Phase-Functioned Neural Networks for Character Control

DANIEL HOLDEN, University of Edinburgh
TAKU KOMURA, University of Edinburgh
JUN SAITO, Method Studios

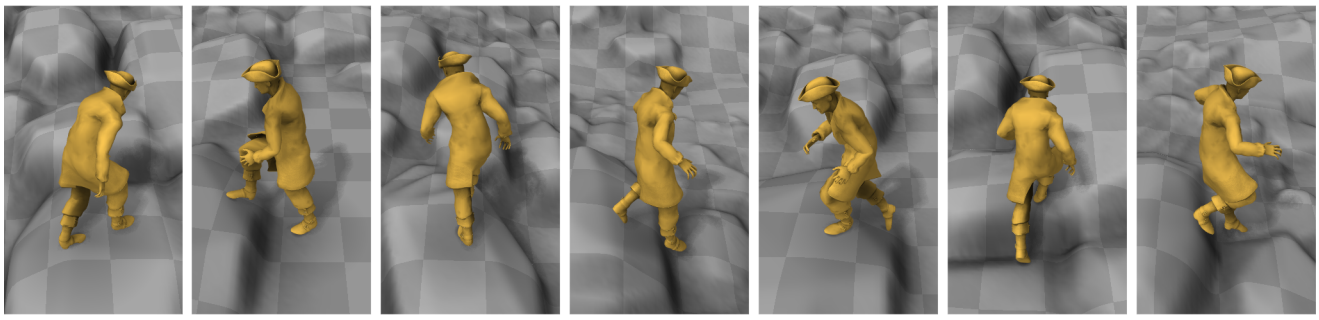

Fig. 1. A selection of results using our method of character control to traverse rough terrain: the character automatically produces appropriate and expressive locomotion according to the real-time user control and the geometry of the environment.

We present a real-time character control mechanism using a novel neural network architecture called a Phase-Functioned Neural Network. In this network structure, the weights are computed via a cyclic function which uses the phase as an input. Along with the phase, our system takes as input user controls, the previous state of the character, the geometry of the scene, and automatically produces high quality motions that achieve the desired user control. The entire network is trained in an end-to-end fashion on a large dataset composed of locomotion such as walking, running, jumping, and climbing movements fitted into virtual environments. Our system can therefore automatically produce motions where the character adapts to different geometric environments such as walking and running over rough terrain, climbing over large rocks, jumping over obstacles, and crouching under low ceilings. Our network architecture produces higher quality results than time-series autoregressive models such as LSTMs as it deals explicitly with the latent variable of motion relating to the phase. Once trained, our system is also extremely fast and compact, requiring only milliseconds of execution time and a few megabytes of memory, even when trained on gigabytes of motion data. Our work is most appropriate for controlling characters in interactive scenes such as computer games and virtual reality systems.

CCS Concepts: • **Computing methodologies → Motion capture**;

Additional Key Words and Phrases: neural networks, locomotion, human motion, character animation, character control, deep learning

ACM Reference format:
Daniel Holden, Taku Komura, and Jun Saito . 2017. Phase-Functioned Neural Networks for Character Control. *ACM Trans. Graph.* 36, 4, Article 42 (July 2017), 13 pages.
DOI: http://dx.doi.org/10.1145/3072959.3073663

This work is supported by Marza Animation Planet and RIKEN..
Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.
0730-0301/2017/7-ART42 $15.00
DOI: http://dx.doi.org/10.1145/3072959.3073663

1 INTRODUCTION

Producing real-time data-driven controllers for virtual characters has been a challenging task even with the large amounts of readily available high quality motion capture data. Partially this is because character controllers have many difficult requirements which must be satisfied for them to be useful - they must be able to learn from large amounts of data, they must not require much manual pre-processing of data, and they must be extremely fast to execute at runtime with low memory requirements. While a lot of progress has been made in this field almost all existing approaches struggle with one or more of these requirements which has slowed their general adoption.

The problem can be even more challenging when the environment is composed of uneven terrain and large obstacles which require the character to perform various stepping, climbing, jumping, or avoidance motions in order to follow the instruction of the user. In this scenario a framework which can learn from a very large amount of high dimensional motion data is required since there are a large combination of different motion trajectories and corresponding geometries which can exist.

Recent developments in deep learning and neural networks have shown some promise in potentially resolving these issues. Neural networks are capable of learning from very large, high dimensional datasets and once trained have a low memory footprint and fast execution time. The question now remains of exactly how neural networks are best applied to motion data in a way that can produce high quality output in real time with minimal data processing.

Previously some success has been achieved using convolutional models such as CNNs [Holden et al. 2016, 2015], autoregressive models such as RBMs [Taylor and Hinton 2009], and RNNs [Fragkiadaki et al. 2015]. CNN models perform a temporally local transformation on each layer, progressively transforming the input signal until the desired output signal is produced. This structure naturally lends itself to an offline, parallel style setup where the whole input is

given at once and the whole output is generated at once. In some situations such as video games this is undesirable as future inputs may be affected by the player's actions. RNNs and other autoregressive models [Fragkiadaki et al. 2015; Taylor and Hinton 2009] are more appropriate for video games and online motion generation as they only require a single frame of future input, yet they tend to fail when generating long sequences of motion as the errors in their prediction are fed back into the input and accumulate. In this way autoregressive models tend to "die out" when frames of different phases are erroneously blended together or "explode" when high frequency noise is fed back into the system [Fragkiadaki et al. 2015]. Such artifacts are difficult to avoid without strong forms of normalization such as blending the output with the nearest known data point in the training data [Lee et al. 2010] - a process which badly affects the scalability of the execution time and memory usage.

We propose a novel neural network structure that we call a Phase-Functioned Neural Network (PFNN). The PFNN works by generating the weights of a regression network at each frame as a function of the phase - a variable representing timing of the motion cycle. Once generated, the network weights are used to perform a regression from the control parameters at that frame to the corresponding pose of the character. The design of the PFNN avoids explicitly mixing data from several phases - instead constructing a regression function which evolves smoothly over time with respect to the phase. Unlike CNN models, this network structure is suitable for online, real-time locomotion generation, and unlike RNN models we find it to be exceptionally stable and capable of generating high quality motion continuously in complex environments with expressive user interaction. The PFNN is fast and compact requiring only milliseconds of execution time and a few megabytes of memory, even when trained on gigabytes of motion capture data. Some of this compactness can additionally be traded for runtime speed via precomputation of the phase function, allowing for a customized trade off between memory and computational resources.

Dynamically changing the network weights as a function of the phase instead of keeping them static as in standard neural networks significantly increases the expressiveness of the regression while retaining the compact structure. This allows it to learn from a large, high dimensional dataset where environmental geometry and human motion data are coupled. Once trained, the system can automatically generate appropriate and expressive locomotion for a character moving over rough terrain and jumping, and avoiding obstacles - both in natural and urban environments (see Fig. 1 and Fig. 9). When preparing the training set we also present a process to fit motion capture data into a large database of artificial heightmaps extracted from video game environments.

In summary, the contribution of the paper is as follows:

- a novel real-time motion synthesis framework that we call the Phase-Functioned Neural Network (PFNN) that can perform character control using a large set of motion data including interactions with the environment, and
- a process to prepare training data for the PFNN by fitting locomotion data to geometry extracted from virtual environments.

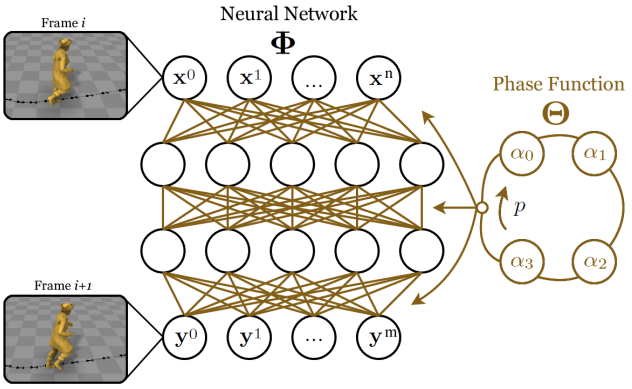

Fig. 2. Visual diagram of Phase Functioned Neural Network. Shown in yellow is the cyclic *Phase Function* - the function which generates the weights of the regression network which performs the control task.

2 RELATED WORK

In this section, we first review data-driven approaches for generating locomotion. Next, we review methods for synthesizing character movements that interact with the environment. Finally, we review methods based on neural networks that focus on mapping latent variables to some parameters of the user's interest.

Data-Driven Locomotion Synthesis. Data-driven locomotion synthesis is a topic that has attracted many researchers in the computer animation and machine learning community. Frameworks based on linear bases, kernel-based techniques, and neural networks have all been successfully applied for such a purpose.

Techniques based on linear bases such as principal component analysis (PCA) are widely adopted for reducing the dimensionality of motion data and also for predicting full body motion from a smaller number of inputs [Howe et al. 1999; Safonova et al. 2004]. As global PCA can have issues representing a wide range of movements in the low dimensional latent space, local PCA is adopted for handling arbitrary types of movements [Chai and Hodgins 2005; Tautges et al. 2011]. Chai and Hodgins [2005] apply local PCA for synthesizing full body motion with sparse marker sets. Tautges et al. [2011] produce similar local structures for predicting full body motion from sparse inertia sensors. Such structures require a significant amount of data preprocessing and computation both for training (i.e., motion segmentation, classification and alignment) and during run-time (i.e., nearest neighbor search).

Kernel-based approaches are proposed to overcome the limitations of linear methods and consider the nonlinearity of the data. Radial Basis Functions (RBF) and Gaussian Processes (GP) are common approaches for blending different types of locomotion [Mukai 2011; Mukai and Kuriyama 2005; Park et al. 2002; Rose et al. 1998]. Gaussian Process Latent Variable Models (GPLVM) are applied for computing a low dimensional latent space of the data to help solve the redundancy of inverse kinematics problems [Grochow et al. 2004] or to improve the efficiency for planning movements [Levine et al. 2012]. Wang et al. [2008] propose a Gaussian Process Dynamic Model (GPDM) that learns the dynamics in the latent space and

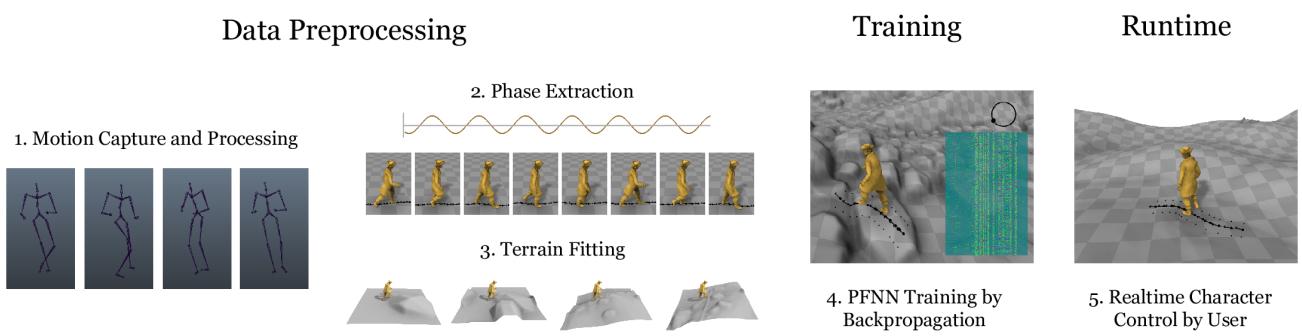

Fig. 3. The three stages of the proposed system: In the data preprocessing stage (left), the captured motion is processed and control parameters are extracted. Next this data is fitted to heightmap data from virtual environments. The PFNN is then trained by back propagation such that the output parameters can be generated from the input parameters (middle). Finally, during runtime, the character motion is computed on-the-fly given the user control and the environmental geometry (right).

projects the motion to the full space using another GP. Kernel-based approaches suffer from the high memory cost of storing and inverting the covariance matrix, which scales in the square and cube order of the number of data points, respectively. Local GP approaches that limit the number of samples for interpolation are proposed to overcome this issue [Rasmussen and Ghahramani 2002], but require $k-$nearest neighbor search which has a large memory usage and a high cost for precomputation and run-time when used with high dimensional data such as human movements.

Data-driven motion synthesis using neural networks is attracting researchers in both the computer animation and machine learning communities thanks to its high scalability and runtime efficiency. Taylor et al. [2009] propose to use the conditional Restricted Boltzmann Machine (cRBM) for predicting the next pose of the body during locomotion. Fragkiadaki et al. [2015] propose an Encoder-Recurrent-Decoder (ERD) network that applies an LSTM model in the latent space for predicting the next pose of the body. These methods can be classified as autoregressive models, where the pose of the character is predicted based on the previous poses of the character during locomotion. Autoregressive models are suitable for real-time applications such as computer games as they update the pose of the character every frame. The cRBM and RNN models are more scalable and runtime-efficient than their classic linear [Xia et al. 2015] or kernel-based counterparts [Wang et al. 2008]. Despite such advantages, they suffer from drifting issues, where the motion gradually comes off the motion manifold due to noise and under-fitting, eventually converging to an average pose. Holden et al. [2016] instead applies a CNN framework along the time domain to map low dimensional user signals to the full body motion. This is an offline framework that requires the full control signal along the time-line to be specified ahead of time for synthesizing the motion. Our framework in this paper is a time-series approach that can predict the pose of the character given the user inputs and the previous state of the character.

Interaction with the Environment. Automatic character controllers in virtual environments that allow the character to avoid obstacles and adapt to terrains are useful for real-time applications such as computer games: these approaches can be classified into methods based on optimization and shape matching.

Methods based on optimization [Lau and Kuffner 2005; Safonova and Hodgins 2007], sampling-based approaches [Coros et al. 2008; Liu et al. 2010], maximum a posteriori (MAP) estimates [Chai and Hodgins 2007; Min and Chai 2012], and reinforcement learning techniques [Lee and Lee 2004; Lee et al. 2010; Levine et al. 2012; Lo and Zwicker 2008; Peng et al. 2016], predict the action of the character given the current state of the character (including the pose) and its relation to the geometry of the environment. They require cost/value functions that evaluate each action under different circumstances. Although it is shown that these methods can generate realistic movements, in some cases the computational cost is exponential with respect to the number of actions and thus not very scalable. More importantly, when using kinematic data as the representation, it is necessary to conduct $k-$nearest neighbour search within the samples [Clavet 2016; Lee et al. 2010] to pull the motion onto the motion manifold. This can be a limiting factor in terms of scalability, especially in high dimensional space. Levine et al. [2012] cope with such an issue by conducting reinforcement learning in the latent space computed by GPLVM but they require classification of the motion into categories and limit the search within each category. Peng et al. [2016] apply deep reinforcement learning in the control space of physically-based animation in a way which can handle high dimensional state spaces. This is a very promising direction of research, but the system is only tested in relatively simple 2D environments. Our objective is to control characters in the full 3D kinematic space with complex geometric environments where previous learning based approaches have not been very successful.

Another set of approaches for controlling characters in a given environment is to conduct geometric analysis of the environments and adapt the pose or motion to the novel geometry. Lee et al. [2006] conduct rigid shape matching to fit contact-rich motions such as sitting on chairs or lying down on different geometries. Grabner et al. [2011] conduct a brute-force search in order to discover locations in the scene geometry where a character can conduct a specific action. Gupta et al. [2011] produce a volume that occupies

the character in various poses and fits this into a virtual Manhattan world constructed from a photograph. Kim et al. [2014] propose to make use of various geometric features of objects to fit character poses to different geometries. Kang et al. [2014] analyze the open volume and the physical comfort of the body to predict where the character can be statically located with each pose. Savva et al. [2016] capture human interactions with objects using the Microsoft Kinect and produce statistical models which can be used for synthesizing novel scenes. These approaches only handle static poses and do not handle highly dynamic interactions. For animation purposes, the smoothness of the motions and the connection of the actions must be considered. Kapadia et al. [2016] estimate the contacts between the body and the environment during dynamic movements and make use of them for fitting human movements into virtual environments. Their method requires an in-depth analysis of the geometry in advance which does not allow real-time interaction in new environments the character may face for the first time. Our technique is based on regression of the geometry to the motion and so can overcome these limitations of previous methods.

Mapping User Parameters to Latent Variables. In many situations people prefer to map scene parameters, such as viewpoints or lighting conditions in images, to latent variables such that users have control over them during synthesis. Kulkarni et al. [2015] propose a technique to map the viewpoint and lighting conditions of face images to the hidden units of a variational autoencoder. Memisavic [2013] proposes a multiplicative network where the latent variables (viewpoint) directly parameterize the weights of the neural network. Such a network is especially effective when the latent parameter has a global effect on the entire output. We find this style of architecture is applicable for locomotion and use the phase as a common parameter between all types of locomotion, thus adopting a similar concept for our system.

3 SYSTEM OVERVIEW

A visual diagram of the PFNN is shown in Fig. 2: it is a neural network structure (denoted by Φ) where the weights are computed by a periodic function of the phase p called the *phase function* (denoted by Θ). The inputs $\mathbf{x}$ to the network include both the previous pose of the character and the user control, while the outputs $\mathbf{y}$ include the change in the phase, the character's current pose, and some additional parameters descibed later.

There are three stages to our system: the preprocessing stage, training stage and the runtime stage. During the **preprocessing stage**, we first prepare the training data and automatically extract the control parameters that will later be supplied by the user (see Section 4). This process includes fitting terrain data to the captured motion data using a separate database of heightmaps (see Section 4.2). During the **training stage**, the PFNN is trained using this data such that it produces the motion of the character in each frame given the control parameters (see Section 5 for the setup of the PFNN and its training) . During the **runtime stage**, the input parameters to the PFNN are collected from the user input as well as the environment, and input into the system to determine the motion of the character (see Section 6).

4 DATA ACQUISITION & PROCESSING

In this section, we first describe about how we do the motion capture and extraction of control parameters which are used for training the system (see Section 4.1). We then describe about how we fit terrain to the motion capture data (see Section 4.2). Finally we summarize the parameterization of the system (see Section 4.3).

4.1 Motion Capture and Control Parameters

Once the motion data is captured, the control parameters, which include the phase of the motion, the semantic labels of the gait, the trajectory of the body, and the height information of the terrain along the trajectory, are computed or manually labeled. These processes are described below.

Motion Capture. We start by capturing several long sequences of locomotion in a variety of gaits and facing directions. We also place obstacles, ramps and platforms in the capture studio and capture further locomotion - walking, jogging and running over obstacles at a variety of speeds and in different ways. We additionally capture other varieties of locomotion such as crouching and jumping at different heights and with different step sizes. Once finished we have around 1 hour of raw motion capture data captured at 60 fps which constitutes around 1.5 GB of data. An articulated body model the same as the BVH version of the CMU motion capture data with 30 rotational joints is used with an additional root transformation added on the ground under the hips.

Phase Labelling. Next the *phase* must be labeled in the data, as this will be an input parameter for the PFNN. This can be performed by a semi-automatic procedure. Firstly, foot contact times are automatically labeled by computing the magnitude of the velocity of the heel and toe joints and observing when these velocities go below some threshold [Lee et al. 2002]. Since this heuristic can occasionally fail, the results are manually checked and corrected by hand. Once these contact times are acquired the phase can be automatically computed by observing the frames at which the right foot comes in contact with the ground and assigning a phase of 0, observing the frames when the left foot comes in contact with the ground and assigning a phase of π, and observing when the next right foot contact happens and assigning a phase of 2π. These are then interpolated for the inbetween frames. For standing motions some minimum cycle duration ($\sim 0.25s$) is used and the phase is allowed to cycle continuously.

Gait Labels. We also provide semantic labels of the gait of the locomotion to the system represented as a binary vector. This is done for two reasons. Firstly they provide a method of removing ambiguity, since a fast walk and slow jog can often have the same trajectory, and secondly there are often times when the game designer or the user wishes to observe specific motions in specific scenarios (walking, jogging, jumping etc). This process is performed manually but can be greatly simplified by ensuring during capture that the actor does not change the gait type frequently and instead long captures are made containing only a single type of gait.

Trajectory and Terrain Height. The root transformation of the character is extracted by projecting center of the hip joints onto the

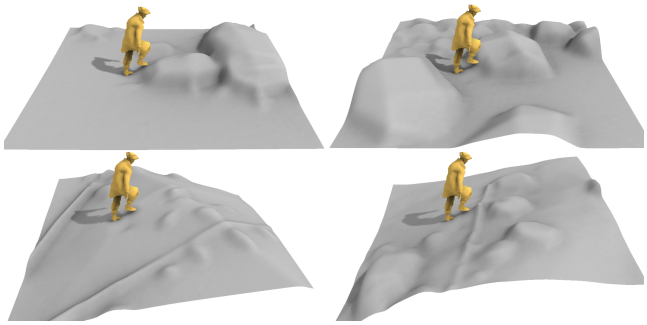

Fig. 4. Results of fitting terrain to motion data. A variety of patches from the separately acquired terrain database are fitted to the same motion.

ground below. The facing direction is computed by averaging the vector between the hip joints and the vector between the shoulder joints, and taking the cross product with the upward direction. This direction is smoothed over time to remove any small, high-frequency movements.

Once the trajectory of this root transformation has been extracted, and terrain has been fitted to the motion (see Section 4.2), the height of the terrain is computed at locations under the trajectory, as well as at locations either side of the trajectory, perpendicular to the facing direction and ~25cm away from the center point (see Fig. 5). More details about this process are described in Section 4.3.

Finally, once this process is completed, we create mirrored versions of all captures to double the amount of data.

4.2 Terrain Fitting

In order to produce a system where the character automatically adapts to the geometry of the environment during runtime, we need to prepare training data which includes the character moving over different terrains. Since simultaneous capture of geometry and motion in a motion capture studio is difficult, we instead present an offline process that fits a database of heightmaps gathered from video games or other virtual environments to separately captured motion data. Once fitted, these terrains allow us to use parameters relating to the geometry of the terrain as input to the control system.

Our database of heightmaps is extracted from several scenes built using the Source Engine. We trace rays into these scenes from above to capture the full geometric information in heightmap form with a resolution of one pixel per inch. From these heightmaps we randomly sample orientations and locations for approximately 20000 patches 3x3 meters in area. These patches are used in the fitting process.

The fitting process takes place in two stages. Firstly, for each locomotion cycle in the motion data we find the 10 best fitting patches in the database by searching in a brute-force way - attempting to fit every patch and selecting the patches which minimize a given error function. Secondly, we use a simple Radial Basis Function (RBF) mesh editing technique to refine the result, editing the terrain such that the feet of the character are exactly on the ground during contact times.

Let us now describe the details of the fitting process. For each motion cycle in the database, given the left/right heel and toe joint indices $J \in \{lh\ rh\ lt\ rt\}$, we first compute for every frame i their heights $f_i^{lh}, f_i^{rh}, f_i^{lt}, f_i^{rt}$, and contact labels $c_i^{lh}, c_i^{rh}, c_i^{lt}, c_i^{rt}$ (a binary variable indicating if the joint is considered in contact with the floor or not). We find the average position of these joints for all the times they are in contact with the floor and center each patch in the database at this location. For each patch, we then compute the heights of the terrain under each of these joints $h_i^{lh}, h_i^{rh}, h_i^{lt}, h_i^{rt}$.

The fitting error E_{fit} is then given as follows:

$$E_{fit} = E_{down} + E_{up} + E_{over} \tag{1}$$

where E_{down} ensures the height of the terrain matches that of the feet when the feet are in contact with the ground,

$$E_{down} = \sum_i \sum_{j \in J} c_i^j (h_i^j - f_i^j)^2, \tag{2}$$

E_{up} ensures the feet are always above the terrain when not in contact with the ground (preventing intersections),

$$E_{up} = \sum_i \sum_{j \in J} (1 - c_i^j) \max(h_i^j - f_i^j, 0)^2, \tag{3}$$

and E_{over}, which is only activated when the character is jumping (indicated by the variable g_i^{jump}), ensures the height of the terrain is no more than l in distance below the feet (in our case l is set to ~30cm). This ensures that large jumps are fitted to terrains with large obstacles, while small jumps are fitted to terrains with small obstacles.

$$E_{over} = \sum_i \sum_{j \in J} g_i^{jump} (1 - c_i^j) \max((f_i^j - l) - h_i^j, 0)^2. \tag{4}$$

Once we have computed the fitting error E_{fit} for every patch in the database we pick the 10 with the smallest fitting error and perform the second stage of the fitting process. In this stage we edit the heightmap such that the feet touch the floor when they are in contact. For this we simply deform the heightmap using a simplified version of the work of Botsch and Kobbelt [2005]. We apply a 2D RBF to the residuals of the terrain fit using a linear kernel. Although we use this method, any other mesh editing technique should also be appropriate as the editing required in almost all cases is quite minor.

Total data processing and fitting time for the whole motion database is around three hours on an Intel i7-6700 3.4GHz CPU running single threaded. Fig. 4 visualizes the results of this fitting process.

4.3 System Input/Output Parameters

In this section, we describe about the input/output parameters of our system. For each frame i our system requires the phase p for computing the network weights. Once these are computed the system requires neural network input $\mathbf{x}_i$ which includes the user control parameters, the state of the character in the previous frame, and parameters of the environment. From this it computes $\mathbf{y}_i$ which includes the state of the character in the current frame, the change in phase, the movement of the root transform, a prediction of the trajectory in the next frame, and contact labels for the feet joints for use in IK post-processing.

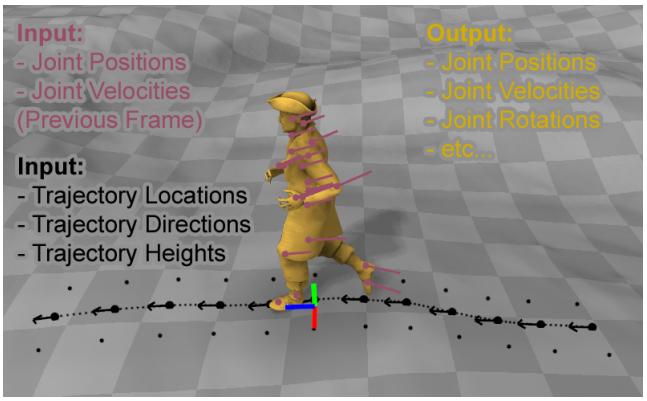

Fig. 5. A visualization of the input parameterization of our system. In pink are the positions and velocities of character's joints from the previous frame. In black are the subsampled trajectory positions, directions and heights. In yellow is the mesh of the character, deformed using the joint positions and rotations output from our system.

Now let us describe the details of the input parameters $\mathbf{x}_i$. Our parameterization is similar to that used in Motion Matching [Clavet 2016] and consists of two parts. For the state of the character we take the positions and velocities of the character's joints local to the character root transform. For the user control we look at a local window centered at frame i and examine every tenth surrounding frame which, in our case, produces $t = 12$ sampled surrounding frames covering 1 second of motion in the past and 0.9 seconds of motion in the future. From each surrounding sampled frame we extract a number of features including the character trajectory position and trajectory direction local to the character root transform at frame i, the character's gait represented as a binary vector, and the heights of the terrain under the trajectory and at two additional points 25cm away to the left and right of the trajectory. See Fig. 5 for a visual demonstration of this parameterization.

The full parameterization of the input control variables for a single frame i consists of a vector $\mathbf{x}_i = \{\ \mathbf{t}_i^p\ \mathbf{t}_i^d\ \mathbf{t}_i^h\ \mathbf{t}_i^g\ \mathbf{j}_{i-1}^p\ \mathbf{j}_{i-1}^v\ \} \in \mathbb{R}^n$ where $\mathbf{t}_i^p \in \mathbb{R}^{2t}$ are the subsampled window of trajectory positions in the 2D horizontal plane relative to frame i, $\mathbf{t}_i^d \in \mathbb{R}^{2t}$ are the trajectory directions in the 2D horizontal plane relative to frame i, $\mathbf{t}_i^h \in \mathbb{R}^{3t}$ are the trajectory heights of the three left/right/center sample points relative to frame i, $\mathbf{t}_i^g \in \mathbb{R}^{5t}$ are the trajectory semantic variables indicating the gait of the character and other information (represented as a 5D binary vector), $\mathbf{j}_{i-1}^p \in \mathbb{R}^{3j}$ are the local joint positions of the previous frame, and $\mathbf{j}_{i-1}^v \in \mathbb{R}^{3j}$ are the local joint velocities of the previous frame, where j is the number of joints (in our case 31). In this work, the individual components of the additional semantic variables $\mathbf{t}_i^g$ are active when the character is in the following situations: 1. standing, 2. walking, 3. jogging, 4. jumping, 5. crouching (also used to represent the ceiling height).

The full parameterization of the output variables for a frame i consists of a vector $\mathbf{y}_i = \{\ \mathbf{t}_{i+1}^p\ \mathbf{t}_{i+1}^d\ \mathbf{j}_i^p\ \mathbf{j}_i^v\ \mathbf{j}_i^a\ \dot{\mathbf{r}}_i^x\ \dot{\mathbf{r}}_i^z\ \dot{\mathbf{r}}_i^a\ \dot{p}_i\ \mathbf{c}_i\ \} \in \mathbb{R}^m$ where $\mathbf{t}_{i+1}^p \in \mathbb{R}^{2t}$ are the predicted trajectory positions in the next frame, $\mathbf{t}_{i+1}^d \in \mathbb{R}^{2t}$ are the predicted trajectory directions in the next frame, $\mathbf{j}_i^p \in \mathbb{R}^{3j}$ are the joint positions local to the character root transform, $\mathbf{j}_i^v \in \mathbb{R}^{3j}$ are the joint velocities local to the character root transform, $\mathbf{j}_i^a \in \mathbb{R}^{3j}$ are the joint angles local to the character root transform expressed using the exponential map [Grassia 1998], $\dot{\mathbf{r}}_i^x \in \mathbb{R}$ is the root transform translational x velocity relative to the forward facing direction, $\dot{\mathbf{r}}_i^z \in \mathbb{R}$ is the root transform translational z velocity relative to the forward facing direction, $\dot{\mathbf{r}}_i^a \in \mathbb{R}$ is the root transform angular velocity around the upward direction, $\dot{p}_i \in \mathbb{R}$ is the change in phase, and $\mathbf{c}_i \in \mathbb{R}^4$ are the foot contact labels (binary variables indicating if each heel and toe joint is in contact with the floor).

5 PHASE-FUNCTIONED NEURAL NETWORK

In this section we discuss the construction and training of the Phase-Functioned Neural Network (PFNN). The PFNN (see Fig. 2) is a neural network with weights that cyclically change according to the *phase* value. We call the function which generates the network weights the *phase function*, which in this work is defined as a cubic Catmull-Rom spline for reasons outlined in Section 5.2. To begin we first describe the chosen neural network structure (see Section 5.1), followed by the phase function used to generate the weights for this network structure (see Section 5.2). Finally, we describe about the training procedure (see Section 5.3).

5.1 Neural Network Structure

Given input parameters $\mathbf{x} \in \mathbb{R}^n$, output parameters $\mathbf{y} \in \mathbb{R}^m$, and a single phase parameter $p \in \mathbb{R}$ described in Section 4.3 we start by building a simple three layer neural network Φ as follows:

$$\Phi(\mathbf{x}; \boldsymbol{\alpha}) = \mathbf{W}_2\ \mathrm{ELU}(\ \mathbf{W}_1\ \mathrm{ELU}(\ \mathbf{W}_0\ \mathbf{x} + \mathbf{b}_0) + \mathbf{b}_1) + \mathbf{b}_2, \tag{5}$$

where the parameters of the network $\boldsymbol{\alpha}$ are defined by $\boldsymbol{\alpha} = \{\mathbf{W_0} \in \mathbb{R}^{h \times n}, \mathbf{W_1} \in \mathbb{R}^{h \times h}, \mathbf{W_2} \in \mathbb{R}^{m \times h}, \mathbf{b_0} \in \mathbb{R}^h, \mathbf{b_1} \in \mathbb{R}^h, \mathbf{b_2} \in \mathbb{R}^m\}$. Here h is the number of hidden units used on each layer which in our work is set to 512 and the activation function used is the exponential rectified linear function [Clevert et al. 2015] defined by

$$\mathrm{ELU}(x) = \max(x, 0) + \exp(\min(x, 0)) - 1. \tag{6}$$

5.2 Phase Function

In the PFNN, the network weights $\boldsymbol{\alpha}$ are computed each frame by a separate function called the *phase function*, which takes as input the phase p and parameters $\boldsymbol{\beta}$ as follows: $\boldsymbol{\alpha} = \Theta(p; \boldsymbol{\beta})$. Theoretically, there are many potential choices for Θ. For example Θ could be a another neural network, or a Gaussian Process, but in this work we choose Θ to be a cubic Catmull-Rom spline.

Using a cubic Catmull-Rom spline is good for several reasons - it is easily made cyclic by letting the start and end control points be the same, the number of parameters is proportional to the number of control points, and it varies smoothly with respect to the input parameter p. Choosing a cubic Catmull-Rom spline to represent the phase function means each control point $\boldsymbol{\alpha}_k$ represents a certain configuration of weights for the neural network $\boldsymbol{\alpha}$, and the function Θ performs a smooth interpolation between these neural network weight configurations. Another way to imagine Θ is as a one-dimensional cyclic manifold in the (high-dimensional) weight space of the neural network. This manifold is then parameterized

by the phase, and in this sense training the network is finding an appropriate cyclic manifold in the space of neural network weights which performs the regression from input parameters to output parameters successfully.

We found a cyclic spline of only four control points was enough to express the regression required by this system. Given four control points (consisting of neural network weight configurations) $\boldsymbol{\beta} = \{\boldsymbol{\alpha}_0\ \boldsymbol{\alpha}_1\ \boldsymbol{\alpha}_2\ \boldsymbol{\alpha}_3\}$, the cubic Catmull-Rom spline function Θ which produces network weights for arbitrary phase p can be defined as follows:

$$\begin{aligned}\Theta(p;\boldsymbol{\beta}) = \quad & \boldsymbol{\alpha}_{k_1} \\ & + w\,(\tfrac{1}{2}\boldsymbol{\alpha}_{k_2} - \tfrac{1}{2}\boldsymbol{\alpha}_{k_0}) \\ & + w^2(\ \boldsymbol{\alpha}_{k_0} - \tfrac{5}{2}\boldsymbol{\alpha}_{k_1} + 2\boldsymbol{\alpha}_{k_2} - \tfrac{1}{2}\boldsymbol{\alpha}_{k_3}) \\ & + w^3(\tfrac{3}{2}\boldsymbol{\alpha}_{k_1} - \tfrac{3}{2}\boldsymbol{\alpha}_{k_2} + \tfrac{1}{2}\boldsymbol{\alpha}_{k_3} - \tfrac{1}{2}\boldsymbol{\alpha}_{k_0}) \\ w = & \frac{4p}{2\pi} \pmod 1 \\ k_n = & \left\lfloor \frac{4p}{2\pi} \right\rfloor + n - 1 \pmod 4.\end{aligned} \tag{7}$$

5.3 Training

For each frame i the variables $\mathbf{x}_i$, $\mathbf{y}_i$, and the phases p_i are stacked into matrices $\mathbf{X} = [\ \mathbf{x}_0\ \mathbf{x}_1\ \dots\ ]$, $\mathbf{Y} = [\ \mathbf{y}_0\ \mathbf{y}_1\ \dots\ ]$ and $\mathbf{P} = [\ p_0\ p_1\ \dots\ ]$. The mean values $\mathbf{x}_\mu\ \mathbf{y}_\mu$ and standard deviations $\mathbf{x}_\sigma\ \mathbf{y}_\sigma$ are used to normalize the data which is then additionally scaled by weights $\mathbf{x}_w\ \mathbf{y}_w$ which give the relative importances of each dimension: in our experiments the best results are achieved by scaling all input variables relating to the joints by a value of 0.1 to shrink their importance. This increases the influence of the trajectory in the regression, resulting in a more responsive character. The binary variables included in the input are normalized as usual and don't recieve any special treatment. After the terrain fitting is complete and the final parameterization is constructed for each of the 10 different associated patches, we have a final dataset that contains around 4 million data points.

To train the network we must ensure that for a given set of control parameters $\mathbf{X}$ and phase parameters $\mathbf{P}$, we can produce the corresponding output variables $\mathbf{Y}$ as a function of the neural network Φ. Training is therefore an optimization problem with respect to the phase function parameters $\boldsymbol{\beta} = \{\boldsymbol{\alpha}_0\ \boldsymbol{\alpha}_1\ \boldsymbol{\alpha}_2\ \boldsymbol{\alpha}_3\}$ and the following cost function:

$$\mathrm{Cost}(\mathbf{X},\mathbf{Y},\mathbf{P};\boldsymbol{\beta}) = \|\mathbf{Y} - \Phi(\mathbf{X};\Theta(\mathbf{P};\boldsymbol{\beta}))\|_2^2 + \gamma\ |\boldsymbol{\beta}|_1. \tag{8}$$

In this equation the first term represents the mean squared error of the regression result, while the second term represents a small regularization which ensures the weights do not get too large. It also introduces a small amount of sparsity to the weights. This term is controlled by the constant γ, which in this work is set to 0.01.

We use the stochastic gradient descent algorithm Adam [Kingma and Ba 2014] with a model implemented in Theano [Bergstra et al. 2010] which automatically calculates the derivatives of the cost function with respect to $\boldsymbol{\beta}$. Dropout [Srivastava et al. 2014] is applied with a retention probability of 0.7 and the model is trained in mini-batches of size 32. Full training is performed for 20 epochs which takes around 30 hours on a NVIDIA GeForce GTX 660 GPU.

6 RUNTIME

During runtime the PFNN must be supplied at each frame with the phase p and neural network input $\mathbf{x}$. The phase p can be stored and incremented over time using the computed change in phase $\dot{p}$, modulated to loop in the range $0 \le p \le 2\pi$. For the neural network input $\mathbf{x}$, the variables relating to the joint positions and velocities are used in an autoregressive manner, using the computed result from the previous frame as input to the next. Our system also uses past/future trajectories $\mathbf{t}^p\ \mathbf{t}^d$ as input $\mathbf{x}$ (see Section 4.3): the elements related to the *past* are simply recorded, while some care is required for those of the *future*, which we discuss next.

Inputs Relating to Future Trajectories. When preparing the runtime input $\mathbf{x}$ for PFNN, the future elements of the trajectory $\mathbf{t}^p\ \mathbf{t}^d$ are computed by blending the trajectory estimated from the game-pad control stick and those generated by the PFNN in the previous frame.

In Motion Matching, Clavet [2016] uses the position of the stick to describe the desired velocity and facing direction of the character. In our method this desired velocity and facing direction is then blended at each future frame with the velocity and facing direction predicted by the PFNN in the previous frame ($\mathbf{t}^p_{i+1}\ \mathbf{t}^d_{i+1}$). To do this we use the blending function specified below:

$$\mathrm{TrajectoryBlend}(\mathbf{a}_0,\mathbf{a}_1,t,\tau) = (1 - t^\tau)\ \mathbf{a}_0 + t^\tau\ \mathbf{a}_1, \tag{9}$$

where t ranges from 0 to 1 as the trajectory gets further into the future, and τ represents an additional bias that controls the responsiveness of the character. In the results shown in this paper we set the bias for blending velocities τ_v to 0.5, which results in blending function which biases toward the PFNN predicted velocities, and the bias for blending facing directions τ_d to 2.0, which results in a bias toward the facing direction of the game-pad stick. This produces a character which looks natural yet remains responsive, as most perceived responsiveness comes from when the character is responding quickly to changes in the desired facing direction.

Also included in the future trajectory are variables related to the semantic information of the motion $\mathbf{t}^g$. This includes the desired gait of the character (represented as a binary vector) as well as other information such as the height of the ceiling and if the character is required to jump instead of climb. These are all set either by user interaction (e.g. we use the right shoulder button of the game-pad to indicate the gait should switch to a jog), or by checking the location of the trajectory against elements of the environment (e.g. when the trajectory passes over certain areas the variable indicating a jumping motion is activated).

Once the variables relating to the future trajectory have been found, the final step is to project the trajectory locations vertically onto the scene geometry and extract the heights to prepare $\mathbf{t}^h$. This constitutes all the required input variables for the PFNN, at which point the output $\mathbf{y}$ can be computed.

Given the output $\mathbf{y}$ the final joint transformations are computed from the predicted joint positions and angles $\mathbf{j}^p, \mathbf{j}^a$. These joint transforms are then edited to avoid foot sliding using a simple two-joint IK along with the contact labels $\mathbf{c}$. The root transform position and rotation is updated using the predicted root translational and rotational velocities $\dot{\mathbf{r}}^x, \dot{\mathbf{r}}^z, \dot{\mathbf{r}}^a$. This completes the runtime process for an individual frame.

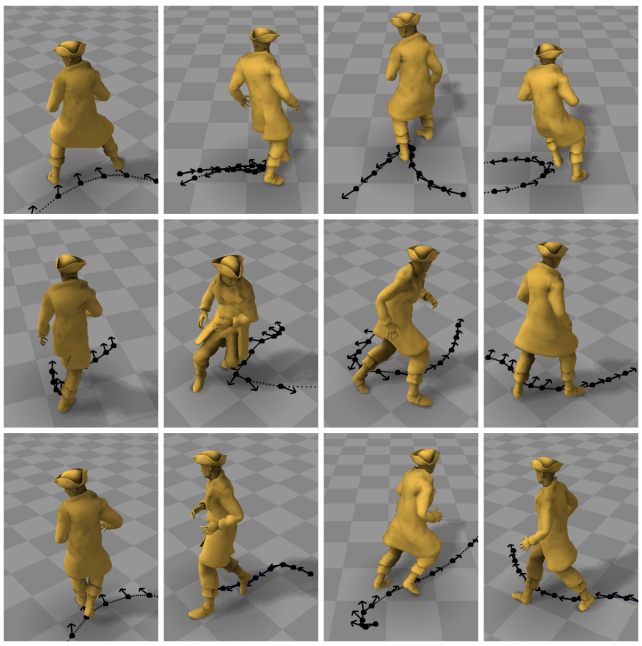

Fig. 6. Results showing the character traversing a planar environment. By adjusting the future trajectory position and direction the user can make the character turn and sidestep.

Precomputation of the Phase Function: Once trained, the PFNN is extremely compact, requiring ~ 10 megabytes to store the variables $\boldsymbol{\beta}$. Yet the phase function Θ may require almost a millisecond of time to compute, which in some cases can be too slow. Because the phase function Θ is a one-dimensional function over a fixed domain $0 \leq p \leq 2\pi$ it is possible to avoid computation of Θ at runtime by precomputing Θ offline for a number of fixed intervals in the range $0 \leq p \leq 2\pi$ and interpolating the results of this precomputation at runtime.

Several options of precomputation are available offering different trade offs between speed and memory consumption (See Table 1). The *constant* method is to precompute Θ for $n = 50$ locations along the phase space and at runtime simply use the neural network weights at the nearest precomputed phase location. This increases the memory consumption by $\frac{n}{4}$ times, but effectively removes the computation of Θ entirely. Alternately, $n = 10$ samples can be taken and a piecewise *linear* interpolation performed of these samples. This approach requires less memory and may be more accurate but the piecewise linear interpolation also requires more computation time. Alternately the full *cubic* Catmull-Rom spline can be evaluated at runtime.

7 RESULTS

In this section we show the results of our method in a number of different situations. For a more detailed demonstration the readers are referred to the supplementary material. All results are shown using the *constant* approximation of the phase function.

In Fig. 6 we show our method applied to a character navigating a planar environment performing many tight turns, changes in speed, and facing directions. Our system remains responsive and adapts well to the user input, producing natural, high quality motion for a range of inputs.

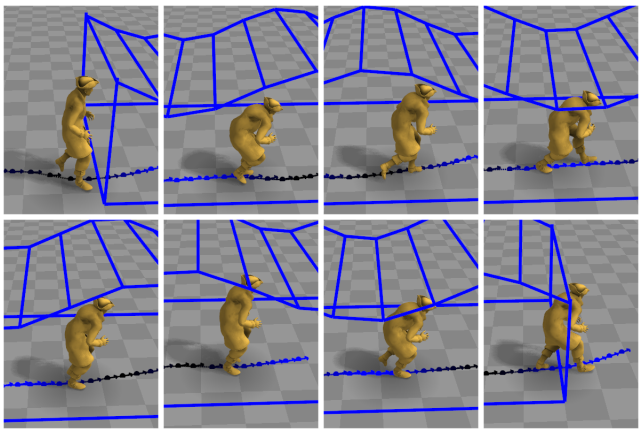

Fig. 7. Results showing the character crouching as the variable indicating the ceiling height is adjusted by the environment.

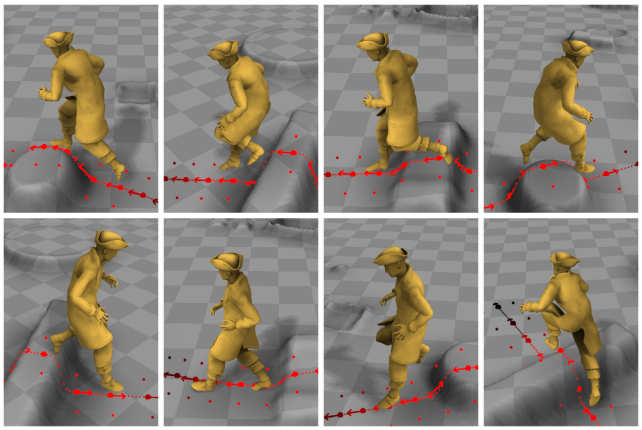

Fig. 8. Results showing the character performing jumping motions over obstacles that have been labeled to indicate jumping should be performed.

In Fig. 1 we show the results of our method applied to a character navigating over rough terrain. Our character produces natural motion in a number of challenging situations, stepping, climbing and jumping where required.

In Fig. 7 we show our method applied in an environment where the ceiling is low such that the character must crouch to proceed. By adjusting a semantic variable in $\mathbf{t}^g$ relating to the height of the ceiling, either using the environment or the game-pad, the character will crouch at different heights.

Sometimes the game designer wants the character to traverse the environment in a different way. By adjusting a different semantic variable in $\mathbf{t}^g$ we can get the character to jump over obstacles instead of climb over them, as shown in Fig. 8.

By colliding the future trajectory with walls or other untraversable objects we can get the character to slow down and avoid obstacles.

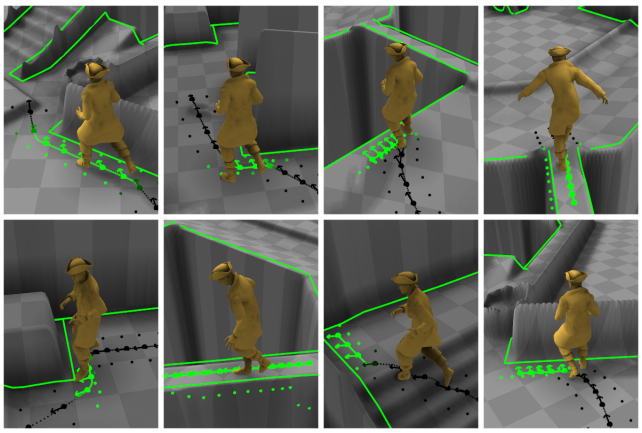

Fig. 9. Result of the character where the future trajectory has been collided with walls, pits and other objects in the environment. By colliding the future trajectory with non-traversable objects the character will slow down or avoid such obstacles. When walking along the beam, since the measured heights on either side of the character are significantly lower than in the center, a balancing motion is naturally produced.

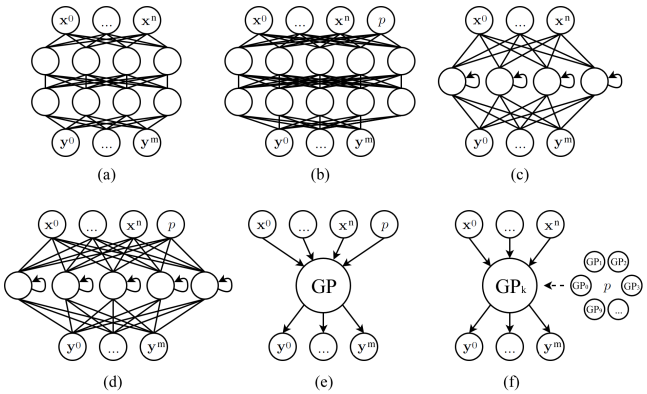

Fig. 10. The configurations of neural network structures and Gaussian process structures evaluated in our comparison: (a) NN without the phase given as input, (b) NN with the phase given as an additional input variable, (c) an ERD network, (d) an ERD network with the phase given as an additional input variable, (e) a GP autoregressor, and (f) a GP autoregressor that is selected using the phase.

Alternately the character can be forced into certain environments to create specific movements such as balancing on a beam, as demonstrated in the urban environment shown in Fig. 9.

8 EVALUATION

In this section we compare our method to a number of other techniques including a standard neural network where the phase is not given as an input (see Fig. 10(a)), a standard neural network where the phase is given as an additional input variable (see Fig. 10(b)), a Encoder-Recurrent-Decoder (ERD) network [Fragkiadaki et al. 2015] (see Fig. 10(c),(d)), an autoregressive Gaussian Process trained on a subset of the data (see Fig. 10(e)) and a similarly structured

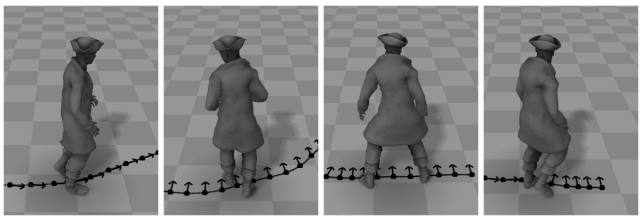

Fig. 11. Results of using a neural network where the phase is given as an additional input. The character motion appears stiff and unnatural as the input relating to the phase is often ignored by the network and other variables used to infer the pose instead.

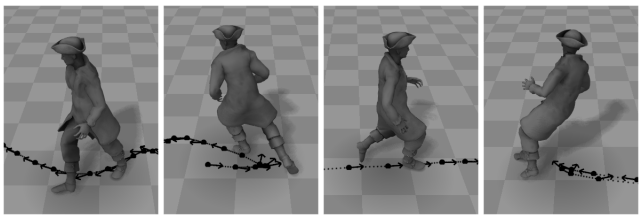

Fig. 12. The Phase Functioned Gaussian Process (PFGP) achieves an effect similar to our method, but the quality of the produced motion and the computational costs are much worse since it cannot learn from as much data.

Gaussian Process approach which builds separate regressors for 10 different locations along the phase space (see Fig. 10(f)). All neural network approaches are trained until convergence and the number of weights in each network adjusted such that the memory usage is equal to our method to provide a fairer comparison.

We show that each of these models fails to produce the same high quality motion seen in our technique and additionally some have fundamental issues which make them difficult to train and use effectively on motion data. We also include a performance comparison detailing training time, runtime cost and memory usage. We then evaluate our data-driven terrain fitting model - comparing the results to a simple procedural model that fits a basic surface to the footstep locations. Finally, we evaluate the responsiveness and following ability of our system by adjusting the blending method of the future trajectory and measuring how these changes affect the results.

Technique	Training	Runtime	Memory
PFNN *cubic*	30 hours	0.0018s	10 MB
PFNN *linear*	30 hours	0.0014s	25 MB
PFNN *constant*	30 hours	0.0008s	125 MB
NN (a) (b)	3 hours	0.0008s	10 MB
ERD (c) (d)	9 hours	0.0009s	10 MB
GP (e)	10 minutes	0.0219s	100 MB
PFGP (f)	1 hour	0.0427s	1000 MB

Table 1. Numerical comparison between our method and other methods described in Fig. 10.

Standard Neural Network. When using a neural network that does not explicitly provide the phase as an input (see Fig. 10, (a)), the system will blend inputs of different phases which results in poses from different phases being averaged and the character appearing to float [Holden et al. 2016]. As our system only blends data at the same phase it does not suffer from such issues.

In an attempt to avoid such erroneous blending it is possible to explicitly provide the phase as an additional input variable to the neural network (see Fig. 10, (b)). During training the influence of the phase is in some cases ignored and the other variables end up dominating the prediction, causing the motion to appear stiff and unnatural (see Fig. 11 and supplementary video). This effect can be partially explained by the usage of dropout [Srivastava et al. 2014], which is an essential procedure for regularization, where input nodes are randomly disabled to make the system robust against over-fitting and noise. As the input variable relating to the phase is often disabled during dropout, the system attempts to dilute its influence where possible, instead erroneously learning to predict the pose from the other input variables. To verify this theory we measure the change in output $\mathbf{y}$ with respect to the change in phase $\|\frac{\delta \mathbf{y}}{\delta p}\|$ in each network. In the standard neural network with the phase given as an additional input this value is relatively small (~ 0.001), while in the PFNN, as the phase is allowed to change all of the network weights simultaneously, it is around fifty times larger (~ 0.05). While it is possible to rectify this somewhat by reducing dropout just for the phase variable, or including multiple copies of the phase variable in the input, these techniques provide a less elegant solution and weaker practical guarantee compared to the full factorization performed by our method.

Encoder-Recurrent-Decoder Network. Next, we compare our system against the Encoder-Recurrent-Decoder (ERD) Network [Fragkiadaki et al. 2015] - one of the current state-of-the-art neural network techniques which is a variant of the RNN/LSTM network structure (see Fig. 10 (c)). The ERD network is produced by taking Φ and making the middle transformation into an LSTM recurrent transformation. Although this structure is not identical to previous versions of the ERD network, the nature of the construction is the same.

As the ERD network has a memory, even when the phase is not given directly, it can learn some concept of phase from the observable data and use this to avoid the issue of ambiguity in the input. The ERD network conducts the recurrent process on the manifold of the motion (middle layer), thus significantly delaying the time that the "dying out" process starts [Fragkiadaki et al. 2015].

Unfortunately some pathological cases still exist - for example if the character is standing still and the user indicates for the character to walk, as it is impossible to observe the phase when the character is stationary, the ERD network cannot know if the user intends for the character to lead with their left foot or their right, resulting in an averaging of the two, and a floating effect appearing. In this sense the phase cannot be learned in all cases, and is a *hidden* latent variable which must be supplied independently as in our method.

Providing the phase as an additional input to this network (see Fig. 10 (d)) can improve the generation performance significantly but still does not entirely remove the floating artefacts (see supplementary video).

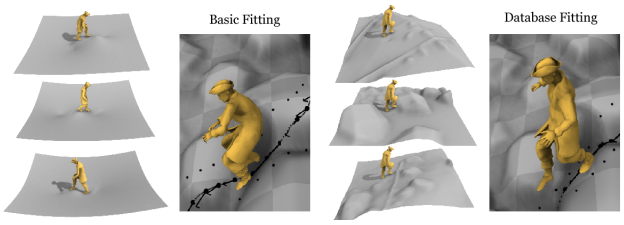

Fig. 13. Comparison of terrain fitting approaches. Simple mesh editing (left) creates a basic surface which touches the footstep locations. In this case the system over-fits to the style of terrains produced by this technique and produces odd motions when terrains unobserved during training are encountered at runtime. Our method (right) uses similar terrains to those encountered at runtime therefore producing more natural motion.

Autoregressive Gaussian Processes. Gaussian Processes (GP) have been applied to autoregressive problems with some success in previous work [Wang et al. 2008]: here we compare our approach with respect to two architectures based on GP.

Firstly, we use a GP to perform the regression from $\mathbf{x}$ to $\mathbf{y}$ with phase given as an additional input (see Fig. 10 (e)): this can be considered similar to a Gaussian Process Dynamic Model (GPDM) [Wang et al. 2008] if you consider the input control parameters to be the latent variables, which in this case are hand-tuned instead of automatically learned. Since it is difficult to avoid the GP over-fitting on the small amount of data it is provided, the system becomes unstable and jittery (see supplementary video). Additionally, the GP cannot adapt to many complex situations since the cost of construction grows in the square order for memory and cubic order for computational cost. Thus, we could only test this system for motions on a planar surface, limiting the training samples to 3,000. Still, it has bad runtime performance and memory costs on large data sets.

Next, we build several independent GPs for 10 different locations along the phase space, selecting the nearest two GPs to perform the regression at runtime and interpolating the result (see Fig. 10(f)). Practically this achieves an effect similar to the PFNN but we find the quality of this regression as well as the memory and runtime performance to be much worse as it cannot be trained on nearly as much data (see Fig. 12).

Performance. In Table 1 we compare the performances of the various methods shown above including memory usage and runtime. GP based techniques are limited by their data capacity, memory usage, and runtime performance. When using the *constant* approximation our method has comparable runtime performance to other neural network based techniques but with a higher memory usage. When using the full *cubic* Catmull-Rom spline interpolation of the phase function it has similar memory usage but with a longer runtime. One downside of our method is that it requires longer training times than other methods. All runtime measurements were made using an Intel i7-6700 3.4GHz CPU running single threaded.

Terrain Fitting. In Fig. 13 we show a comparison to a different terrain fitting process. In this process we start with a flat plane and use the mesh editing technique described in Section 4.2 to deform the surface to touch the footstep locations. Terrain synthesized in such a way are smooth and without much variation. As a result

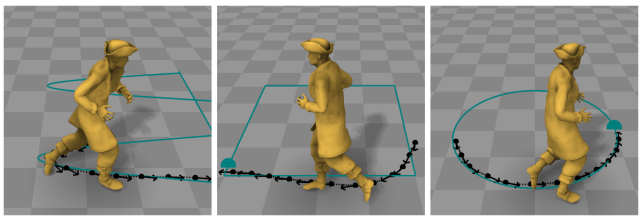

Fig. 14. Evaluation of the responsiveness of our method. The character is directed to follow several predefined paths and the difference between the desired path and actual path made by the character is measured.

Path	τ_v	τ_d	Avg. Error
Wave	0.5	2.0	17.29 cm
	2.0	5.0	10.66 cm
	5.0	10.0	8.88 cm
Square	0.5	2.0	21.26 cm
	2.0	5.0	11.25 cm
	5.0	10.0	8.17 cm
Circle	0.5	2.0	13.70 cm
	2.0	5.0	8.03 cm
	5.0	10.0	5.82 cm

Table 2. Numerical evaluation of character responsiveness and following ability. For each scene in Fig. 14 we measure the average error between the desired path and that taken by the character with different biases supplied to the future trajectory blending function Eq. (9). Here τ_v represents the blending bias for the future trajectory velocity, and τ_d represents the blending bias for the future trajectory facing direction (see Section 6 for a more detailed explanation).

of over-fitting to such smooth terrains the system produces odd motions during runtime when the character encounters environments such as those with large rocks. The readers are referred to the supplementary material for further details.

Responsiveness. In Fig. 14 we show an evaluation of the responsiveness and following ability of our method. We create several predefined paths and instruct the character to follow them. We then measure the difference between the desired trajectory and the actual trajectory of the character (see Table 2). By increasing the variable τ (described in Section 6) responsiveness can be improved at the cost of a small loss in animation quality. The readers are referred to the supplementary material for further details.

9 DISCUSSIONS

In this section, we first discuss about the network architecture that we adopt, and also about the input/output parameters of the system. Finally we discuss about the limitations of our system.

Network Architecture. As shown in our experiments, our system can compute realistic motions in a time-series manner, without suffering from issues such as dying out or instability which often occur in autoregressive models. This is made possible through several system design decisions. By using the phase as a global parameter for the weights of the neural network we can avoid mixing motions at different phases which notoriously results in the "dying out" effect. This also ensures the influence of the phase is strongly taken into account during training and runtime, and thus there is little chance that its influence is diluted and ignored, something we observed when giving the phase as an additional input parameter to other network structures.

Our network is composed of one input layer, one output layer, and two hidden layers with 512 hidden units. This design is motivated by the wish to make the network as simple as possible to ensure the desired motion can be computed easily for our real-time animation purpose. We find the current network structure is a good compromise between computational efficiency and the richness of the movements produced.

While we only demonstrate the PFNN applied to cyclic motions, it can just as easily be applied to non-cyclic motions by adopting a non-cyclic phase function. In non-cyclic data the phase can be specified as 0 at the start of the motion, 0.5 half way through and 1.0 at the end. If trained with a non-cyclic phase function and appropriate control parameters, the PFNN could easily be used on other tasks such as punching and kicking.

Conceptually, our system is similar to training separate networks for each phase. This is something we tried in early iterations of our research, but when doing so each network learned a slightly different regression due to the random weight initialization and other factors. There was therefore no continuity or cyclic nature preserved and the motion looked jittery, with a noticeable "seam" when the phase looped around. The PFNN provides an elegant way of achieving a similar effect but without the stated problems.

Control Parameters for Real-time Character Control. Our system is designed specifically for real-time character control in arbitrary environments that can be learned from motion capture data: for this purpose we use an input parameterization similar to one that has proven to be effective in Motion Matching [Clavet 2016]. The future trajectory is predicted from the user inputs while the input related to the environment is found by sampling the heights of the terrain under the trajectory of the character. Using a window of past and future positions, directions, and geometry reduces the potential for ambiguity and produces higher quality motion. For additional tasks it should be possible to adapt this input parameterization, for example to include extra input variables related to the style of the motion.

Other Potential Parameterization. Deep Learning has found that it is possible to use input parameterizations which are not hand crafted in this way but use more neural network layers to perform the abstraction. For example - it is possible for our purposes to use a depth or RGB image of the surrounding terrain as the input relating to the environment and use convolutional layers for abstracting this input. This is an interesting approach and may be a good solution especially for robotics, where the actions are limited and the details of the ground are needed for keeping balance. In the early stages of our research we tested an approach similar to this, but noticed a few issues. Firstly, using a CNN required a lot more training data as we found that the character does not interact with any terrain along the sides of the images, meaning we needed a large variety of different images to prevent over-fitting in these places. We also found that

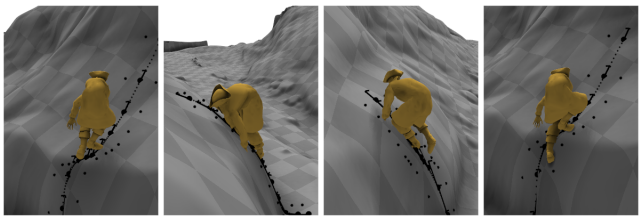

Fig. 15. If the input trajectory is unachievable such as when the terrain is too steep our approach will extrapolate which can often produce bad looking motion.

when using convolutional layers or more complex neural network structures to abstract the input the processing time increased to a point where it became unsuitable for real-time applications such as computer games.

Limitations & Future Work. In order to achieve the real-time performance we only coarsely sample points along the trajectory. This results in the system missing some high resolution details such as sharp small obstacles along the terrain, which need to be avoided in actual applications. One simple solution is to have an additional layer of control on top of the current system to respond to labeled obstacles, while using our system for low frequency geometry of the environment.

Like many methods in this field our technique cannot deal well with complex interactions with the environment - in particular if they include precise hand movements such as climbing up walls or interacting with other objects in the scene. In the future labeling hand contacts and performing IK on the hands may help this issue partially. Alternately, performing the regression in a space more naturally suited to interactions such as that defined by relationship descriptors [Al-Asqhar et al. 2013] may be interesting and produce compelling results.

The PFNN is relatively slow to train as each element in the mini-batch produces different network weights meaning the computation during training is much more expensive than usual. The PFNN can produce acceptable results for testing after just a couple of hours of training, but performing the full 30 hour training each time new data is added is not desirable. For this reason we are interested in ways to speed up the training of the PFNN or ways of performing incremental training.

If the user supplies an input trajectory which is unachievable or invalid in the given context (e.g. the terrain is too steep) our system will extrapolate which may produce undesirable results (see Fig. 15). Additionally the results of our method may be difficult to predict, and therefore hard for artists to fix or edit. A dedicated method for editing and controlling the results of techniques such as ours is therefore desirable.

Another future work can be applying our framework for physically-based animation. For example, it will be interesting to learn a non-linear version of a phase-indexed feedback model [Liu et al. 2016] in addition to a feedforward controller. Such a system may allow the character to stably walk and run over terrains in different physical conditions such as slippery floors, or unstable rope bridges.

Finally, it is also interesting to apply our technique for other modalities, such as videos of periodic data e.g. fMRI images of heartbeats. Using a periodic model such as the PFNN can potentially make the learning process more efficient for such kinds of data.

10 CONCLUSION

We propose a novel learning framework called a Phase-Functioned Neural Network (PFNN) that is suitable for generating cyclic behavior such as human locomotion. We also design the input and output parameters of the network for real-time data-driven character control in complex environments with detailed user interaction. Despite its compact structure, the network can learn from a large, high dimensional dataset thanks to a phase function that varies smoothly over time to produce a large variation of network configurations. We also propose a framework to produce additional data for training the PFNN where the human locomotion and the environmental geometry are coupled. Once trained our system is fast, requires little memory, and produces high quality motion without exhibiting any of the common artefacts found in existing methods.

REFERENCES

Rami Ali Al-Asqhar, Taku Komura, and Myung Geol Choi. 2013. Relationship Descriptors for Interactive Motion Adaptation. In *Proc. SCA*. 45–53. DOI:https://doi.org/10.1145/2485895.2485905

James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. 2010. Theano: a CPU and GPU Math Expression Compiler. In *Proc. of the Python for Scientific Computing Conference (SciPy)*. Oral Presentation.

Mario Botsch and Leif Kobbelt. 2005. Real-Time Shape Editing using Radial Basis Functions. *Computer Graphics Forum* (2005). DOI:https://doi.org/10.1111/j.1467-8659.2005.00886.x

Jinxiang Chai and Jessica K. Hodgins. 2005. Performance Animation from Low-dimensional Control Signals. *ACM Trans on Graph* 24, 3 (2005). DOI:https://doi.org/10.1145/1186822.1073248

Jinxiang Chai and Jessica K. Hodgins. 2007. Constraint-based motion optimization using a statistical dynamic model. *ACM Trans on Graph* 26, 3 (2007). DOI:https://doi.org/10.1145/1275808.1276387

Simon Clavet. 2016. Motion Matching and The Road to Next-Gen Animation. In *Proc. of GDC 2016*.

Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. 2015. Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs). *CoRR* abs/1511.07289 (2015). http://arxiv.org/abs/1511.07289

Stelian Coros, Philippe Beaudoin, Kang Kang Yin, and Michiel van de Pann. 2008. Synthesis of constrained walking skills. *ACM Trans on Graph* 27, 5 (2008), 113. DOI:https://doi.org/10.1145/1409060.1409066

Katerina Fragkiadaki, Sergey Levine, Panna Felsen, and Jitendra Malik. 2015. Recurrent network models for human dynamics. In *Proc. ICCV*. 4346–4354. DOI:https://doi.org/10.1109/ICCV.2015.494

Helmut Grabner, Juergen Gall, and Luc Van Gool. 2011. What makes a chair a chair?. In *Proc. IEEE CVPR*. 1529–1536. DOI:https://doi.org/10.1109/CVPR.2011.5995327

F. Sebastin Grassia. 1998. Practical Parameterization of Rotations Using the Exponential Map. *J. Graph. Tools* 3, 3 (March 1998), 29–48. DOI:https://doi.org/10.1080/10867651.1998.10487493

Keith Grochow, Steven L Martin, Aaron Hertzmann, and Zoran Popović. 2004. Style-based inverse kinematics. *ACM Trans on Graph* 23, 3 (2004), 522–531. DOI:https://doi.org/10.1145/1186562.1015755

Abhinav Gupta, Scott Satkin, Alexei A Efros, and Martial Hebert. 2011. From 3d scene geometry to human workspace. In *Proc. IEEE CVPR*. 1961–1968. DOI:https://doi.org/10.1109/CVPR.2011.5995448

Daniel Holden, Jun Saito, and Taku Komura. 2016. A deep learning framework for character motion synthesis and editing. *ACM Trans on Graph* 35, 4 (2016). DOI:https://doi.org/10.1145/2897824.2925975

Daniel Holden, Jun Saito, Taku Komura, and Thomas Joyce. 2015. Learning Motion Manifolds with Convolutional Autoencoders. In *SIGGRAPH Asia 2015 Technical Briefs*. Article 18, 4 pages. DOI:https://doi.org/10.1145/2820903.2820918

Nicholas R Howe, Michael E Leventon, and William T Freeman. 1999. Bayesian Reconstruction of 3D Human Motion from Single-Camera Video.. In *Proc. NIPS*. http://papers.nips.cc/paper/1698-bayesian-reconstruction-of-3d-human-motion-from-single-camera-video

Changgu Kang and Sung-Hee Lee. 2014. Environment-Adaptive Contact Poses for Virtual Characters. In *Computer Graphics Forum*, Vol. 33. Wiley Online Library, 1–10. DOI: https://doi.org/10.1111/cgf.12468

Mubbasir Kapadia, Xu Xianghao, Maurizio Nitti, Marcelo Kallmann, Stelian Coros, Robert W Sumner, and Markus Gross. 2016. Precision: precomputing environment semantics for contact-rich character animation. In *Proc. I3D*. 29–37. DOI: https://doi.org/10.1145/2856400.2856404

Vladimir G. Kim, Siddhartha Chaudhuri, Leonidas Guibas, and Thomas Funkhouser. 2014. Shape2pose: Human-centric shape analysis. *ACM Trans on Graph* 33, 4 (2014), 120. DOI: https://doi.org/10.1145/2601097.2601117

Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *CoRR* abs/1412.6980 (2014). http://arxiv.org/abs/1412.6980

Tejas D Kulkarni, William F Whitney, Pushmeet Kohli, and Josh Tenenbaum. 2015. Deep convolutional inverse graphics network. In *Proc. NIPS*. 2539–2547. http://papers.nips.cc/paper/5851-deep-convolutional-inverse-graphics-network.pdf

Manfred Lau and James J Kuffner. 2005. Behavior planning for character animation. In *Proc. SCA*. DOI: https://doi.org/10.1145/1073368.1073408

Jehee Lee, Jinxiang Chai, Paul SA Reitsma, Jessica K Hodgins, and Nancy S Pollard. 2002. Interactive control of avatars animated with human motion data. *ACM Trans on Graph* 21, 3 (2002), 491–500. DOI: https://doi.org/10.1145/566570.566607

Jehee Lee and Kang Hoon Lee. 2004. Precomputing avatar behavior from human motion data. *Proc. SCA* (2004), 79–87. DOI: https://doi.org/10.1145/1028523.1028535

Kang Hoon Lee, Myung Geol Choi, and Jehee Lee. 2006. Motion patches: building blocks for virtual environments annotated with motion data. *ACM Trans on Graph* 25, 3 (2006), 898–906. DOI: https://doi.org/10.1145/1141911.1141972

Yongjoon Lee, Kevin Wampler, Gilbert Bernstein, Jovan Popović, and Zoran Popović. 2010. Motion fields for interactive character locomotion. *ACM Trans on Graph* 29, 6 (2010), 138. DOI: https://doi.org/10.1145/1882261.1866160

Sergey Levine, Jack M Wang, Alexis Haraux, Zoran Popović, and Vladlen Koltun. 2012. Continuous character control with low-dimensional embeddings. *ACM Trans on Graph* 31, 4 (2012), 28. DOI: https://doi.org/10.1145/2185520.2185524

Libin Liu, Michiel van de Panne, and KangKang Yin. 2016. Guided Learning of Control Graphs for Physics-Based Characters. *ACM Trans on Graph* 35, 3 (2016). DOI: https://doi.org/10.1145/2893476

Libin Liu, KangKang Yin, Michiel van de Panne, Tianjia Shao, and Weiwei Xu. 2010. Sampling-based contact-rich motion control. *ACM Trans on Graph* 29, 4 (2010), 128. DOI: https://doi.org/10.1145/1778765.1778865

Wan-Yen Lo and Matthias Zwicker. 2008. Real-time planning for parameterized human motion. In *Proc. I3D*. 29–38. http://dl.acm.org/citation.cfm?id=1632592.1632598

Roland Memisevic. 2013. Learning to relate images. *IEEE PAMI* 35, 8 (2013), 1829–1846. DOI: https://doi.org/10.1109/TPAMI.2013.53

Jianyuan Min and Jinxiang Chai. 2012. Motion graphs++: a compact generative model for semantic motion analysis and synthesis. *ACM Trans on Graph* 31, 6 (2012), 153. DOI: https://doi.org/10.1145/2366145.2366172

Tomohiko Mukai. 2011. Motion rings for interactive gait synthesis. In *Proc. I3D*. 125–132. DOI: https://doi.org/10.1145/1944745.1944767

Tomohiko Mukai and Shigeru Kuriyama. 2005. Geostatistical motion interpolation. *ACM Trans on Graph* 24, 3 (2005), 1062–1070. DOI: https://doi.org/10.1145/1073204.1073313

Sang Il Park, Hyun Joon Shin, and Sung Yong Shin. 2002. On-line locomotion generation based on motion blending. In *Proc. SCA*. 105–111. DOI: https://doi.org/10.1145/545261.545279

Xue Bin Peng, Glen Berseth, and Michiel van de Panne. 2016. Terrain-Adaptive Locomotion Skills Using Deep Reinforcement Learning. *ACM Trans on Graph* 35, 4 (2016). DOI: https://doi.org/10.1145/2897824.2925881

Carl Edward Rasmussen and Zoubin Ghahramani. 2002. Infinite mixtures of Gaussian process experts. In *Proc. NIPS*. 881–888. http://papers.nips.cc/paper/2055-infinite-mixtures-of-gaussian-process-experts

Charles Rose, Michael F. Cohen, and Bobby Bodenheimer. 1998. Verbs and Adverbs: Multidimensional Motion Interpolation. *IEEE Comput. Graph. Appl.* 18, 5 (1998), 32–40. DOI: https://doi.org/10.1109/38.708559

Alla Safonova and Jessica K Hodgins. 2007. Construction and optimal search of interpolated motion graphs. *ACM Trans on Graph* 26, 3 (2007), 106. DOI: https://doi.org/10.1145/1275808.1276510

Alla Safonova, Jessica K Hodgins, and Nancy S Pollard. 2004. Synthesizing physically realistic human motion in low-dimensional, behavior-specific spaces. *ACM Trans on Graph* 23, 3 (2004), 514–521. DOI: https://doi.org/10.1145/1186562.1015754

Manolis Savva, Angel X. Chang, Pat Hanrahan, Matthew Fisher, and Matthias Nießner. 2016. PiGraphs: Learning Interaction Snapshots from Observations. *ACM Trans on Graph* 35, 4 (2016). DOI: https://doi.org/10.1145/2992138.2992147

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *The Journal of Machine Learning Research* 15, 1 (2014), 1929–1958. http://dl.acm.org/citation.cfm?id=2627435.2670313

Jochen Tautges, Arno Zinke, Björn Krüger, Jan Baumann, Andreas Weber, Thomas Helten, Meinard Müller, Hans-Peter Seidel, and Bernd Eberhardt. 2011. Motion reconstruction using sparse accelerometer data. *ACM Trans on Graph* 30, 3 (2011), 18. DOI: https://doi.org/10.1145/1966394.1966397

Graham W Taylor and Geoffrey E Hinton. 2009. Factored conditional restricted Boltzmann machines for modeling motion style. In *Proc. ICML*. ACM, 1025–1032. DOI: https://doi.org/10.1145/1553374.1553505

Jack M. Wang, David J. Fleet, and Aaron Hertzmann. 2008. Gaussian Process Dynamical Models for Human Motion. *IEEE PAMI* 30, 2 (2008), 283–298. DOI: https://doi.org/10.1109/TPAMI.2007.1167

Shihong Xia, Congyi Wang, Jinxiang Chai, and Jessica Hodgins. 2015. Realtime style transfer for unlabeled heterogeneous human motion. *ACM Trans on Graph* 34, 4 (2015), 119. DOI: https://doi.org/10.1145/2766999