is then converted to a desired angle of that joint in local coordinate frame, and a joint-local PD control is used to drive the joint to that angle. An example is shown in Figure 3.3 that tracks the child link's angle in world coordinate frame.

In practice, the joint angles do not exactly follow the desired joint angle trajectories because the dynamics of the joint can be affected by the other parts of the body system. PD controls by themselves also do not take external forces into account such as gravity. Joints that support significant weight or that support large contact forces are prone to large errors. In order to alleviate the errors, we need the help of virtual force and gravity compensation. These "feed-forward" mechanisms

will be discussed in the next two sections.

3.2 Virtual Force

Virtual forces allow joints to be abstracted away and instead allow for motions to be controlled in Cartesian space. They help counter external forces, and can help in achieving desired end-effector trajectories. The virtual force technique works by generating a set of joint torques along a chain of links that effectively results in a desired force on one end of the chain. To apply a virtual force to a body link, a desired force is specified on that link of application, and one of the links is selected to serve as the root or "base" of the kinematic chain. Along the chain of links between the link of application and the base link, torques are automatically computed for all these joints. The virtual force technique can be used to accelerate the body, to compensate for gravity, and to achieve given forces on an end effector.

At any given time, a force, F, applied on a given point on a kinematic chain produces a power, P, given by

$$P = F^\top \cdot v$$

where $F^\top$ is the transpose of the force, and v is the velocity of the point of application. To produce the same power by using internal torques, the internal torques along the kinematic chain to the base link also has to satisfy

$$P = \tau^\top \cdot \omega$$

where τ and ω are the joint torques and angular velocity along all the joints. The variables v and ω are furthermore related by the Jacobian matrix:

$$v = J \cdot \omega$$

Therefore we have

$$F^\top \cdot v = \tau^\top \cdot \omega$$

Substituting v with $J \cdot \omega$ we get :

$$F^\top \cdot J = \tau^\top$$

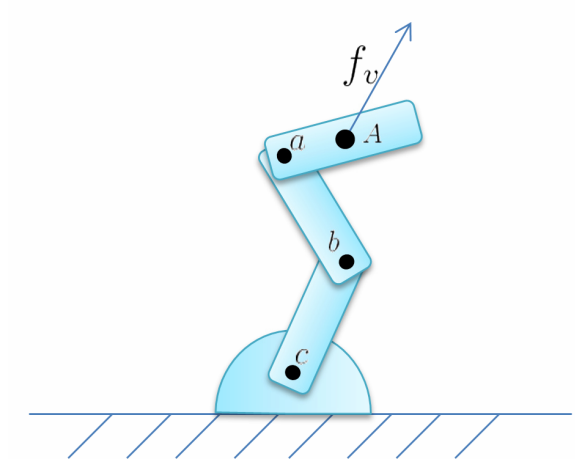


Figure 3.4: Virtual force of on a three-link arm

and equivalently,

$$\tau = J^{\top} \cdot F \tag{3.3}$$

For the 2-D three link chain illustrated in Figure 3.4, the Jacobian is defined as following:

$$J = \begin{bmatrix} \partial v_x/\partial \omega_a & \partial v_x/\partial \omega_b & \partial v_x/\partial \omega_c \\ \partial v_y/\partial \omega_a & \partial v_y/\partial \omega_b & \partial v_y/\partial \omega_c \end{bmatrix}$$

where v_x and v_y are the horizontal and vertical velocity of the point of interest and ω_a, ω_b and ω_c are the angular velocities of the three joints. For this specific example, this can be reduced to

$$J = \begin{bmatrix} y_a - y_A & y_b - y_A & y_c - y_A \\ x_A - x_a & x_A - x_b & x_A - x_c \end{bmatrix}$$

This yields the torque as

$$\tau = J^{\top} \cdot F = \begin{bmatrix} (y_a - y_A) f_{vx} + (x_A - x_a) f_{vy} \\ (y_b - y_A) f_{vx} + (x_A - x_b) f_{vy} \\ (y_c - y_A) f_{vx} + (x_A - x_c) f_{vy} \end{bmatrix}$$

The implementation of virtual forces in our framework allow for the application

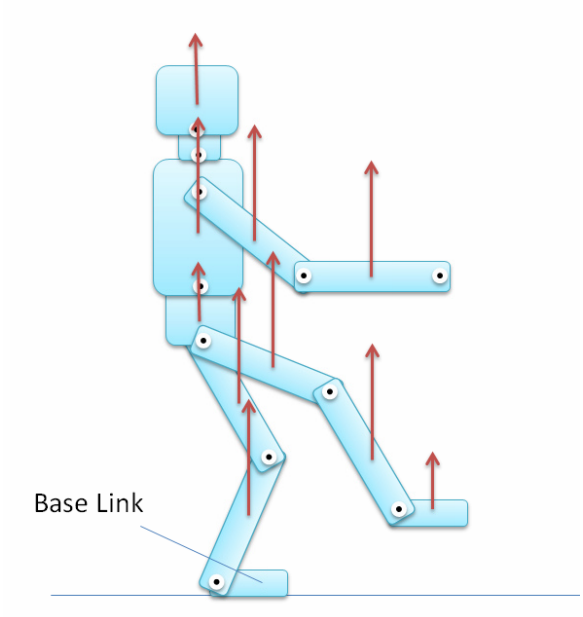


Figure 3.5: Gravity compensation on physics-based character

of virtual forces on arbitrary chains of our planar character model.

3.3 Gravity Compensation

Gravity compensation (GC) is one application of virtual force. It works by producing joint torques to help body links cancel the impact of gravity. This allows simple PD control to achieve more accurate joint angle tracking, and is therefore helpful in a variety of situations. When using GC on a humanoid character, it is assumed that one of the foot is firmly planted on ground and that the corresponding foot thus serves as the base link. A virtual force $F_i = -m_i g$ is applied to every body link, b_i, as shown in Figure 3.5. The resulting joint torques from these virtual forces are aggregated for each joint. In the case where two feet are both well planted on the ground, we choose to distribute the virtual forces arising from upper bodies evenly to the two base links, i.e. both feet.

In practice, it is sometimes useful to use an approximation when computing gravity compensation for the support limbs. We can apply a virtual force of $-Mg$

to the CoM of the torso, where M is the total mass of the entire body. This allows for better PD control over the lower body, and also makes the task of balance easier.

3.4 Quiescent Stance Balance Feedback

This type of feedback helps the character remain statically balanced while standing on the ground. The mechanism uses a feedback rule to compute a horizontal virtual force that regulates the CoM position in order to maintain balance. The goal is to keep the CoM right above the middle of the supporting foot span. The virtual force is computed by a horizontal PD controller:

$$f = k_p(x_d - x_{com}) - k_d\dot{x}_{com} \tag{3.4}$$

where x_d is the middle point of foot span, x_{com} is the horizontal projection of the full body CoM, $\dot{x}_{com}$ is full body CoM's velocity and k_p and k_d are the adjustable proportional and derivative parameters that have been seen in the PD controllers.

3.5 Inverse Kinematics

Inverse Kinematics (IK) is a method to determine joint angles required to achieve a desired end effector position in Cartesian coordinates. Certain motions are more easily controlled in Cartesian space, e.g., reaching a location with the hand. For our planar humanoid character, IK is made available for the two-link limb structures, i.e. arms and legs. A user can specify the reaching target of the end effector, i.e. wrists and ankles, in Cartesian coordinates. The angle of the first joint along the link will be computed in the world coordinate frame and that of the second joint will be computed in the local coordinate frame. These two joints are then tracked by the PD controllers to the computed desired angles in their corresponding coordinate frames.

There are two solutions for two link planar IK. One of them represents the case where middle joint bends forward and the other solution represents backward bending. These are typically referred to as the "elbow up" and "elbow down" solutions in robotics. Since the elbow joint for a human figure only bends one way, its IK has a unique solution. The computation of IK for the arms is illustrated in

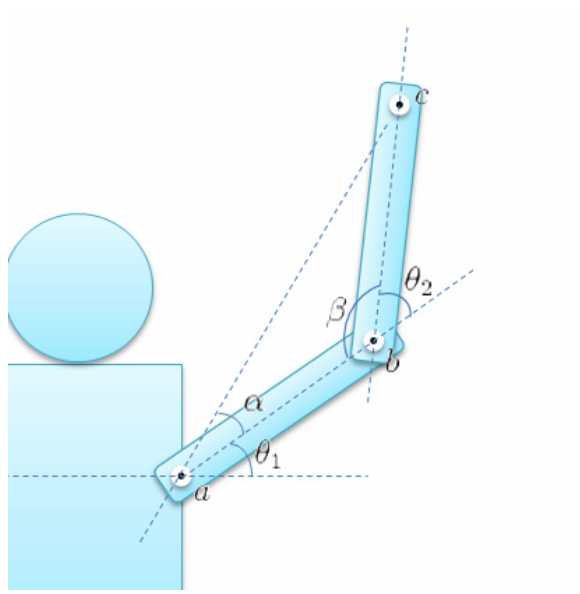


Figure 3.6: Sagittal view of inverse kinematics of two-link arm. Given the target position of c as input, IK finds the angle θ_1 in world frame, θ_2 for joint a and b.

Figure 3.6. The position of the wrist, c, is specified by the user. The position of the shoulder, a, is obtained at run time from the simulation. The lengths of two links l_{ab} and l_{bc} are predefined in the character specification. Therefore, for the given two link case, the analytical solution can be easily derived using the cosine law:

$$l_{bc}^2 = l_{ab}^2 + l_{bc}^2 - 2l_{ab}l_{ac}\cos\alpha \tag{3.5}$$

$$\Rightarrow \cos\alpha = (l_{ab}^2 + l_{ac}^2 - l_{bc}^2)/2l_{ab}l_{ac} \tag{3.6}$$

The joint angle of the first joint θ_1 in world frame is given by

$$\theta_1 = \mathrm{acos}((c_x - a_x)/l_{ac}) - \alpha \tag{3.7}$$

Likewise, we can find angle $\cos\beta$ using the cosine law, and therefore find the angle β:

$$\cos\beta = (l_{ab}^2 + l_{ab}^2 - l_{ac}^2)/2l_{ab}l_{bc} \tag{3.8}$$

Lastly θ_2 is given by

$$\theta_2 = \pi - \beta \tag{3.9}$$

IK for the legs is computed in an analogous fashion.

3.6 Phase Transition Models

All the planar physics-based character motions are authored using a phase-based finite state machine. Each phase is concerned with a specific task and uses a set of motion primitives to accomplish that task. Each phase will transition to the next phase unless it is the final phase of a non-cyclic motion. A phase transition takes place upon the occurence of a predefined transition event. These events fall into two categories: timeout events and contact change events. A timeout event happens whenever a specified duration has elapsed since the start of the phase. A contact change event happens whenever one of a specified set of links has a change in contact, i.e. a body link has just established contact or has just lost contact with its surrounding environment. With these simple transition models, many motions can be easily divided into distinct phases. In the next chapter we further discuss specific implementation details.

Chapter 4

Simple Controller Language (SCL) Framework

In this work, we build a simple controller language (SCL) framework that is capable of loading a motion script at runtime and produce a physics-based motion that is simulated in realtime. The scripts are written in a SCL, and they are interpreted by a scripting engine. Once a motion script is interpreted, it is loaded into the application runtime and is represented as a phase-based motion controller where each phase consists of the motion primitives described in the previous chapter. The controller is then applied on a 2D planar humanoid model for dynamics simulation. The initial state of the simulation can be specified by the animator: a predefined initial pose and a predefined static environment can be selected together to form the initial state of the simulation. The physics simulation then produces a motion in realtime.

The scripting framework allows users to use a text-based language for authoring motions using an expressive and flexible syntax. The language provides a built-in set of primitives that are mentioned in the previous chapter. With such a language, our framework opens a window for non-programmers to create physics-based motions and for animators to share motions with each other. It also potentially shortens the test cycle of motion design by allowing modifying and re-loading a motion at runtime in a semi-interactive manner.

In this chapter, we will describe the SCL in detail and show some motions

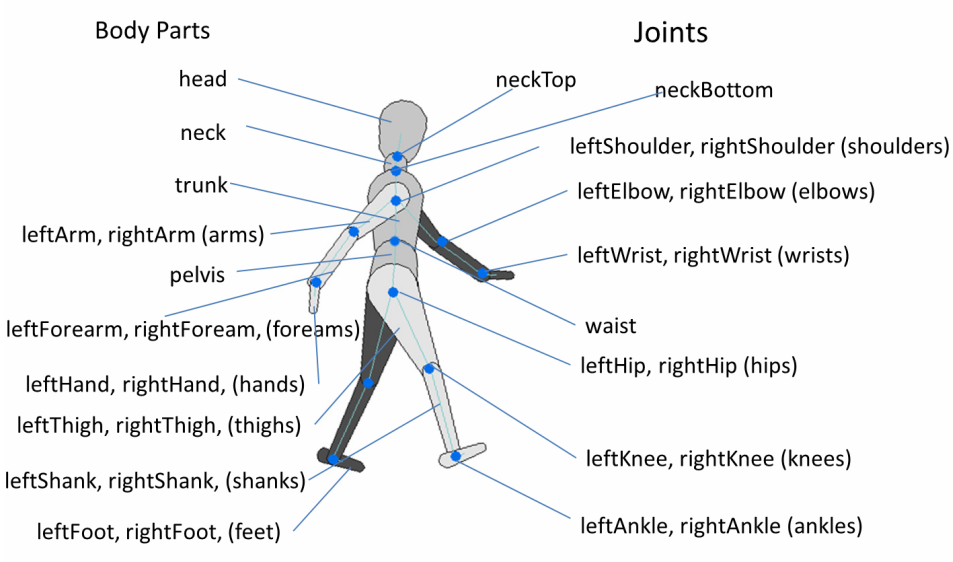


Figure 4.1: Character Anatomy. Left-right symmetric structures have only one-half of the body annotated. The plural name of the symmetrical structures are given in parentheses.

authored using the SCL including kip-up, sit-to-stand, walk and prone-to-stand.

4.1 Character Definition

The character has predefined link dimensions, masses, joint angle limits and joint PD parameters that are held fixed across all simulations. A controller author needs to know the names of the joints and links in order to apply motion primitives to the intended joints and links using the SCL. The character anatomy is illustrated in Figure 4.1. The character is a 2D model in the sagittal plane, consisting of 16 rigid links, 15 one-DOF joints, and a total mass of $89.49kg$. Each body part is modelled as a trapezoid with two semi-circular end caps. The simulation neglects self collisions within the body. The specification of all body parts can be found in Table 4.1. A joint angle is defined as a child link's orientation relative to the orientation of its parent. The head is the root link. For each joint, the parent link is the one that is proximal to the head. The *zero pose* is defined as a stiff straight pose with hands and feet pointing to the left, and head pointing to the

Body Part	Length (cm)	Tip Thickness (cm)	Mass (kg)
head	35	26, 22	5.89
neck	2	12, 12	1.1
trunk	50	30, 20	29.27
pelvis	25	24, 27	16.61
arm	32	16, 8	2.79
forearm	32	8.4, 8.4	1.12
hand	17	5, 4	0.55
thigh	50	28, 10	8.35
shank	48	10, 7	4.16
foot	22.6	8, 5.2	1.34

Table 4.1: Body parts specifications taken from Wooten et al. [40]

Body Joint	Angle Range (°)	k_p ($\frac{Nm}{rad}$), k_d ($\frac{N \cdot m \cdot s}{rad}$)	Max Torque (Nm)
neckTop	-30, 90	170, 5	60
neckBottom	-30, 30	171, 18	122
waist	-40, 90	674, 61	250
shoulder	-90, 180	386, 33	180
elbow	0, 160	300, 20	180
wrist	-90, 90	156, 11	120
hip	-30, 100	500, 70	300
knee	-160, 0	600, 70	300
ankle	30, 150	500, 60	300

Table 4.2: Body joints specifications

right. This is illustrated in Figure 4.2. When the character is in the zero pose, the world orientation of each link as well as each local joint angle is defined to be zero. The rotation of a link is defined as positive when it turns counter-clockwise. The rotation of a joint is positive when the its child link rotates in the positive direction. The k_p parameter is set to be positively correlated with the strength of joints and k_d is set to be approximately $0.1k_p$. One exception is that k_p is relatively low for *neckTop* in order to avoid motion instabilities for the neck and head for our chosen simulation time steps. The joint limits, torque limits, and k_p k_d parameters of each joint are empirically tuned and they are given in Table 4.2.

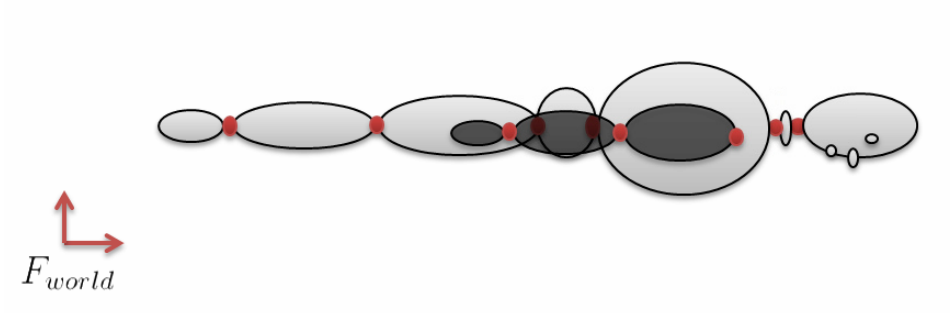


Figure 4.2: An illustration of the zero pose. In this figure, the head points to the positive direction of the horizontal axis and limbs point to the negative direction of the horizontal axis. The zero pose defines the zero angle for each link in the world coordinate frame as well as the zero angle for each joint in the local joint coordinate frame. Note that the zero pose cannot be actually achieved due to the joint limit of the ankles as defined in Table 4.2.

4.2 Specification of Phases and Phase Transitions

The controller framework reads in controller description that are defined in the SCL syntax that we will describe in the following two sections. In this section we focus on the specification of phase and phase transitions.

The controllers have a phase-based structure. A motion and its corresponding controller consist of several phases. In the introductory example shown in Listing 4.1, a two-phase motion is scripted. The first phase attempts to drive the left shoulder to 180°, which is followed 2 seconds later by a second phase that mimics the same behaviour for the right shoulder. Note that the two shoulder joints are named as "*leftShoulder*" and "*rightShoulder*" in the SCL. A full list of joint names and link names can be found in Figure 4.1.

Listing 4.1: A Simple Motion Script

```
{
  (left_shoulder_raise)
  [leftShoulder 180]
}
after 2
```

```
6 {
7   (right_shoulder_raise)
8   [rightShoulder 180]
9 }
```

Each motion phase is specified within a pair of brackets "{ }" as seen in the example. Within each phase specification, one can define an optional phase name between "( )", as seen in line 2 and 7. A phase name can be used to describe its purpose, and can also serve as a unique label that can be referred to later. The remainder of each phase specification consists of one or more motion primitives. In Listing 4.1, the two motion primitives are the PD controllers on the left and right shoulder, which drive the shoulders to 180° relative to the torso. The details for control primitive specifications including PD control will be further described in the next section.

Every phase must be followed by a transition rule with the exception of the last phase. The rule states the condition of the transition and is followed by the next phase that it transitions into. In the above example, the transition rule between the two phases is a simple time-based rule that deactivates the *left_shoulder_run* phase and activates the *right_shoulder_run* phase after 2 seconds spent in the first phase. In Listing 4.1, the second phase is not followed by another phase, and therefore it is the last phase. The last phase remains active thereafter.

A transition rule following the last phase makes the motion cyclic. For example, in Listing 4.2, the last phase (*phase_4*) is followed by a transition specified on line 19 and line 20. This transition is prefixed with a "**finally**" keyword and followed by a special "**backto**" syntax that refers to a previously defined phase name. This instructs the motion to transition back to *phase_1* after *phase_4* is finished. Therefore, the motion control phases repeat in a cyclic fashion.

In addition to being used to build a cyclic motion, a previously defined phase name can also be used to define a symmetric phase. On line 17 of Listing 4.2, *phase_4* is designated as a symmetric duplicate of *phase_3*, using the "(*phase name*):(*existing phase*)" syntax.

Phase transitions fall into two main categories as described in Chapter 3, i.e. *time based* and *contact based*. In addition, the SCL also allows another type of

transition using the conjunction of these two main types. A complete list of transition types are shown in Table 4.3.

Listing 4.2: A Simple Script Illustrating the Transition Models

```
1  {
2    (phase_1)
3    [shoulders 120]
4  }
5  after 2
6  {
7    (phase_2)
8    [shoulders 0, elbows 0, wrists -40]
9  }
10 when contact hands
11 {
12   (phase_3)
13   [leftElbow 90]
14 }
15 after 2
16 {
17   (phase_4) : (phase_3)
18 }
19 finally after 3
20 backto phase_1
```

The two main types of transitions are both demonstrated in Listing 4.2. Time based transitions are used on line 5, 15 and 19. Line 10 specifies a contact-based transition. It states that *phase_2* should transition into *phase_3* when the hands make contact with the environment. The third transition type will be presented later in the chapter.

Transition Type	Specification Syntax	Comment
Time based	**after** *n*	Transition happens after *n* seconds
Contact based	**when contact** (changed | established | lost)? (any | all)? *body_link(s)*	Transition happens when any (or all) of the specified *body_link(s)* has just changed its contact state.
Contact and Time based	**and** *contact based transition time based transition*	Transition happens when both conditions are satisfied

Table 4.3: Transition types

4.3 Motion Primitive Specifications

A phase can contain zero or more motion primitives. Each motion primitive is specified in a pair of square brackets "[]". A motion primitive normally starts with a keyword that represents its type. There are 6 types of motion primitives. They are briefly listed in Table 4.4.

PD Controller. Each PD controller primitive specification defines the PD controller for one or more joints and it does not require a keyword. Its syntax is demonstrated in Listing 4.3. On line 1, PD controllers are applied on three joints with different target angles. The target angle of each joint moves from its current angle to the specified target angle.

Listing 4.3: PD control primitive

```
1 [joint_1 90, joint_2 50, joint_3 40 time: 1.2]
2 [joint_1 90 child, joint_2 50 parent]
```

At each time step, the actual target angle is linearly interpolated between the two angles over a duration of "time: 1.2", i.e. 1.2 seconds. "time" keyword is optional: when it's not specified, PD control tracks to the specified target angle instantly. On line 2, PD controllers are used to track the child and parent links of the two specified joints respectively in world coordinate frame.

When not explicitly specified, a default PD controller is active on each joint. The default PD controller tracks the joint to a desired angle that is inherited from the previous phase or to its starting pose when there is no previous phase. Therefore, when a phase contains no explicitly specified motion primitives, all the joint

Type	Keyword	Description
PD Controller	Not needed	Applies a PD control a set of joints
Virtual Force	*vf*	Applies a virtual force on a body part
Stance Balance Feedback	*sb*	Applies a balance feedback force to the torso
IK	*ik*	Computes joint target angles using IK and applies PD controls on those joints
Relax Joints	*relax*	Reduces the torques produced by PD Controller to make behave more passively
Symmetric primitive	*symm*	Duplicate the last motion primitive and apply it on the symmetric structure

Table 4.4: Motion primitive types

torques are computed from these default PD controllers. The default PD controller is overridden when the relative joint has a explicit PD controller, or the joint is controlled by the *inverse kinematics* or the *relax* primitive as we will describe shortly.

Virtual Force. A virtual force primitive begins with the keyword "vf". It specifies a force, a link of application, and the joint that is connected to what is considered to be the base link for the virtual force. In Listing 4.4, the demonstrated virtual force primitive applies a force, $F = (300N, 300N)$ to *body_1*, with *joint_1* being the first joint from the base link. Figure 4.3 shows the virtual force primitive with the right forearm specified as *body_1* and the left ankle specified as *joint_1*. Thus the base link is the left foot.

Listing 4.4: Virtual force primitive

```
[vf (300, 300) on: body_1, by: joint_1]
```

Quiescent Stance Balance. A quiescent stance balance feedback primitive is used within a virtual force primitive, and it is denoted by the keyword "sb". Its common usage is illustrated in Listing 4.5.

Listing 4.5: Virtual force with stance balance primitive

```
[vf sb(300, 30) on: trunk, by: leftAnkle]
```

The quiescent stance balance feedback primitive computes a desired virtual force using Equation 3.4. The parameters k_p and k_d is chosen to be 300 and 30 respec-

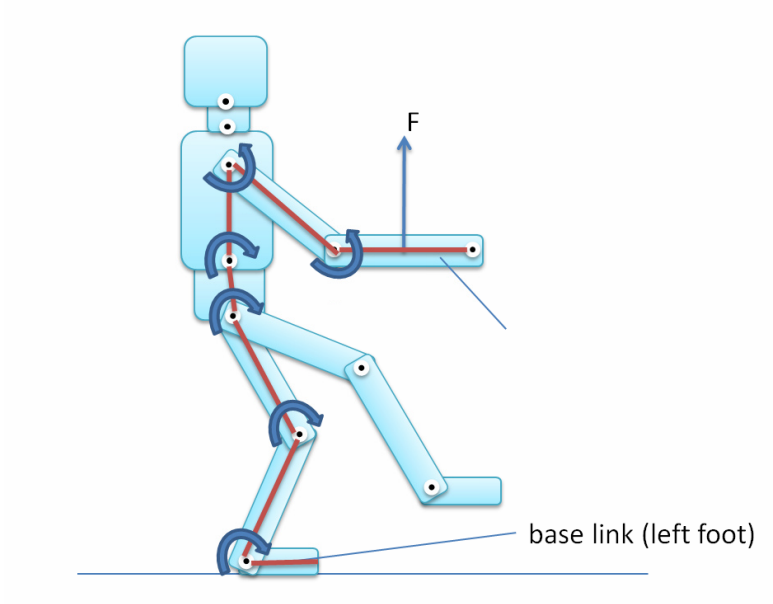


Figure 4.3: The virtual force primitive. The left foot is the base link and the right forearm is the link of application.

tively in Listing 4.5. The desired virtual force is applied to the trunk in order to have a large impact on the full body CoM. When a foot is chosen as the base link, the corresponding ankle will be used as the first joint from the base.

Inverse Kinematics. The inverse kinematics primitive begins with the "ik" keyword. It needs to specify a sequence of target positions for the end effector positions to reach. In Listing 4.6, the IK primitive instructs the right ankle to sequentially reach two positions relative to the position in world coordinates that the left ankle occupied at the beginning of the phase. In our framework, the IK primitive are designed for two-link chains. Therefore, in Listing 4.6 the base of the IK chain is the right hip. Similar to PD, the IK primitive tracks the end effector target locations in a piecewise linear fashion. The intermediate target locations are assumed to be equally spaced in time. The IK primitive overrides the target angle of the relevant joints specified by the default PD controllers.

Listing 4.6: Virtual force with stance balance primitive

```
[ik (0.25, 0.30), (0.20, -0.05) on: rightAnkle rel:
    leftAnkle time: 0.35]
```

Relax Primitive. The *relax* primitive sets the target joint angle of PD controller to its current joint angle at each simulation time step, and scales down the joint torques produced by the PD controller, using the strength parameter to denote the scale factor between 0 and 1, as shown in Equation 4.1 and 4.2.

$$\theta_d = \theta \tag{4.1}$$

$$\tau_{pd} = c_{scale} \cdot (k_p(\theta_d - \theta) - k_d\dot{\theta}) = -c_{scale} \cdot k_d\dot{\theta} \tag{4.2}$$

where θ_d is the desired joint angle and is set to the current joint angle θ. This effectively eliminates the proportional term from the PD controller, leaving only the scaled derivative term which serves as a damper (Equation 4.2). The syntax of the relax primitive is shown in Listing 4.7.

Listing 4.7: Virtual force with stance balance primitive

```
[relax joint strength scale]
```

Symmetric Primitive. The symmetric primitive is used as "[symm]" and it does not need extra parameters. It is required to follow another primitive, and this clones the previous primitive to a symmetric counterpart of the body. For example, using this primitive, a virtual force that affects a chain containing the left leg can be duplicated to the symmetric chain containing the right leg.

4.4 Motion Design Workflow

In general, authoring a controller requires knowledge of the phases useful for structuring a motion and a qualitative understanding of the physics of the motion. A motion needs to be divided into a proper sequence of phases. For example, a sit-to-stand motion can consist of two phases: the first phase shifts the CoM of body forward, and the second raises the CoM. A well chosen phase structure often simplifies the motion design. It is also important for the author to understand the goal of each phase. This aids in the incremental design of a motion. A useful motion

design strategy is to test all the previous phases and ensure they end in an approximate state, before designing the next phase. In the previous example of sit-to-stand motion, before an author designs the second phase, it is important to ensure that the goal of shifting CoM of body out of the chair is achieved at the end of first phase. The author can then proceed to the design of the second phase. Knowledge of the qualitative physics is important as it helps the author to select the suitable motion primitives to achieve the desired goal of a phase. For example, to gain angular momentum, one might need to rapidly swing extended arms instead of keeping the elbows flexed.

Motion design is a trial-and-error process. Guidelines for authoring a motion can be summarized as follows:

1. define the motion phases, with a clear understanding of what each phase should accomplish. A reference motion or motion tutorial for the real human motion can be helpful in achieving this.

2. beginning with the first phase, select a set of primitives that would be useful in accomplishing the desired goal. Test these primitives and use the simulation results to refine the motion primitive parameters. Test with different parameters until the desired end state is achieved.

3. select a transition condition for moving to the next phase.

4. Repeat steps 2-3 until the motion is successfully completed.

With practice, a motion can typically be authored in minutes or hours if a good reference motion is available. In the remainder of the chapter, we will present motions that are authored using the scripting framework and outline the intuition behind choosing the respective motion primitives.

4.5 Results

We present a number of motions scripted using the SCL including statically-balanced motions and highly dynamic and acrobatic motions. The robustness of motion controllers can largely depend on the nature of the motion. As can be expected,

controllers that deal with static-balance tasks are more robust than those dealing with more dynamic motions. For example, sit-to-stand is more robust than kip-up. However, we note that the robustness of controllers is not the primary concern of this framework. Instead, we focus on how to quickly create a successful motion that looks physically plausible and could be easily reused and adapted by others.

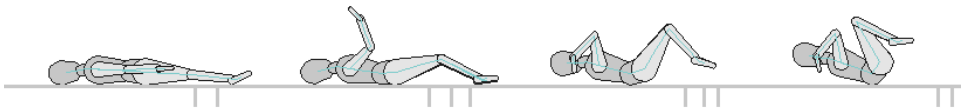
We use JBox2D [4] as the physics simulation engine with a simulation time step of 0.25ms. The position iterations and velocity iterations per physics steps are 3 and 8 respectively. Ground contact is modelled in JBox2D using impulsive forces and unilateral constraints.

4.5.1 Kip-up Motion

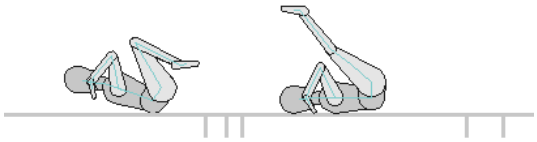
In this section we explore the authoring of a *kip up motion*. In a kip up motion, a humanoid character makes an acrobatic move from a supine position to a standing position by propelling the body away from the ground using leg swings and a push of the hands. It is a highly-dynamic motion with swift momentum change and contact relocation. It also requires careful balance maintenance during the landing and rising up phases. The motion that we develop has five phases. The first two phases prepare for the dynamic leg and arm motion. The third phase does thrust and the fourth phase performs the landing. The last phase takes care of standing up while maintaining balance. Figure 4.4 and Figure 4.5 show the motion with these five phases.

Listing 4.8: Kip up motion script

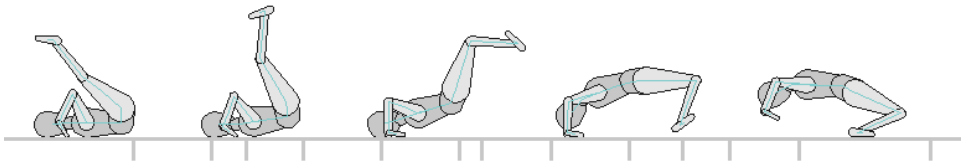
```
{
  (roll_back)
  [shoulders 120, elbows 130, wrists 50 time: 0.3]
  [hips 80, waist 80, knees -170 time: 0.5]
}
after 0.6 // The motion will transition to the next phase
    after 0.6 seconds
{
  (straighten)
  [waist 90, hips 50, knees 0 time: 0]
  [vf (0, 200) on: leftFoot, by: waist]
```



(a) Phase 1: Rollback



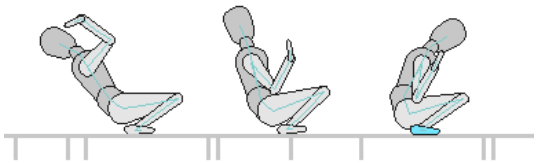
(b) Phase 2: Straighten Legs



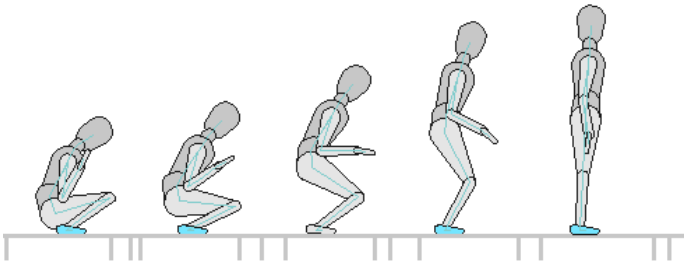
(c) Phase 3: Propel

Figure 4.4: Kip-up Phases 1-3

```
  [symm]
}
after 0.3
{
  (propel)
  // thrust with hips and waist
  [hips 0, waist -20 time: 0]
  [vf (300, 0) on: leftShank, by: waist]
  [symm]

```



(a) Phase 4: Landing



(b) Phase 5: Standing up

Figure 4.5: Kip-up Phases 4-5

```
  // knees and ankles are folded
  [knees -180 time: 0.45]
  [ankles 150 time: 0]

  // hands push the ground, relax to turn down pd control
  [relax shoulders, elbows strength 0.1]
  [vf (0, -200) on: leftHand, by: leftShoulder]
  [symm]
}
when contact all feet
{
  (land)
  // virtual force to strengthen the legs
  [vf (-100, -500) on: leftFoot, by: leftKnee]
```

```
35   [symm]
36   [relax knees, ankles strength 0.4]
37   // rotate the upperbody forward
38   [hips 80, waist 40 time: 0.0]
39   [necks 10 time: 0.3]
40
41       // reduce moment of inertia by folding arms
42   [shoulders 20, elbows 160, wrists 0 time: 0.3]
43 }
44 after 0.6
45 {
46   (stand)
47   [vf sb(1000, 300) on: trunk, by: leftAnkle]
48   [symm]
49   [vf (0, 200) on: trunk, by: leftAnkle]
50   [symm]
51   [knees 0, hips 0, waist 0 time: 1.2]
52   [relax ankles, shoulders, elbows, wrists strength 0.1]
53   [necks 0]
54 }
```

The script for the entire motion is shown in Listing 4.8. The motion is loosely based on a kip up tutorial [5]. The following section briefly describes the design intuition of each phase.

- **Rolling back**. (line 1-5) The first phase starts from the initial supine position. At the end of this phase, it is desired that the legs are flexed back towards the torso and hands curled towards the neck while the shoulders remain close to the ground. These are achieved by applying PD controls on the relevant joints as on line 3 and 4, i.e. waists, hips, knees, shoulders, elbows and wrists. This phase is relatively easy to get correct. An important point to note is that the knees need to bend at a sharp angle to prevent the upper body from being pivoted upwards by the lower body movement. The transition from the phase is time based (line 6), and the transition time was chosen empirically. The intuition for the transition point is that this phase should end when the back is not too high above the ground and there is some

upward momentum for the whole body.

- **Straighten up legs**. (line 7-12) The goal of the second phase is to gain upwards momentum. This is done by applying a PD control to straighten up the legs and point the feet upwards (line 9). An upward virtual force (line 10) is applied on the foot in order to gain momentum more rapidly. The timeout is chosen by intuition similar to the previous phase.

- **Propelling the body** (line 13-29). This phase aims to gain linear and angular momentum for the whole body. It does so by thrusting the legs away from the body and down to the ground (line 17-19) in coordination with hands pushing off the ground using a virtual force (line 26-28). Notice that in order for the hand push to work well, we "relaxed" the upper limbs (line 26) so that torques generated by the virtual force dominate the upper limb joints. Knees and ankles are flexed (line 22-23) in order to prevent unexpected contact with the ground and to prepare for a better landing where the approaching angle of the foot with respect to the ground is almost 0.

- **Landing**. The landing phase starts right after both feet are in contact with the ground, and the goal is to bring CoM above the support of the feet span and also to make sure the character ends in a statically balanced crouching pose. A virtual force (line 34-35) that pushes the feet downwards is needed to make the knees and ankles strong enough to absorb the landing impact. The upper body's moment of inertia is reduced by folding arms (line 42). The angular momentum gained from the previous phase helps to rotate upper body forward (line 38-39).

- **Standing up**. At the beginning of this phase, the CoM of the body has fallen within the support of the foot span, and the goal is for the character to rise with feet on the ground. The stance balance feedback primitive is turned on (line 47-48) to keep the horizontal position of CoM within character's supporting polygon given by the span of its feet. Stand-up motion is achieved by tracking the knees, hips to their zero configuration as on line 51. An upward virtual force is applied on the torso to approximately compensate for gravity as on line 49-50. The ankles are "relaxed" (line 52) and thus the

rotations of these two joints are dominated by the torques from the balance virtual force and the constant upward virtual force.

4.5.2 Sit to Stand Motion

In this section, we demonstrate a sit-to-stand motion. The initial state of the motion sees a character sitting in a chair. The motion scripts consists of two phases. In the first phase, the character leans its body forward so that the CoM is eventually above the support region of the feet. In the second phase, the character stands up in the same way as in the last phase of kip motion. This motion is fairly easy to author. It can take less than 10 minutes when the user has a basic understanding of a stand up motion. The motion is illustrated in Figure 4.6.

Listing 4.9: Sit To Stand Script

```
{
  (lean)
  [waist 20, hips 120 time: 0.3]
}
after 0.4
{
  (standup)
  [vf sb(400, 300) on: trunk, by:leftAnkle]
  [symm]

  [vf (0, 400) on: trunk, by:leftAnkle]
  [symm]
  [knees 0, hips 0, waist 0, necks 0 time: 0.9]
  [relax ankles, shoulders, elbows, wrists strength 0.1]
}
```

4.5.3 Walk Motion

The walk motion given in Listing 4.10 consists of two symmetric phases. With our framework, it is possible to author just one phase and then to use symmetry to define the other phase. Reuse of the phase definition makes it easier to test

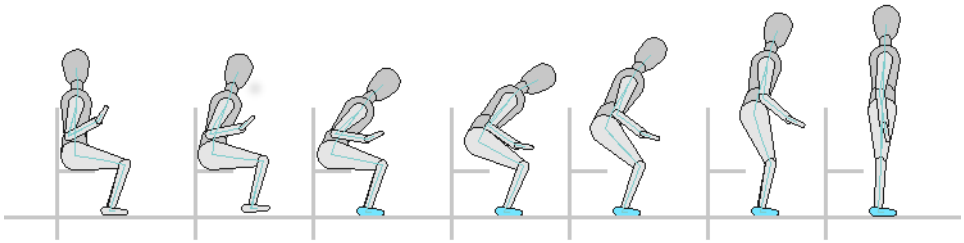


Figure 4.6: Sit to stand motion sequence

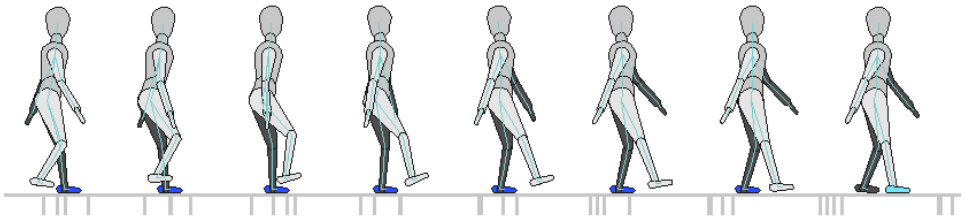
the script because it reduces the chance of subtle human-made errors that comes from inconsistent parameters of the two symmetric phases. Each phase makes the transition using the conjunction of a time based and contact based rule. These two conditions prevent a transition from occurring before the swing foot has left the ground. Each of the phases is composed of four parts: pose of the upper body, pose of the swing leg, pose of the stance leg, and virtual forces that assists the movement. For the upper body poses, on line 5, it tracks the head and trunk's angle to 90 degrees, i.e. up-right in world space. Lines 6 and 7 takes care of arm swings by tracking shoulders and relaxing elbows and wrists to make them more passive. Line 10 and 11 controls the swing foot. The IK on line 10 instructs left wrist to reach two sequential target positions relative to right ankle's starting position at the beginning of the phase. This affects the joint angles of swing knee and swing hip. For the stance foot, line 14 uses left hip to track the world orientation of the pelvis, and it also sets knee to flex. Line 15 "relaxes" the stance ankle from PD controller. The torque for this ankle will be mainly determined by virtual forces. The virtual force primitives (lines 18 to 20) contains both vertical and horizontal components, and therefore help the character to propel itself forward and to compensate for the gravity. Line 18 does gravity compensation for the swing leg. This helps the IK mechanism to be more accurate. Although an accurate full-body gravity compensation will result in a more accurate pose tracking, it is sufficient to apply an approximate gravity compensation force to the trunk. Line 19 uses the stance knee as the base joint to partially compensate the gravity for the entire body (600N) and to propel the body forward (150N). The stance ankle's torque is critical to control

the ground reaction force. Line 20 uses the stance ankle as the base joint to push swing leg, and it effectively causes the stance foot to obtain a ground reaction force that has a horizontal component that pushes the body forward, i.e., to the right.

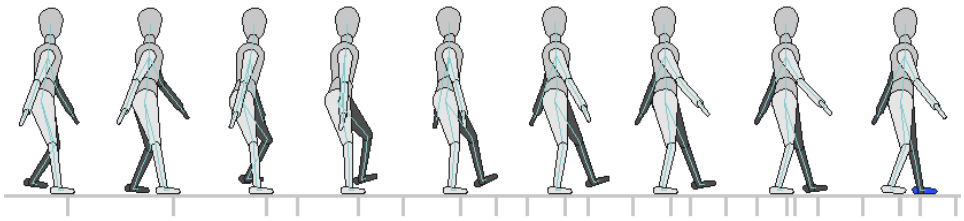
A complete cycle of the walk motion is illustrated in Figure 4.7. The walking motion is not yet very natural and is included here in order to illustrate how walking can be authored in our framework. As with other walking simulations, we expect that the motion could be refined with optimization. In the following chapter, we describe how motions can be optimized.

Listing 4.10: Walk Motion

```
{
  (right_swing)

  // upperbody pose
  [neckTop 90 parent, waist 90 parent]
  [leftShoulder 30, rightShoulder -25 time: 0.4]
  [relax elbows, wrists strength 0.05]

  // swing foot
  [ik (0.25, 0.30), (0.20, -0.05) on: rightAnkle rel:
     leftAnkle time: 0.35]
  [rightAnkle 180 child]

    // stance foot
  [leftHip 90 parent, leftKnee -30]
  [relax leftAnkle strength 0.4]

  // virtual forces and gravity compensations
  [vf (0, 110) on: rightShank, by: rightHip]
  [vf (150, 400) on: trunk, by: leftKnee]
  [vf (60, 0) on: leftThigh, by: leftAnkle]
}
and when contact rightFoot after 0.3
{
  (left_swing) : (right_swing)
}
```



(a) Phase 1: right leg swing



(b) Phase 2: left leg swing

Figure 4.7: Walk motion sequence

```
finally and when contact leftFoot after 0.3
backto right_swing
```

4.5.4 Prone to Stand

We also demonstrate a seven-phase prone-to-stand motion in which the character rises from a prone position and ends in a stand position. The script is shown in 4.11. All the transition rules and motion primitives used in this script have been reviewed in the previous results. Therefore, we will leave out the details on the motion authoring. The resulting motion can be seen in Figure 4.8.

Listing 4.11: Prone to Stand

```
{
  (place_hands)
  [ik ( 0.05, 0.15), ( 0.20, -0.14)
```

```
   on: wrists rel: leftHip time: 0.5]
  [wrists 180 child time: 0.2]
}
and when contact any hands after 0.2
{
  (shoulder_rise)
  [waist -40, knees -30 time: 0.6]
  [relax shoulders, elbows, wrists strength 0.3]
  [vf ( -50, 300) on: trunk, by: leftWrist ]
  [symm]
}
after 0.8
{
  (hip_move_back)
  [knees -110, hips 40, waist 80,
   elbows 60, wrists 180 child time: 0.5]
  [relax shoulders strength 0.4]
  [vf ( -100, 300) on: trunk, by: leftWrist ]
  [symm]
}
after 0.6
{
  (push)
  [waist 90 parent, hips 100 parent, knees -130 time: 0.4]
  [relax shoulders, elbows strength 0.1]
  [vf ( 200, -300) on: leftHand, by: waist]
  [symm]
}
after 1.0
{
  (right_foot_relo)
  [ik ( 0.10, 0.3), ( 0.28, -0.15)
   on: rightAnkle rel: leftAnkle time: 0.5]
  [vf ( 0, 500) on: trunk, by: leftKnee]
  [rightAnkle 180 child time: 0.4]
  [leftHip 90 parent, waist 90 parent, necks 90 parent
     time: 0.0]
```

```
 [shoulders 0, necks 0 time: 0.0]
 [relax shoulders, elbows, wrists strength 0.2]
}
and when contact rightFoot after 0.5
{
 (left_foot_relo)
 [vf sb( 1000, 800) on: trunk, by: rightAnkle]
 [vf ( 0, 800) on: trunk, by: rightAnkle]
 [rightKnee 0, rightHip 0, waist 0, leftHip 0 time: 1.5]
 [leftAnkle 180 child time: 0.5]
 [ik ( 0.0, -0.08) on:leftAnkle rel: rightAnkle time: 0.5]
 [relax rightAnkle strength 0.2]
 [relax shoulders, elbows, wrists strength 0.2]
}
and when contact leftFoot after 1.0
{
 (rise)
 [vf sb( 400, 400) on: trunk, by: leftAnkle]
 [symm]
 [vf ( 0, 500) on: trunk, by: leftAnkle]
 [symm]
 [relax ankles strength 0.2]
 [relax shoulders, elbows, wrists strength 0.2]
 [knees 0, hips 0, waist 0, necks 0 time: 1.5]
}
```

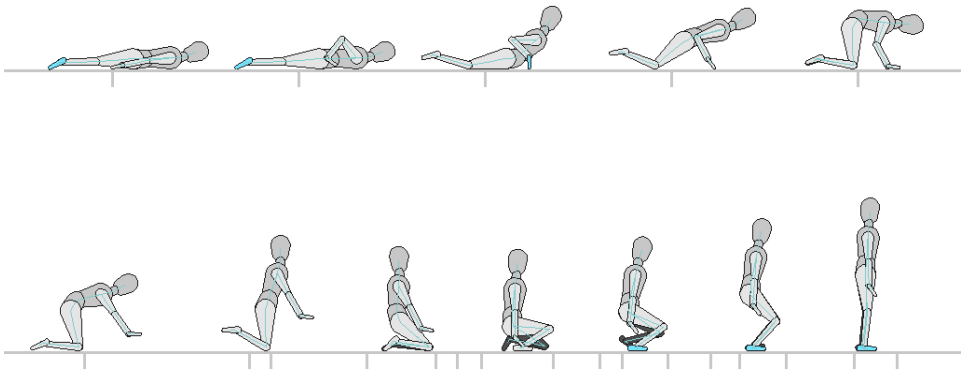


Figure 4.8: Prone-to-stand motion sequence

Chapter 5

Optimization

We have shown that many motions can be authored from the instantiation of several simple primitives using the Simple Scripting Language (SCL). However, an authored motion only exhibits one particular style. It is common that animators want to create different styles of the same motion. For example, given a kip-up motion, they may also wish to develop a faster kip-up. This can be achieved by a parameterized motion controller. Variations of the same motion can be produced by making appropriate choices for the parameter values. In order to produce a particular desired variation of a motion, one can define an objective function to model how well a motion achieves a desired style, and optimize the control parameters for the given objective function. In this chapter, we develop a parameterization and optimization framework that allows animators to specify the parameterization of a scripted motion that is then optimized for a custom defined objective function.

5.1 Problem Definition

We define π_x as a motion control policy parameterized by a vector $x \in \mathbb{R}^N$. The optimization problem can be defined as minimizing the objective function:

$$x^* = \underset{x}{\operatorname{argmin}}\, J(\pi_x)$$

The objective function J is defined by the user and its value is observed during a simulation. The value returned by the objective function can range from a de-

sired kinematic property at a particular instance in time, e.g., maximum height of a human body CoM, to an aggregation of a dynamic property over the course of the entire simulation e.g., total energy used by the motion.

Given the state of physics world s_t at time t, the dynamics simulation produces the world state at the next step $t + \delta t$:

$$s_{t+\delta t} = S(s_t, \delta t, \pi_x)$$

where $s_t, s_{t+\delta t} \in \mathbb{S}$ are vectors in a high dimensional space of all plausible states of the simulated world; δt is the time of each simulation step. The entire motion of duration T is then defined as the sequence $\{s_0, s_1, ...s_i, ..., s_n\}$, where $s_i = S^i(s_0, \delta t, \pi_x)$ and $n \cdot \delta t = T$. In order to obtain information for each simulation step, an evaluation function is defined according to

$$f(s), f : \mathbb{S} \rightarrow \mathbb{R}^k$$

that maps a state to a vector of dimension k. The objective function can then be rewritten as

$$J(\pi_x) = agg_{i=0}^{n}(f(S^i(s_0, \delta t, \pi_x))) \tag{5.1}$$

where agg is an aggregation function that collects all the evaluated values for all the simulation steps, aggregates them using a user-defined rule, typically a weighted summation, and returns a scaler value.

In Equation 5.1, the free variables of optimization are x. The other variables remain fixed during optimization; $\delta t = 0.25ms$ in all simulations; S^i is a deterministic function representing the dynamic simulation. The remaining components that need to be defined by a user are:

- s_0, the initial state as designed by users using a pre-populated list of poses and environments. For example, an initial state can be one with the character standing on ground, lying on ground, sitting on a chair, etc.

- π_x is the parameterization of motion that is defined by the user in the motion script. A user can select the numerical fields in a script and prefix them with a "@" annotation to make them free optimization variables.

- $f(s)$ is a user-defined function that maps the state of the current simulation step to a desired custom-sized vector. For example, a user can define $f(s)$ to return the CoM position of the current simulation step.

- agg is also a user-defined function. It aggregates the custom-sized vectors returned by $f(s)$ for all simulation steps, and returns a scalar value that will be used by optimization as the objective.

- n is the number of simulation steps needed for a simulation episode, given by $n = T/\delta t$, where T is the desired duration of simulation. T is set by the user empirically.

Once all these components are specified by a user, $J(\pi_x)$ can be treated as a black-box function. An invocation on this function will result in a physics simulation for T seconds. During the simulation certain conditions need to be satisfied in order for the motion to be feasible, e.g., a kip-up motion needs to successfully end in a standing position. Feasibility can only be evaluated after a motion is simulated. It is the user's responsibility to define $f(s)$ and agg to implement feasibility as a soft constraints, i.e., $J(\pi_x)$ should return a large penalty value when the motion is infeasible.

5.2 CMA-ES Optimization

The objective function $J(\pi_x)$ does not have a well-defined gradient or Hessian because of its unrestricted nature. Therefore, we choose a derivative-free optimization algorithm. Recent studies [8, 26, 36] have shown that Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [3, 20] is an optimization technique that is well suited for solving non-convex and discontinuous problems. CMA-ES is a stochastic and derivative-free algorithm. It falls into the family of evolutionary optimization strategies. In each iteration, new generation of candidate solutions are generated by variation of its ancestors, i.e. solutions from previous iteration, and some solutions are selected according to fitnesses, i.e. objective function values, from the new generation to inform creation of the next generation. This process repeats until a good solution is found or the number of iterations exceeds a limit. In CMA-ES, a Gaussian distribution with mean μ , covariance matrix Σ and step

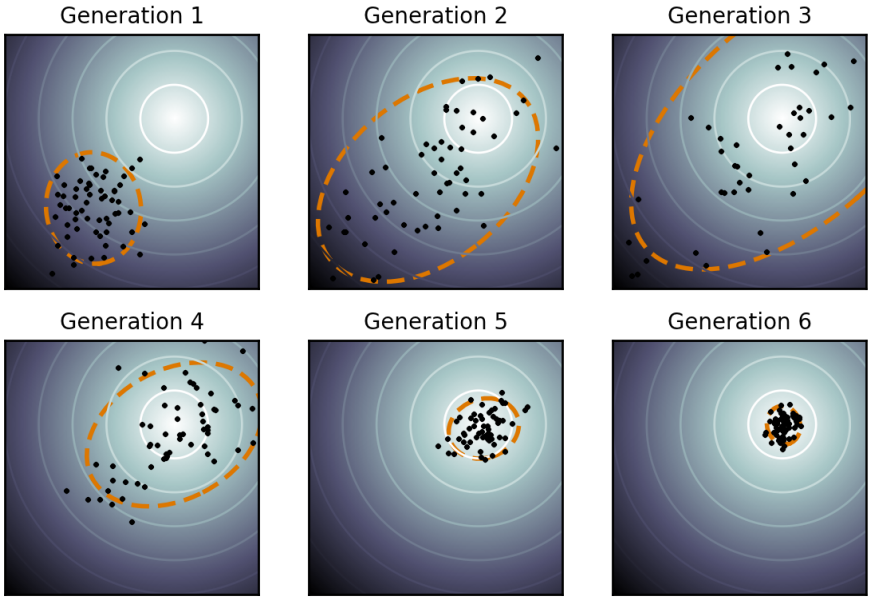


Figure 5.1: An illustration of CMA-ES running on a simple 2D problem. The value of the function is depicted by the white contours with brighter regions representing a better value. The black dots are individual solutions in a generation and orange dashed boundaries represent the Gaussian distribution $\mathcal{N}(\mu, \sigma^2\Sigma)$. Adapted from Wikipedia [1].

size σ is used to sample every new generation, where the covariance matrix decides the shape of the distribution and the step size decides its magnitude. Upon the sampling of a new generation, the fitness of each candidate is used to update the mean, the covariance matrix and the step size. An intuition is that when the fitness increases along one particular direction, the step size becomes larger and the shape of covariance matrix gets thinner in the direction orthogonal to the increasing direction, thus accelerating the convergence speed.

5.3 Results

For the CMA-ES algorithm, we use the freely Java implementation [2]. The optimization is performed offline. The most expensive operation in the algorithm is to evaluate the fitness of each sampled solution. Fortunately, performance can be

boosted if the evaluation of sampled solutions are done concurrently. After sampling for each generation, the fitness evaluations for each individual candidate can be made independently from each other. This makes it possible to modify the CMA-ES algorithm to do parallel evaluation of fitness for all the sampled solutions within one generation. As discussed later, we use a multithreaded multicore implementation.

5.3.1 Kip-up Motion

We optimize the kip-up motion for a more athletic style, namely, a kip-up motion in which the character thrusts higher and stays longer in air but ends up in a standing position more quickly. The motion script that we used is taken from the kip-up motion from previous chapter, and we modified the script to add parameterization by adding a "@" annotation before every numerical field to make them free parameters for optimization. This makes for a vector of 50 free parameters.

The goal of the optimization is to minimize an objective $J(\pi_x)$ that is defined as the weighted sum of five terms:

$$J(\pi_x) = \sum_i w_i J_i(\pi_x) \tag{5.2}$$

where $i \in \{time, height, airborne, energy, skid, final\_state\}$ is the index of these different terms. We will describe each term in further details.

The *time* objective captures the total time used to reach a stand position. Its evaluation function $f_{time}(s)$ is given by:

$$f_{time}(s) = \delta t \cdot H(y_{head} - 1.8)$$

where δt is the simulation time step that can be accessed by the $f(s)$ functions, $H(x)$ is a step function, y_{head} is the current height of character's head, and 1.8 meters is the approximate height of the head when the character is standing upright. The agg_{time} function is a summation of the values $\{f_{time}(s_i)\}$. Thus $J_{time}(\pi_x)$ returns the total duration for which the character's head is lower than 1.8 meters.

The *height* term describes the maximum height reached by the character's pelvis during the airborne phase. While we want to maximize the actual height

value, the optimization performs a minimization by default. Therefore, we negate the height value in this objective term, as defined in f_{height}:

$$f_{height} = \begin{cases} -y_{pelvis} & \text{if airborne} \\ 0 & \text{otherwise} \end{cases}$$

where y_{pelvis} is the height of the pelvis. The agg_{height} function is defined as the min operation, indicating J_{height} returns the height that has the largest absolute value.

The *airborne* term captures the duration of the airborne phase. Similarly to the height term, we negate the actual time. Its evaluation function $f_{airborne}(s)$ is given by:

$$f_{airborne} = \begin{cases} -\delta t & \text{if airborne} \\ 0 & \text{otherwise} \end{cases}$$

The *energy* term is used to calculate the effort spent during the motion. The effort is approximated as the sum of τ_i^2. We want to minimize the effort defined by $f_{energy}(s)$ as the follows:

$$f_{energy} = \delta t \sum_{i}^{\text{joints}} \tau_i^2$$

where τ_i is the torque of each joint.

The *skid* term is to penalize the foot sliding behaviour. f_{skid} is defined as:

$$f_{skid} = \begin{cases} \delta t \sum_i^{\text{feet}} |\theta_i - \pi|^2 \cdot H(|\theta_i - \pi| - \frac{1}{9}\pi) & \text{if foot in contact} \\ 0 & \text{otherwise} \end{cases}$$

where $H(x)$ is the step function, θ_i is the orientation of foot in radian, and π is the foot orientation when it is well planted on ground. It penalizes foot sliding by introducing a quadratic penalty on a foot that tilts for larger than 20 degrees ($\frac{1}{9}\pi$ rad) when it is in contact with the ground. The agg_i function for airborne, energy and skid terms are all defined as sum operators.

Lastly, the *final_state* term defines the constraint forcing the character to be in a standing position at the end of motion. It simply returns a very large positive value if the head position is lower than 1.8 meters at the last simulation step, meaning

the character should not fall onto the ground.

We set the weight for each term as the follows: $w_{time} = 1$, $w_{height} = 10$, $w_{airborne} = 10$, $w_{energy} = \frac{1}{50000}$, $w_{skid} = 10$ and $w_{final\_state} = 1$. This scales each objective term to approximately the same magnitude. The height term and the airborne term have a bit larger weight in an attempt to produce motions with a more easily identified airborne style. The simulation is set to run for 5 seconds before returning the objective value. The motion is simulated using JBox2D [4]. The initial state s_0 is chosen to have the character lying on the ground. δt is set to 0.25ms. The CMA-ES algorithm samples 19 candidates per generation and optimization runs on a cluster with 20 cores. Each simulation ($J(\pi_x)$) finishes in about 3 seconds, and one generation of optimization takes about 4 seconds. It takes around an hour to generate 1000 iterations. After 1000 iterations the objective function decreased in value from -0.336 to -13.921. The rise up time is reduced from $2.962s$ to $2.577s$, the maximum height increases from $0.586m$ to $1.253m$, and the airborne time increases from $0.170s$ to $0.846s$, yielding a 17% decrease in kip-up time, 114% and 398% of increase in maximum height and airborne time respectively. The energy term remains the almost the same: $210142.719(Nm)^2$ v.s. $215812.203(Nm)^2$ before and after optimization. However, it is clear that with the same energy, the optimized motion shown in Figure 5.2 is noticeably more athletic than the original motion as seen in Figure 4.4 and Figure 4.5.

5.3.2 Prone-to-Stand Motion

We also optimize the prone-to-stand motion for a quicker and more efficient motion. The objective function takes the same form as in Equation 5.2. We use a slightly different selection of terms, where $i \in \{time, energy, contactless, skid, final\_state\}$. *time*, *energy* and *final_state* are same as in the previous section. We made a change to the *skid* term and added a *contactless* term. For the skid term, we modified $f_{skid}(s)$ as follows:

$$f_{skid} = \begin{cases} \delta t \sum_i^{\text{feet}} |\theta_i - \pi|^2 \cdot H(|\theta_i - \pi| - \frac{1}{9}\pi) & \text{if foot in contact and } y_{head} > 1.5m \\ 0 & \text{otherwise} \end{cases}$$

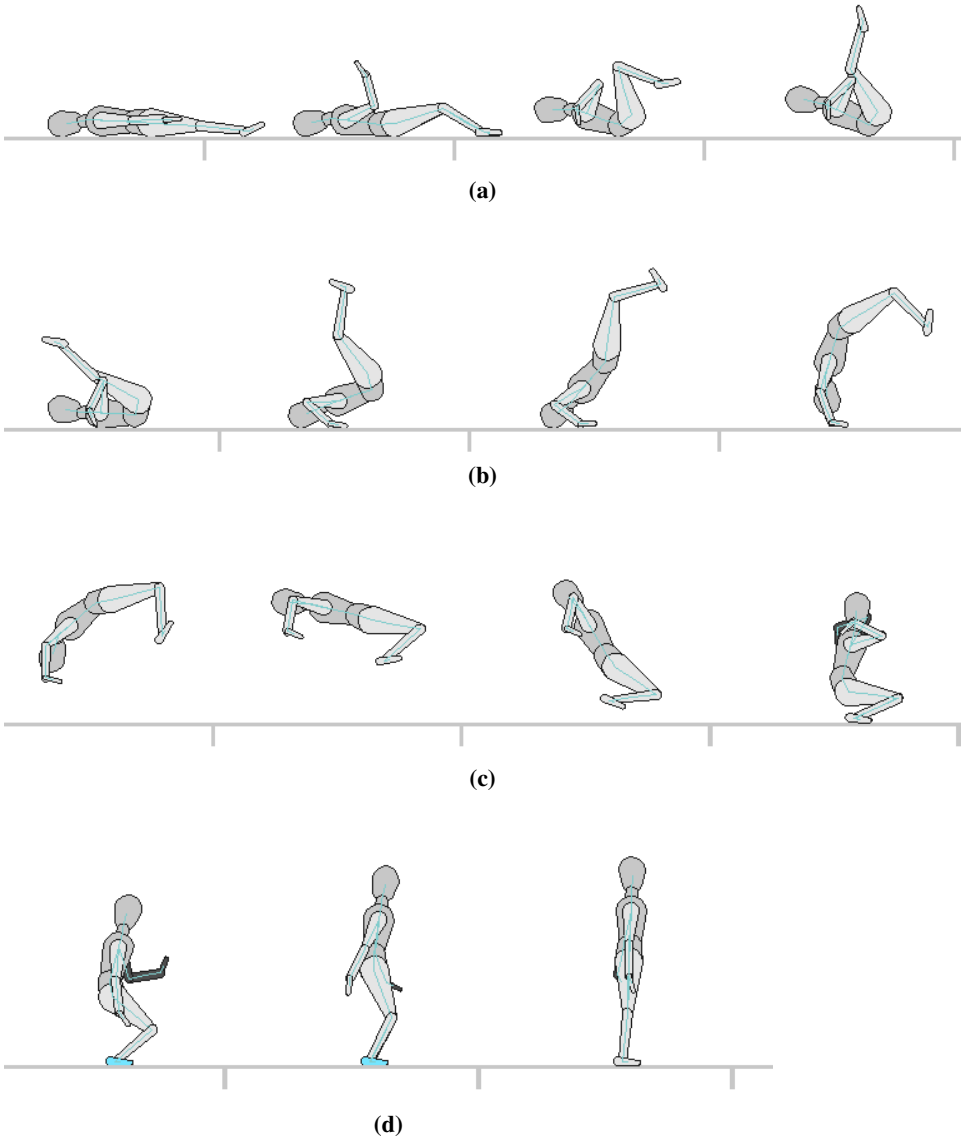
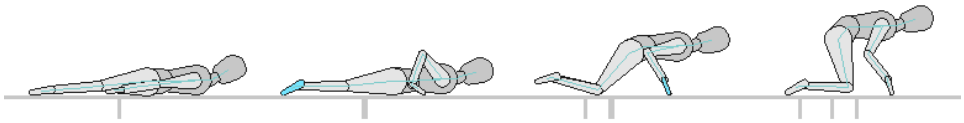


Figure 5.2: More athletic kip-up motion achieved after 1000 iterations of optimization

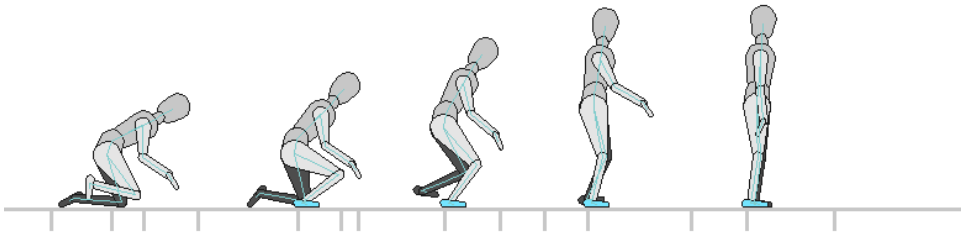
The only change made to the f_{skid} term is that it now only evaluates the skid term when the character's head rises above 1.5m. This ensures that the feet orientations in the early phases are not accounted into the objective. The contactless term penalizes the time for which the character has no contact with the ground. It is a negation of $f_{airborne}$:

$$f_{contactless} = \begin{cases} \delta t & \text{if character has no contact with ground} \\ 0 & \text{otherwise} \end{cases}$$

The weight of each term is set to: $w_{time} = 1$, $w_{energy} = 1/50000$, $w_{contactless} = 10000$, $w_{skid} = 100$. Each simulation runs for 6 seconds in JBox2D. The character has an initial prone pose on the ground. δt is set to 0.25ms. The motion controller has 45 free parameters. The CMA-ES algorithms samples 16 candidates per generation. We use 17 cores to run the optimization. It takes around 40 minutes to finish 1000 iterations of optimization. The objective function value of the optimized motion decreases from 498.5 to 6.1. Consequently, the optimized motion looks more efficient and more natural than the input motion. Noticeably, the energy term has decreased from $206414.4(Nm)^2$ to $83262.1(Nm)^2$, and the rising time has decreased from $4.4s$ to $1.9s$. The values of both terms have decreased by around 57%, resulting in a much faster and more efficient motion as seen in Figure 5.3.



(a)



(b)

Figure 5.3: More efficient prone-to-stand motion achieved after 1000 iterations of optimization

Chapter 6

State-Action Compatibility Model

For many motions, there are often a multitude of actions that can be used to perform a given motion task, or a multitude of situations in which we wish to perform the motion. In this chapter we introduce a framework for modeling the space of all possible ways that a motion can be achieved within a predefined space of actions. We call this a *space-action compatibility* model because it consists of learning an enclosed hypervolume in the state-action space. This hypervolume will be modelled using a Support Vector Machine (SVM). Any point inside the hypervolume means that the state (the "situation") and the action are compatible, i.e., that the given action applied in the given situation will lead to the successful completion of the task. The model thus attempts to capture all possible valid actions for a given situation, and conversely all possible situations (or states) in which a given action leads to successful motions. As an illustrative example, consider the task of rising up from a chair. In this task, the state can be described as the dimensions of the chair, and action can be described as the controller that attempts to perform the rise-up motion. The state-action compatibility model then maintains a hypervolume that contains compatible points, each point meaning the chair dimension is compatible with controller, i.e., the controller can perform a successful sit-to-stand motion from that chair.

Given a d_s-dimensional state or situation instance s, $s \in \mathbb{R}^{d_s}$, and a d_a-dimensional action parameterization a, $a \in \mathbb{R}^{d_a}$, a state-action point is denoted as: (s,a) and

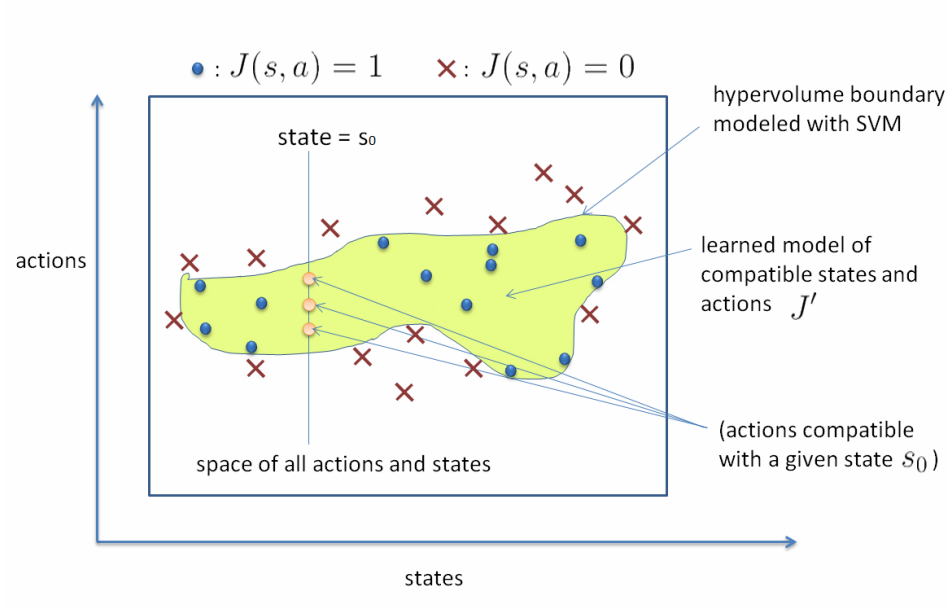


Figure 6.1: Abstract view of state-action compatibility model

compatibility function is then defined as:

$$J((s,a)) = \begin{cases} 1 & \text{if } s \text{ and } a \text{ are compatible} \\ 0 & \text{otherwise} \end{cases}$$

which is evaluated through the process of physics simulation.

We propose to learn a corresponding state-action compatibility model:

$$J'((s,a)) \approx J((s,a))$$

where $J'((s,a))$ is an approximate model of $J((s,a))$ but can be evaluated in significantly less time.

The concept of state-action compatibility model is illustrated in Figure 6.1. The process to acquire J' is defined by two steps:

1. *Sampling*: given an initial point of compatible state-action (s_0, a_0), where $J((s_0, a_0)) = 1$, we explore the compatibility and incompatibility regions

within the state-action space $\mathbb{R}^{(d_s+d_a)}$ by sampling points starting from (s_0, a_0) and gradually moving towards more distant regions.

2. *Learning*: a classification model $J'((s,a))$ is learned from the sampled points that can be used to predict compatibility.

Once the classification model $J'((s,a))$ is acquired, it can be used to perform different operations. For example, it can be used directly, i.e., given (s,a), it determines if the they are compatible. Alternatively, it can be used indirectly as part of a *query engine* to get compatible actions or states: given s, it returns $\{a_i\}$, i.e., a set of actions that are compatible with s; or given a, return $\{s_i\}$, i.e., a set of situations compatible with a.

The query engine takes as input the desired state s or the desired action a, and formulates a diverse optimization problem [7] that finds a diverse set of points that are compatible with the input, i.e., a set of actions $\{a_i\}$ for state input s or a set of states $\{s_i\}$ for action input a.

In the remainder of this chapter, we will go through each step and demonstrate the results of state-action compatibility for the sit-to-stand motion.

6.1 Sampling

In order to build the state-action compatibility model, we need to generate state-action data and use this data to learn a classification model. Each of these data points is a trial of a specific action for a specific state. It is performed by a physics-based simulation and evaluated by the function $J((s,a))$ to indicate whether state-action point is compatible. The sampled data points should ideally cover as much of the compatible regions as possible. The sampling algorithm should also clearly model the boundary between compatible and incompatible regions by sampling points close to the actual boundary.

We develop an iterative algorithm for probabilistically sampling points along different paths. An abstract view of this method is illustrated in Figure 6.2. Each sampling path maintains a multivariate Gaussian distribution, $\mathcal{N}(\mu, \Sigma)$, i.e. a covariance matrix, Σ, and a location point μ, from which all the samples are drawn. The Gaussian distributions are updated at each iteration using the sampled points.

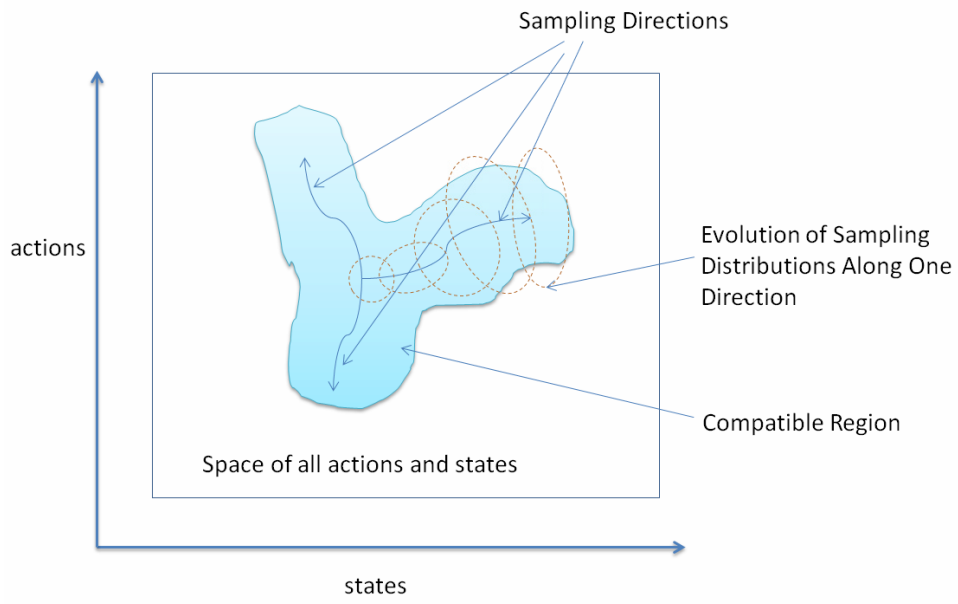


Figure 6.2: Abstract view of sampling algorithm. There are three sampling paths. The evolution of Gaussian distribution along one sampling path is illustrated.

The sampling path is loosely defined as the iterative evolution of location points, $\{\mu :\}$. Over repeated iterations, we update the Gaussian distribution so that the location point μ along each sampling path remains in the compatible region, but it is pushed further away from μ in the other sampling paths. This helps sampling paths go to different directions and thus makes sure that the algorithm explores different regions that are distant from each other. The covariance matrix Σ is updated to adjust its shape along the dimensions orthogonal to the direction of the sampling path. Intuitively, these orthogonal dimensions in the covariance matrix are shaped wide enough so that points at a certain distance from μ, i.e. incompatible points, can be sampled with a non-trivial probability, but not so wide that points close to μ, i.e. compatible points, have too little probability to be sampled.

We use a round-robin CMA algorithm for the purpose of maintaining Gaussian distributions and sampling points. For a state-action problem with $n-1$ dimensions, we use n sampling paths. Each sampling path maintains a CMA algorithm.

The n sampling paths all start with the same initial state-action point. For each iteration, CMA of each sampling path samples a new generation of points and updates the internal Gaussian distribution using these points. This process is repeated until it reaches a specified number of iterations. All the points sampled using CMA are kept as training data.

Each path's CMA optimization attempts to maximize an objective function which we will define shortly. The objective function is designed with two purposes in mind: *depth*, i.e., a path's location point μ should reach to distant regions, and *breadth*, i.e., the covariance matrix Σ should adapt to the orthogonal dimensions of each path's direction. Let $p = (s,a)$ be a state-action point. Every sampled point along the sampling path m is initially assigned an objective value of

$$d_i = \sum_{j \neq m}^{n} \left| p_i - p_{best_j} \right| + K \min_{j \neq m} \left| p_i - p_{best_j} \right| - C \cdot (1 - J(p_i)) \tag{6.1}$$

where $\{p_i\}$ are the sampled points within the iteration, p_{best_j} is the point with the best objective in another sampling path j, and $J(p) = J((s,a))$ returns 1 if the state-action point is compatible and 0 otherwise. The first two terms on the right hand side of Equation 6.1 represents the sum of distances and the minimum distance from the current sampled point to the best points in other paths. We set $K = 5$ to reward a large minimum distance. C is set to a very large positive number to penalize an incompatible point. We can then find the best point p_k from this generation:

$$p_{best} = p_k \,,\, k = \operatorname*{argmax}_i d_i$$

This can be seen as finding the point with the best depth objective. The objective value of the non-best points are further updated with both depth and breadth objectives. We approximate the direction of the current path using the line:

$$(p_{last}, p_{best})$$

where p_{last} is the point in the last generation that had the best objective or simply the initial point for the first generation. For every sampled point p_i, its orthogonal projection onto the line (p_{last}, p_{best}) is denoted by p_{proj_i}. The two quantities

$\left|p_i - p_{proj_i}\right|$ and $\left|p_{proj_i} - p_{best_i}\right|$ are then related to the breadth and depth respectively. We then use the updated objective value j_i, written as:

$$j_i = \begin{cases} d_i & \text{if } i = k \\ \max(0, d_i + K_2(\left|p_i - p_{proj_i}\right|) - \left|p_{proj_i} - p_{best_i}\right|^2) & \text{if } i \neq k, J(p_i) > 0 \\ -\left|p_i - p_{best_i}\right| & \text{if } J(p_i) = 0 \end{cases} \tag{6.2}$$

where K_2 is the parameter to reward point in large breadth, and we choose 2 as its value. The underlying CMA algorithms then iteratively updates its Gaussian distribution by trying to maximize this objective value.

6.2 Learning

From the sampling stage, we get a set of training data of the form

$$\mathscr{D} = \left\{ ((\mathbf{s}, \mathbf{a})_i, y_i) \mid (\mathbf{s}, \mathbf{a})_i \in \mathbb{R}^{d_s + d_a}, y_i \in \{0, 1\} \right\}_{i=1}^{n}$$

We use a support vector machine (SVM) with a Gaussian radial basis kernel to train a classification model that predicts the compatibility of any point in the state-action space. An SVM finds a set of support vectors among the input data set that divides the entire space into two regions by maximizing the smallest distance of a training point to its predicted boundary.

6.3 Querying

Once the classification model $J'(s,a)$ is learned, we can use it to query for actions. The query problem is defined as follows: given a state s_0, and the number of points n, we want to return $\{a\}_n$, i.e., n different actions that are likely to be compatible with the given state.

The algorithm starts by randomly selecting a support vector (s_{init}, a_{init}) from the learned SVM that is known to lie in a compatible region and s_{init} is close to s_0: $|s_{init} - s_0| < \delta$. We use s_{init} as the seed point and optimize (s,a) to minimize the following objective:

$$|s - s_0| + C((s,a))$$

where $C(s,a)$ is a penalty function that evaluates to positive infinity for incompatible points of state-actions.

The optimization problem returns a point (s_0, a_0). From this point, we formulate a diversity optimization problemto find a set of n action parameters that minimize the diversity objective similar to [7]:

$$D = -\sum_{i=0}^{n}(C(s_0,a_i) - \sum_{j=0}^{n}(\left|a_i - a_j\right| + Kd_{\min}(a_i)))$$

where $\{a_i\}$ denotes the set of n action parameters, $C(s,a)$ is the same penalty function described earlier. The parameter K rewards a point a_i that has a large distance to the closest neighbour in set $\{a_i\}$. We use $K = 5$. A round-robin CMA algorithm is used to minimize D. One generation of CMA optimization is applied to each of the n action parameters in turn, before moving on to the next generation of CMA optimization for each of the action parameters.

6.4 Results

We experiment with the state-action model on a sit-to-stand motion for a planar human character. The input motion is a modification of the sit-to-stand motion in the previous motion, with arm movements added in *phase_1*. The motion script is given in Listing 6.1:

Listing 6.1: Sit To Stand Script

```
{
  (lean)
  [waist @30, hips 120 time: 0.3]
  [shoulders @20, elbows 40 time: 0.3]
}
after 0.4
{
  (standup)
  [vf sb(400, 300) on: trunk, by:leftAnkle]
  [symm]

  [vf (0, 400) on: trunk, by:leftAnkle]
```

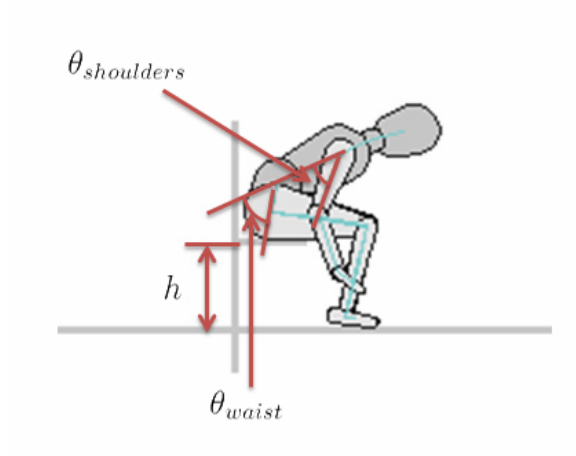


Figure 6.3: State-action parameterization of the sit-to-stand problem. The state parameter is the height of the chair h, the action parameters are the target waist angle θ_{waist}, and the target shoulder angles $\theta_{shoulders}$.

```
13  [symm]
14  [knees 0, hips 0, waist 0, necks 0 time: 1.9]
15  [relax ankles, shoulders, elbows, wrists strength 0.1]
16 }
```

The action parameters are θ_{waist}, i.e., the target waist angle in the "lean" phase, and $\theta_{shoulders}$, i.e., the target shoulder angles in the "lean" phase: $x = (\theta_{waist}, \theta_{shoulders})$. They are parameterized in line 3 and 4 in Listing 6.1, and can be visualized in Figure 6.3. The state parameter is the height of chair $s = (h)$. We thus have a 3-D state-action space in which each point is in the form of $p = (h, \theta_{waist}, \theta_{shoulders})$. We limit the sampling range on each dimension: $0.15 < h < 0.75$, $0 < \theta_{waist} < 90$, and $0 < \theta_{shoulders} < 180$. Every point outside this range is deemed as infeasible.

Sampling starts with the compatible point $(0.45, 30, 20)$, and keeps $n = 4$ paths of CMA. For the purpose of distance measurement, we scaled the state-action points with the vector $(10, 1/40, 1/40)$ in order to compensate for the dimensions that have smaller magnitudes. Each generation of a single CMA direction samples 7 points, each point requires a simulation of 2.5 physics seconds with 0.25ms time step, and it takes around 2 seconds to finish simulation. The sampling is done offline. In 6 hours, 359 iterations are sampled with a total 10052 points, among

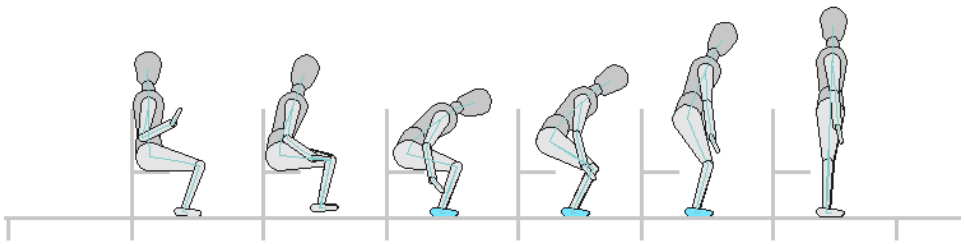


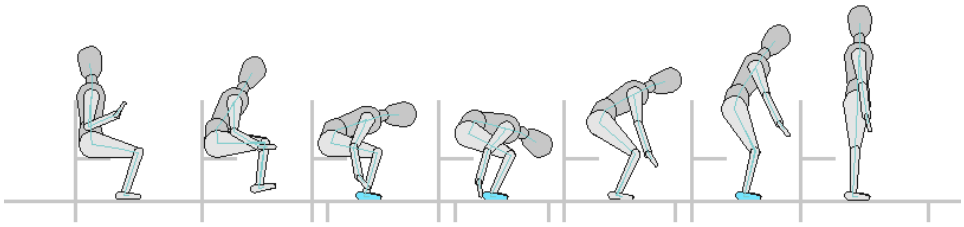
Figure 6.4: Input motion $(s,a) = (0.45, 30, 20)$

which 1323 are incompatible points and 8729 are compatible points.

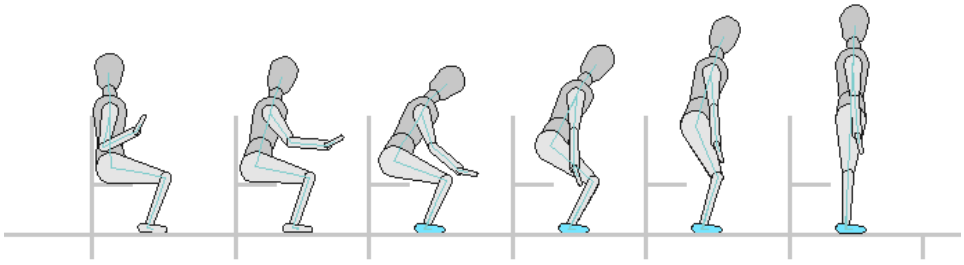
These 10052 points are used as the training data set for the SVM. We use libsvm [10] to train the data sets. The slack parameter c is set to 64 and kernel parameter g is set to 4. We assign compatible points a weight of 1 and incompatible points a weight of 4, because the numbers of compatible and incompatible points are not balanced. The training process takes less than 1 s. The 5-fold cross validation rate is 96.5778%, and in the result model the SVM prediction model consists of 176 support vectors. The SVM predict operation has complexity of $O(N)$ where N is the number of support vectors.

The query engine first runs CMA optimization to get the initial action point a_{init}, and then run the diverse optimization for 50 iterations. The number of iteration is picked empirically so that it is not too small such that resulting motions look very similar but not too big that returned points are very close to the boundary of state-action model. For each query, we set $n = 3$, i.e. querying for 3 different actions. We test the query for three given states $s = (0.45), s = (0.16), s = (0.65)$ respectively.

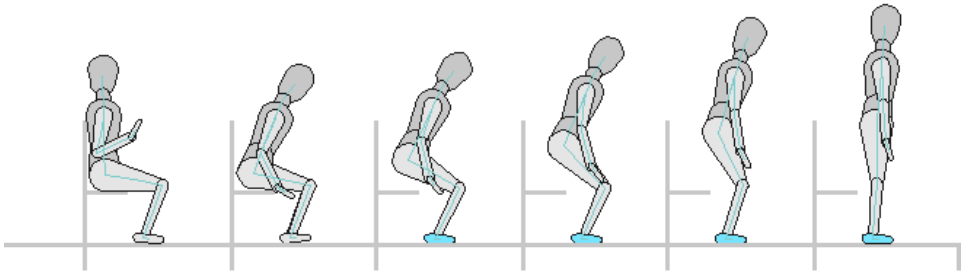
The input motion uses a chair height of $h = 0.45$ and it is shown in Figure 6.4. We use the state action model to query for three motions with the different heights $h = 0.45$, $h = 0.16$ and $h = 0.65$. Running the three queries requires 831ms in average for this problem. The result is shown in Figures 6.5, 6.6 and 6.7. The character is capable of standing from a chair which is significantly lower than the original chair as shown in Figure 6.6. It also shows the ability to jump out of a higher chair and finally stands on ground shown in Figure 6.7. The state-action



(a) motion 1



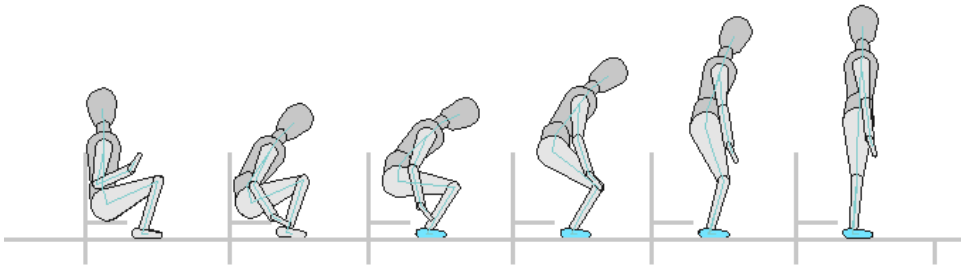
(b) motion 2



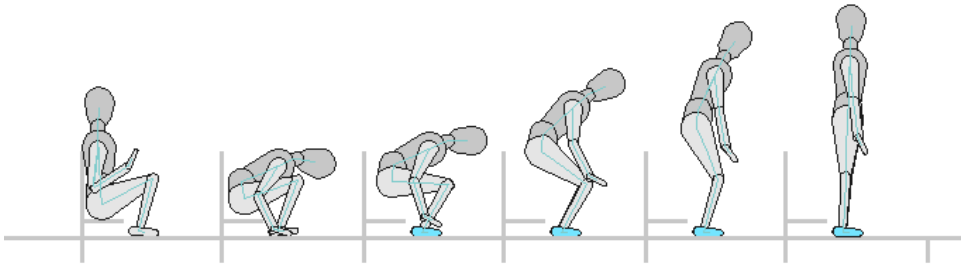
(c) motion 3

Figure 6.5: Three motions are queried from state-action model with the same height as the initial chair height, $h = 0.45$. Motion 1 has a highly flexed waist; motion 2 does not bend the waist; motion 3 uses hands to push the chair.

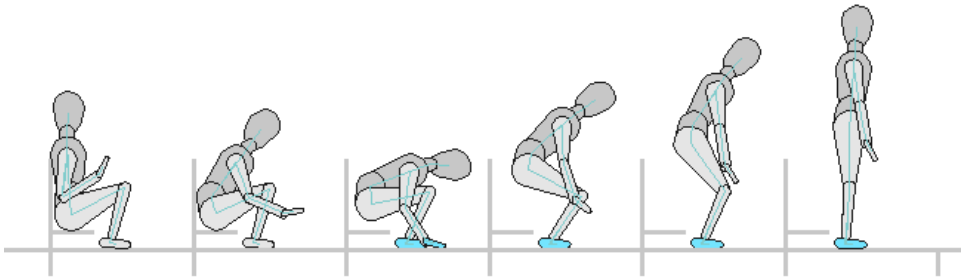
compatibility model does not guarantee all queires can perform actions successfully. This is shown in the motion 1 of Figure 6.7.



(a) motion 1

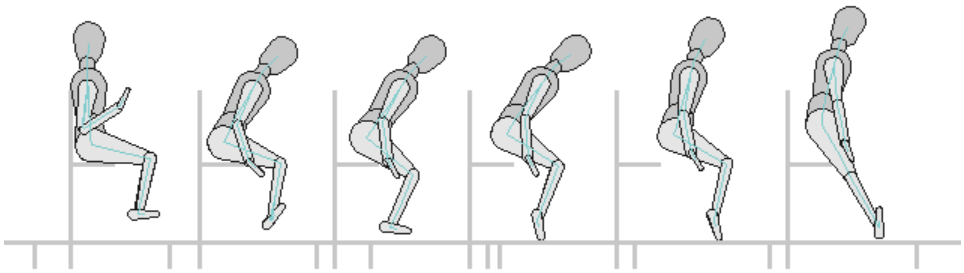


(b) motion 2

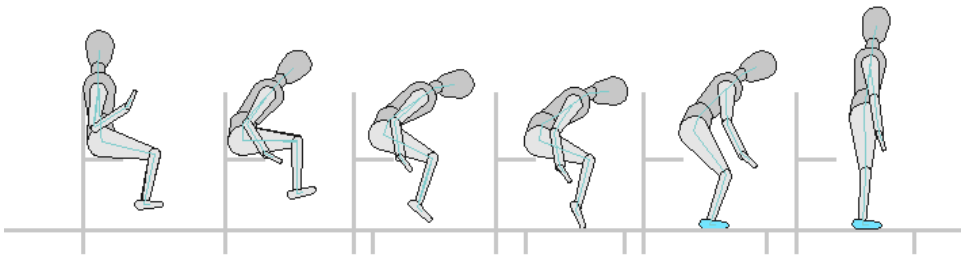


(c) motion 3

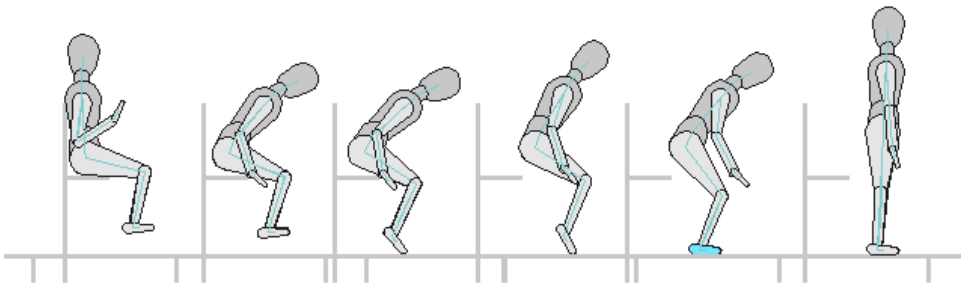
Figure 6.6: Three motions are queried from state-action model with the a different chair height $h = 0.16$. Motion 1 has a small bend on waist; motion 2 has a highly flexed waist and pushes the chair; motion 3 also has a highly flexed waist but pushes the ground.



(a) motion 1



(b) motion 2



(c) motion 3

Figure 6.7: Three motions are queried from state-action with the height $h = 0.65$. The human character's feet are hanging in the air at the beginning. Motions 2 and 3 look very similar and motion 1 failed.

Chapter 7

Conclusion

We have presented a framework for creating physics-based character animations using a flexible simple controller language (SCL) that has several useful built-in motion primitives. The language has a simple syntax and does not require animator's expertise in programming. It provides animators with a semi-interactive environment to debug the motion, thus potentially help designing and tuning physics motions more efficiently. This also opens a window for sharing of motion controllers and reproducible animation research. We demonstrate several motions produced using the SCL including kip-up, sit-to-stand, walk and prone-to-stand. An optimization framework makes it easier to refine and stylize motions. We show that the optimization tool has automatically produced a more energetic variation of a kip motion that is produced by an hand-engineered motion script.

To synthesis a wide family of motions given varied environments, we explored the state-action compatibility model, which is built offline and enables animators to query online a number of different motions given a changed environment. This adds to the toolset of physics animators the capability of designing one motion and generalizing it to various scenarios. Using the state-action compatibility model, we show various sit-to-stand motion variations that can perform on different chair heights.

7.1 Discussion

When using FSM-based controllers to create physics motions, it is useful to have a set of control primitives. The primitives that we use in this thesis include PD control, virtual force, inverse kinematics and simple quiescent stance feedbacks. In many cases, these primitives can be used qualitatively to create a rich set of motions. However, it is yet unclear how we can directly arrive at a specific robust controller for a given motion. For example, for a walk motion, how can we relate a virtual force's magnitude to the walk speed? Due to this reason, manual tuning, optimization or more complex feedback rules may need to be employed in designing a motion.

While the existing set of control primitives constitute a powerful set of control language, it can also be useful to think at a higher level of abstractions. For example, virtual force in general can be applied to a character to potentially move the CoM of the whole body in a desired direction, but it does not yet work well with a situation where the two stance feet are not colocated. It would be helpful to have a higher level of abstraction that regulates the exact position of CoM yet automatically takes care of other constraints such as balance. Another potential higher level primitive could be one that relocates a stance foot contact position while maintaining the upper body pose.

7.2 Future Work

The motions produced for this thesis are all planar motions. An immediate future work direction is to extend all the current motions to 3D characters. This will require introducing more degrees of freedom for the joint angle space as well as the Cartesian space. The SCL will likely remain almost the same except a most noticeable change is that joints will likely need to be specified by quaternions. The controls strategies will also need to consider lateral balance.

One limitation of this work is that it requires motion authors to tweak control parameters in order to get a proper motion. This can require a significant amount of work for complicated motions. A helpful improvement on the system would be to use online optimization to achieve desired goals of a motion phase. For example, at the end of kip-up motion's phase 4, the character requires a large momentum to

propel itself off the ground. It is rather difficult and time consuming to manually find the right parameters to achieve larger momentum. An online optimization tool can help the author to refine the parameters of this phase, while freeing the author to start designing the next phase. This makes a motion design easier and quicker.

A trade-off needs to be made between the conciseness of the SCL and its versatility. The more built-in primitives the language provides, the more powerful it is, but also the less readable it becomes. It currently provides a single feedback primitive that helps maintain balance while in stance. It is clear that more feedback rules will make it easier to create a greater variety of motions. For example, a built-in feedback rule for foot placement based on an inverse pendulum model can help the user create walking and running locomotion more easily, but this rule is very specific to the semantics of a walk, and does not make too much sense to authors who want to develop rising controllers.

Bibliography

[1] CMA-ES. http://en.wikipedia.org/wiki/CMA-ES. → pages iii, viii, 45

[2] CMA-ES source code. https://www.lri.fr/~hansen/cmaes_inmatlab.html. → pages 45

[3] The CMA evolution strategy. https://www.lri.fr/~hansen/cmaesintro.html. → pages iii, 44

[4] jbox2d. http://code.google.com/p/jbox2d/. → pages 30, 48

[5] Tutorial - kip up / kick up. http://www.youtube.com/watch?v=l2F1MtzFw2c. → pages 33

[6] Valley of the dogbots of war. http://hplusmagazine.com/2010/01/15/valley-dogbots-war/. → pages iii

[7] Shailen Agrawal, Shuo Shen, and Michiel van de Panne. Diverse motion variations for physics-based character animation. In *Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 37–44. ACM, 2013. → pages iii, 9, 54, 58

[8] Mazen Al Borno, Martin de Lasa, and Aaron Hertzmann. Trajectory optimization for full-body movements with complex contacts. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 2012. → pages 8, 44

[9] BBC. 'wildcat' robot put through its paces, 2013. http://www.bbc.co.uk/news/technology-24429063. → pages 2

[10] Chih-chung Chang and Chih-Jen Lin. Libsvm – a library for support vector machines. http://www.csie.ntu.edu.tw/~cjlin/libsvm/. → pages 60

[11] Stelian Coros, Philippe Beaudoin, and Michiel van de Panne. Robust task-based control policies for physics-based characters. In *ACM Transactions on Graphics (TOG)*, volume 28, page 170. ACM, 2009. → pages 9

[12] Stelian Coros, Philippe Beaudoin, and Michiel van de Panne. Generalized biped walking control. *ACM Transactions on Graphics (TOG)*, 29(4):130, 2010. → pages vii, 8, 9, 10

[13] Stelian Coros, Andrej Karpathy, Ben Jones, Lionel Reveret, and Michiel Van De Panne. Locomotion skills for simulated quadrupeds. *ACM Transactions on Graphics (TOG)*, 30(4):59, 2011. → pages 8

[14] Petros Faloutsos, Michiel Van de Panne, and Demetri Terzopoulos. Composable controllers for physics-based character animation. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 251–260. ACM, 2001. → pages 9

[15] Christian Gehring, Stelian Coros, Marco Hutter, Michael Bloesch, Markus A Hoepflinger, and Roland Siegwart. Control of dynamic gaits for a quadrupedal robot. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2013. → pages 8

[16] Thomas Geijtenbeek and Nicolas Pronost. Interactive character animation using simulated physics: A state-of-the-art review. In *Computer Graphics Forum*, volume 31, pages 2492–2515. Wiley Online Library, 2012. → pages 6, 8

[17] Thomas Geijtenbeek, Nicolas Pronost, and Frank van der Stappen. Simple data-driven control for simulated bipeds. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 211–219. Eurographics Association, 2012. → pages 9, 10

[18] Sehoon Ha, Yuting Ye, and C Karen Liu. Falling and landing motion control for character animation. *ACM Transactions on Graphics (TOG)*, 31(6):155, 2012. → pages 9, 10

[19] N. Hansen and A. Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9(2):159–195, 2001. → pages iii

[20] Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–195, 2001. → pages 44

[21] Sumit Jain, Yuting Ye, and C Karen Liu. Optimization-based interactive motion synthesis. *ACM Transactions on Graphics (TOG)*, 28(1):10, 2009. → pages 9

[22] Taesoo Kwon and Jessica Hodgins. Control systems for human running using an inverted pendulum model and a reference motion capture sequence. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 129–138. Eurographics Association, 2010. → pages 8

[23] Joseph Laszlo, Michiel van de Panne, and Eugene Fiume. Limit cycle control and its application to the animation of balancing and walking. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 155–162. ACM, 1996. → pages 9

[24] Joseph Laszlo, Michiel van de Panne, and Eugene Fiume. Interactive control for physically-based animation. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 201–208. ACM Press/Addison-Wesley Publishing Co., 2000. → pages 8

[25] Yoonsang Lee, Sungeun Kim, and Jehee Lee. Data-driven biped control. *ACM Transactions on Graphics (TOG)*, 29(4):129, 2010. → pages 6

[26] Libin Liu, KangKang Yin, Michiel van de Panne, and Baining Guo. Terrain runner: control, parameterization, composition, and planning for highly dynamic motions. *ACM Transactions on Graphics (TOG)*, 31(6):154, 2012. → pages 44

[27] Adriano Macchietto, Victor Zordan, and Christian R Shelton. Momentum control for balance. In *ACM Transactions on Graphics (TOG)*, volume 28, page 80. ACM, 2009. → pages vii, 6, 7

[28] Igor Mordatch, Emanuel Todorov, and Zoran Popović. Discovery of complex behaviors through contact-invariant optimization. *ACM Transactions on Graphics (TOG)*, 31(4):43, 2012. → pages 7

[29] Igor Mordatch, Jack M Wang, Emanuel Todorov, and Vladlen Koltun. Animating human lower limbs using contact-invariant optimization. *ACM Transactions on Graphics (TOG)*, 32(6):203, 2013. → pages vii, 7

[30] Jerry Pratt, Chee-Meng Chew, Ann Torres, Peter Dilworth, and Gill Pratt. Virtual model control: An intuitive approach for bipedal locomotion. *The International Journal of Robotics Research*, 20(2):129–143, 2001. → pages 8

[31] Marc H Raibert and Jessica K Hodgins. Animation of dynamic legged locomotion. In *ACM SIGGRAPH Computer Graphics*, volume 25, pages 349–358. ACM, 1991. → pages 8

[32] Yuval Tassa, Tom Erez, and Emanuel Todorov. Synthesis and stabilization of complex behaviors through online trajectory optimization. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 4906–4913. IEEE, 2012. → pages 8

[33] Michiel Van de Panne, Ryan Kim, and Eugene Fiume. Virtual wind-up toys for animation. In *Graphics Interface*, pages 208–208. Citeseer, 1994. → pages 8

[34] Kiril Vidimče, Szu-Po Wang, Jonathan Ragan-Kelley, and Wojciech Matusik. Openfab: A programmable pipeline for multi-material fabrication. *ACM Transactions on Graphics*, 32, July 2013. → pages 3

[35] Jack M Wang, David J Fleet, and Aaron Hertzmann. Optimizing walking controllers. In *ACM Transactions on Graphics (TOG)*, volume 28, page 168. ACM, 2009. → pages 9

[36] Jack M Wang, Samuel R Hamner, Scott L Delp, and Vladlen Koltun. Optimizing locomotion controllers using biologically-based actuators and objectives. *ACM Transactions on Graphics (TOG)*, 31(4):25, 2012. → pages 44

[37] Wikipedia. Bigdog, 2013. http://en.wikipedia.org/wiki/BigDog. → pages 2

[38] Wikipedia. Renderman, 2013. http://en.wikipedia.org/wiki/RenderMan. → pages 3

[39] Andrew Witkin and Michael Kass. Spacetime constraints. In *ACM Siggraph Computer Graphics*, volume 22, pages 159–168. ACM, 1988. → pages 7

[40] Wayne L Wooten and Jessica K Hodgins. Animation of human diving. In *Computer Graphics Forum*, volume 15, pages 3–13. Wiley Online Library, 1996. → pages vi, 21

[41] KangKang Yin, Kevin Loken, and Michiel van de Panne. Simbicon: Simple biped locomotion control. In *ACM Transactions on Graphics (TOG)*, volume 26, page 105. ACM, 2007. → pages vii, 3, 8, 9, 10

[42] Peng Zhao and Michiel van de Panne. User interfaces for interactive control of physics-based 3d characters. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 87–94. ACM, 2005. → pages 8