Using a Genetic Algorithm to Implement Pose Tracking for Locomotion Control

ETHAN WILSON

Advised By:

GREG TURK, YUNBO ZHANG

1 Introduction

The goal of this project is to create an animation system that can import custom models and generate stable walk cycles (i.e. completing multiple cycles without falling over, yet still moving in the forward direction) within an affordable amount of time. There are several existing ways to animate a character; the most common methods include keyframes, motion capture, and physics-based animations.

Animated motion of a digital character is derived from a few different methods which depend on the simulation environment. Traditional animation, applied in video games, animated film, etc., keeps a record of keyframes, or values for the rotation of joints within a model's underlying skeleton. These keyframes correspond to different points in time, and the program interpolates between them to generate the animation. These keyframes are typically animated by hand; therefore, they might not be physically accurate motions.

Motion capture is a method that relies on tracking markers on a real person using video cameras. These markers typically are placed near important joints on the person. This motion captured data can then be used to animate a character with a similar build to the physical model, or it can be remapped to a different model entirely. When data is retargeted to a new model, it may become physically implausible.

Animations that must conform to the laws of physics, i.e. existing within physics simulations, must be animated in a different manner. Real humans / creatures perform an infinite amount of subconscious motions to preserve balance and save energy, but it would take an exorbitant amount of time for a human animator to replicate those balance changes using keyframes in a physical simulation. Without those corrections, realistic-looking animation loops might lose their stability over time, resulting in the agent falling over, and successfully human-animated motion would likely look robotic or simulated.

My approach is physics-based and meant to be stable for as long as possible within a physical simulation. In order to achieve this, I have implemented a genetic algorithm that finds and fine-tunes keyframe poses for a given model.

2 Related Work

Traditionally, the solution for animating with respect to physical constraints has been to employ some heuristic optimization algorithm. Heuristic algorithms do not search for a single perfect solution; instead they approximate a solution that is "good enough" and can be found in a reasonable amount of time. These algorithms are especially effective in locomotion because of the large problem space (there are infinite variations of acceptable forms of locomotion). Evolutionary algorithms work particularly well as they tend to avoid local maxima and can indefinitely continue searching for a more efficient solution. For example, Kodjabachian used an evolutionary approach to build a neural network controller for simulated two-dimensional insects. These insects were able to learn locomotion and perform higher level behaviors, such as obstacle avoidance and an odor-gradient food detection system [1].

Recent improvements in computational technology have opened the gateway to apply more computationally expensive algorithms. Reinforcement learning (RL), a subset of machine learning that balances an agent's environmental exploration vs. exploitation, has become an especially popular method. There are various existing implementations of RL locomotion training, an example being DeepLoco for bipedal locomotion. The DeepLoco approach implements two RL controllers that operate cooperatively. The high-level controller makes broad decisions such as where to step to reach the target location, while the lower-level controller performs quick computations such as balance corrections [2]. However, these solutions remain too computationally expensive to perform without a dedicated setup. While a research lab might be able to afford a week of training on a dedicated multi-GPU system, I am restricted to using a mid-range laptop.

Some locomotion implementations have extended the core walking functionality for various scenarios. DeepLoco's high-level goal-setting implementation allowed the model to follow non-uniform paths or even dribble a soccer ball. Another example is Simbicon, a three-dimensional biped locomotion controller that incorporates a state machine populated with distinct poses to interpolate between. This approach allows models to react to varied terrain and outside forces by choosing an appropriate pose and performing simultaneous balance computations [3].

3 My Approach

The goal of my system is to create a locomotion controller for a given character anatomy. My implementation used a Genetic Algorithm (GA), an evolutionary algorithm in which various character controllers compete within a population that evolves over time. The best character controllers (the ones that move the character the farthest) are propagated to the next generation, possibly with crossover and mutation. In order to save computation time and limit the problem's search space, I opted to model the character's genome after the keyframe system used in traditional animation. The genome of a particular controller is a small collection of character poses (typically 2 to 4) that are represented as joint angles. The motion for the character is given by interpolating between this set of poses.

3.1 Character Controller within DART

The physics simulation I chose to use was the Dynamic Animation and Robotics Toolkit (DART). DART is an open-source library that was written at the Georgia Institute of Technology collaboratively by the Graphics Lab and Humanoid Robotics Lab [4]. I chose DART both because it is an ideal environment for simulating rigid body systems and because of my advisors' experience with the simulator.

DART's functionality includes a customizable controller for every rigid body entity within the simulation. This controller can be used to dynamically apply torque to the joints of the rigid body during a given simulation. The controller I implemented was an extension of the Stable Proportional-Derivative (SPD) controller [5]. In the context of a rigidbody system, a SPD controller provides stable tracking to a target pose. This is achieved by applying a control force to the rigid body's joints, computed using the joint's current kinematics, the target position, and the simulation's timestep. A proportional gain (tracking strength) and derivative gain (damping strength) are included in the equation used and can be changed to tweak the tracking intensity. SPD controllers differ from traditional Proportional-Derivative (PD) controllers in their calculation; SPD use a computed estimate of the simulation's next time-step rather than the current time-step. This method ensures stability when undergoing large motions or high time steps, whereas the traditional PD controller may break down.

My controller utilizes the SPD controller's computations for all torque calculations. At the initialization phase of the simulation, my controller is fed a list of tracking poses. Then, when applicable, these poses are mirrored and/or interpolated to generate a large cycle of poses for the controller to use.

Many characters have a symmetric anatomy and use the same (mirrored) motions on both sides of their body. To mirror a pose, specific values within the pose (referred to as degrees of freedom, explained in-depth in a later section) that map from left to right are transferred to the opposite side, and vice versa. The code to mirror depends on the model at hand. In effect, mirroring doubles the number of keyframe poses for a given character controller.

During locomotion, the character's pose is given by smooth interpolation between the given keyframes. To interpolate, two adjacent poses from within the loop are chosen. Multiple intermediate poses are generated by linearly interpolating between the two chosen poses. These generated poses are then added to a new list which ends up being much larger. This process is repeated between each adjacent pair of poses, with the interpolated poses being added to the same new list, which finally replaces the initial list of poses.

At runtime, the controller tracks to a pose and constantly computes an error value between the model's current position and the pose being tracked. Once this error drops beneath a specific value, the target pose shifts to the next pose in the list. Using error to determine when to track a new pose differs from animation's traditional keyframe approach, where the tracking of keyframes are based completely on time. My approach generated smoother motions than a linear time-based system would; additionally, this system discouraged unexpected behaviors by adding physical constraints to the rigidbody. If a model was unable to reach its next pose (from falling over and being blocked by the ground, or any other reason), it would get stuck, whereas a strictly time-based keyframe system's model would ignore the fact that it was stuck and continue attempting to reach poses, resulting in unnatural, spastic motion.

3.2 Genetic Algorithm

A genetic algorithm (GA) is a framework for solving optimization problems which is based upon the process of natural selection found in biological evolution. Different potential solutions to a given problem, referred to as genomes, are initially randomly generated by the algorithm. These genomes are stored in a list of fixed size known as the population. The algorithm progresses by alternating between evolutionary and evaluation stages. At each evaluation stage, every genome in the population is tested against a fitness function and given a score which corresponds to how effectively they have solved the problem. The fitness functions in my implementation are customized per-model, but all functions are built upon giving a high reward for the longest distance travelled. The evolutionary stage typically involves a combination of crossover and mutation. Crossover is a process in which two genomes, now known as the parents, are selected from the population and their information is combined to produce some number of children. Mutation takes in a single genome sequence and slightly changes some of its values, creating a slightly different solution to the problem. Crossover and mutations are performed on the population in an effort to produce better solutions that can be used in later generations. At the end of each evolutionary stage, the population, which may contain extra genome data from crossover, must be culled back to its original size.

3.3 Data Storage

As described above, I chose to represent the genomes as a list of target poses for the controller within DART to use. Every pose was a list of values, and each value corresponded to the rotation value of one degrees of freedom within the model.

A model is composed of various body sections, which are connected by joints. There are multiple joint classifications to choose from, with the more complicated joint types having more degrees of freedom (DOFs). A DOF is a single value that corresponds to one axis of rotation in the world (the axis can be of any orientation, however). The types of joints are as follows:

1. Revolute joints contain one DOF. An example would be a human's elbow, which can only rotate along the bicep's orientation. Notice, however, that as the shoulder moves the elbow's global rotation also changes, so the revolute joint's constraint is only local to the body part it is placed on.
2. Universal joints contain two DOFs. An example is the human knee, which can rotate along the thigh's orientation or rotate slightly along the calf's axis. It can not rotate all the way around the thigh due to having two axes of rotation in a 3D space.
3. Euler joints contain three DOFs, and can rotate along all axes. An example is the shoulder. Notice that the shoulder still has value constraints. It can rotate along every axis, but the extent of rotation is limited by the person's flexibility.
4. Free joints connect two separate objects and contain three DOFs for rotation and three DOFs for position. These joints can not be directly manipulated and exist solely to inform the simulation.
5. Weld joints have zero DOFs and rigidly connect two body parts. These help to create more complex geometry by combining primitive shapes.

Limiting the number of DOFs on a model would decrease computation time, so models were created creatively to have the simplest combination of joints while still being able to achieve motion similar to the creatures they were modeled after. In addition, I limited the values of these DOFs within the genome to have a roughly 90 degree range. This restriction acted similarly to a human shoulder in that without constraints, a shoulder would be able to clip into its attached body. This helped to both keep movement natural and to decrease the problem's search space significantly.

To save additional computation time, for symmetrical models I implemented a mirror system. Each genome would only contain info for half of a walk cycle, which is then mirrored over at the beginning of the evaluation phase. This halved the amount of data in the GA, which decreased search time.

3.4 Fitness Evaluation

Every evaluation stage of the GA, each genome needs to be run against a fitness function so that they can be ranked as effective solutions to the problem at hand. My fitness functions varied for different models, but were all based around running a DART simulation for a specific amount of time and returning a value based on how far forward the model travelled.

Some models needed more complicated fitness functions to accommodate unexpected behaviors that arise. The agent's only goal is to get the highest score, so without extra bounds to restrict movement, the best solution the GA founds could be an unnatural looking exploit. For example, when given a simple "go forward" fitness function, my quadruped model discovered that it could travel fastest by somersaulting instead of walking. To fix this, I added the constraint in which the agent's average foot-to-ground distance influenced the highest score. This heavily penalized the somersault behavior without discouraging a regular walk cycle.

The most common exploit agents found were that they could use their poses to perform one large jump instantly, and the distance they traveled would be further than an actual walk cycle. The agent would get stuck at this local maxima for many generations. I hypothesize that given a large enough simulation runtime and enough training generations this behavior would self-correct, but I did not have the patience to run any tests long enough to see this happen (three days were not long enough). More specific results will be explained in a later section.

3.5 Evolution

My methods for mutation and crossover were customized to accomodate my specific locomotion problem. My mutation function would take a genome sequence, choose a random pose from within the list, and randomly select two DOFs to change. These DOFs were reassigned random values within the 90 degree target range. My mutation function's goal was to discover slight tweaks that could finely tweak the balance of a model's walk cycle. My method for crossover focused only on the poses, ignoring the specific values contained in each pose. Two parent genomes were chosen randomly from the population and their pose combinations were permuted to produce unique children. The parents and children were all returned into the population.

The evolution stage of my algorithm would repeatedly perform crossovers between random parents. This grew the overall population, so the repeated crossovers terminated once the new population reached a certain size. At that point, the entire [large] population would be evaluated and sorted based on the genomes' fitness evaluation. Then the population would be culled back to its original size, with checks put in place to prevent identical genome copies from surviving. By only performing crossovers, the population eventually would normalize and mix to the point of little genetic variation.

The algorithm kept track of a population's average fitness as well as the highest individual and underwent a special period of "forced evolution" if no significant growth was made over multiple generations. This forced evolution period consisted of rapid mutations and the generation of entirely random new genome. If a generated genome was deemed fit, it would be inserted into the population. Once the population had been sufficiently revitalized, the algorithm would continue and the regular crossover evolution would resume. The forced evolution stage's purpose was to work around the local maxima and broaden the algorithm's search space to find better solutions. As the found solution became better, the forced evolutionary period would take longer and would (typically) result in less extreme jumps in progress. This gave me insight for when to terminate the program, as progress would eventually slow enough to not be worth the runtime.

4 Results

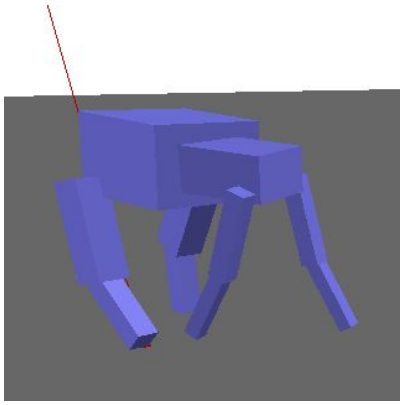
My goal to create stable physics-based walk cycles was achieved. Despite having limited hardware, my learning algorithm provided consistent, quick results. Population size ranged between 30 and 50 during testing. The algorithm would typically converge upon a solution within 8 hours, with simpler models taking less time. This timeframe translates to roughly 75 generations, but could be less or more depending on the length of each simulation (typically 5 - 8 seconds) and the amount of forced evolutionary periods triggered. Tests that ran longer had only marginal efficiency gains.

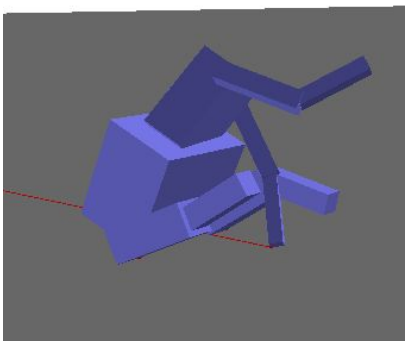
A secondary goal was for the motion to look realistic and mimic that of real creatures' walk cycles. While none of my motion looked robotic, it also did not look completely organic. The walk cycles tended to look sluggish. Smaller motions that do not change the model's center of gravity are typically more stable, so the solutions the algorithm produced tended not to have large movements. Another reason is the manually set tracking and damping strength. I set these values relatively low, which resulted in weaker creatures. I did this mainly because weaker creatures have less unexpected behaviors; therefore, their fitness functions' constraints are easier to write and quicker to test.

Some of the unexpected behaviors were insightful and quite entertaining to look at. For example, the quadruped that opted to somersault shows what organic motion might have been like if there was no concept of tiredness. Some models were designed with specific motions in mind, but performed entirely differently. I designed a creature with two legs and a large tail, thinking it would propel itself forward with the tail; instead, it decided to lift the tail in the air and crawl.

I discovered that I could add early termination to the fitness functions of some models to speed up the training period. If I could identify a behavior that a walk cycle would never have, such as the model being upside down or high up in the air, a conditional in my code would give that genome a very low score and terminate the simulation early. This shortcut improved early performance greatly, when genome were prone to random motion, and become less impactful further on as genome converged to a stable solution.

4.1 Short Quadruped





This was my original test model. I modeled it to loosely represent a dog, while abstracting it to be as simple as possible without discouraging interesting motion. This model was great for benchmarking different approaches during development, as it was simple enough to learn very quickly and for me to rationalize why it was moving the way it was.

Interesting Behaviors:

1. When initially implementing a fitness function, I passed in a simple "go forward" reward function. This model learned how to do somersaults rather than walk.
2. At one point, the model found a local maxima where it sat down and used its front legs to scoot forward.

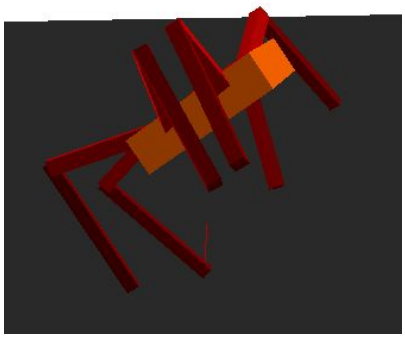
4.2 Tall Quadruped

This model is a modified version of the other quadruped that I created to test whether the algorithm would work in a general sense. I narrowed the back legs and widened the front to lower the overall stability and make the model harder to balance. Despite these changes, my algorithm worked well.

Interesting behaviors:

1. The model relied heavily on its spine joint, which gave it a swaying motion that looked very animated / stylized.

4.3 Crab

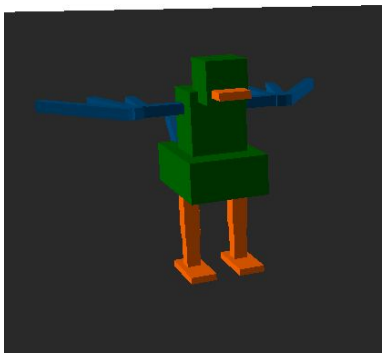


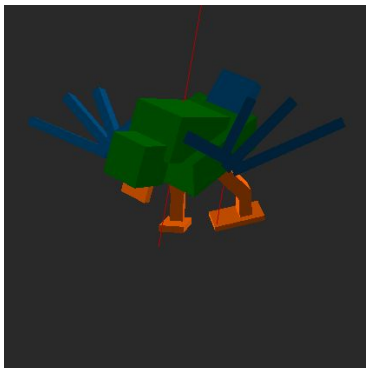
This model was a test to see if my algorithm could be extrapolated to a hexapod (or models with more legs, theoretically). In most cases, the crab would find a motion that flipped it over onto its back and allowed it to then walk on its "elbow" joints. The resulting motion was slow and didn't mimic real life, but was extremely stable and allowed the model to effectively ignore its extremities' DOF values, giving it a more narrow problem space.

Interesting behaviors:

1. In one case, with a low simulation time, the model launched itself forward off of its back legs and propelled itself forward like a biped sprinting. With a higher simulation time (after the sprinting motion would knock itself over), it learned that it was more efficient in the long run to flip onto its back.

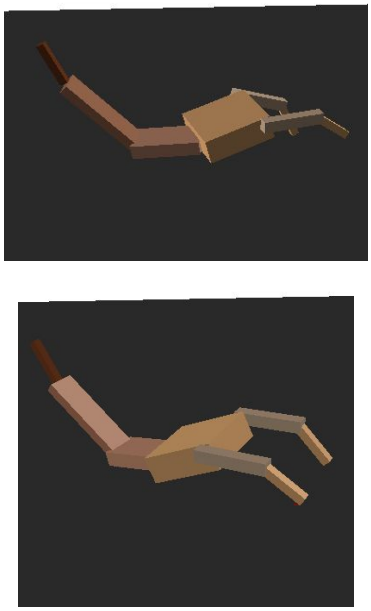
4.4 Duck





The duck model was a test to see if the algorithm could be applied to a biped. I was a bit overenthusiastic and made too complicated a model; it had no level of success learning to walk. I intended for the model to have more balance options by controlling the wing-shoulder joints. Instead, it collapsed upon itself almost immediately on every test.

4.5 Alien



This model was a breakaway from modeling after established creature skeletons found in nature. The model was given two short front legs that couldn't generate much power and a large tail that could potentially be used to propel the model forward.

Interesting behaviors:

1. When I created the model, I intended for the tail to be a key factor in movement. Instead the algorithm determined that holding the tail in the air (completely ignoring it, as if it was dead weight) was the most efficient strategy.
2. Because this model was extremely simple compared to its counterparts, it reached a stable motion cycle extremely quickly. This model took about 6 generations to build a convincing crawl, while the quadrupeds needed about 50 generations to reach a stable walk cycle.

5 Future Implications

This method of locomotion learning is structured very similarly to the keyframe system present in traditional animation. Because of this similarity, traditional animation skeletons could be ported into the learning algorithm. If this method were refined and tweaked to search for a local maxima based on the ported animation, the program would be able to finely tweak hand-made animations to become stable and physically accurate. Obviously, hand-made animation is made to look realistic rather than be realistic, so there is no guarantee that a physically accurate animation would look better. If nothing else, a video game or film that employed this technique would have some added novelty. As an alternative, animators could simply use the program as a training tool to practice their realism.

References

[1] Kodjabachian, Jérôme, and J-A. Meyer. "Evolution and development of neural controllers for locomotion, gradient-following, and obstacle-avoidance in artificial insects." *IEEE transactions on neural networks* 9, no. 5 (1998): 796-812.

[2] Peng, Xue Bin, Glen Berseth, KangKang Yin, and Michiel Van De Panne. "Deeploco: Dynamic locomotion skills using hierarchical deep reinforcement learning." *ACM Transactions on Graphics (TOG)* 36, no. 4 (2017): 41.

[3] Yin, KangKang, Kevin Loken, and Michiel Van de Panne. "Simbicon: Simple biped locomotion control." In *ACM Transactions on Graphics (TOG)*, vol. 26, no. 3, p. 105. ACM, 2007.

[4] "Dynamic Animation and Robotics Toolkit." *DART*, Georgia Tech and Carnegie Mellon University, 6 June 2019, https://dartsim.github.io.

[5] Tan, Jie, Karen Liu, and Greg Turk. "Stable proportional-derivative controllers." *IEEE Computer Graphics and Applications* 31, no. 4 (2011): 34-44.