Evolving Motion Controllers for Articulated Robots

David Rosen

May 12, 2008

Abstract

Articulated robots have a much more complex relationship between motor outputs and locomotion than wheel robots have. We can simplify this relationship using an abstraction layer of cyclic motor outputs corresponding to different kinds of locomotion. One way to create these cyclic motor outputs is by evolving a controller network consisting of oscillators, motor output nodes, and weighted edges. We evolved cyclic movements using this method, and found that simulated annealing and integral frequencies led to the fastest fitness growth. The final controllers resulted in fast and efficient movement. By changing the fitness function and adding sensor inputs, this technique could be extended to evolve movements that are robust and safe as well as efficient.

1 Introduction

With wheeled robots, the relationship between motor outputs and locomotion is fairly simple. For example, the Khepera robot (Figure 1, left) has two wheels, one on either side. We can estimate its forwards velocity as the average of the two wheel velocities, and its angular velocity as the difference between them. However, this relationship is much more complicated with an articulated robot. For example, the Aibo (Figure 1, right) is a dog-like robot whose joints have 20 degrees of freedom in total; that is, a complete controller for the Aibo would need to generate 20 output vectors. Unlike a wheeled robot, there is no individual output vector that will result in locomation. If the robot receives any continuous output vector, it will move to the designated pose and just remain still until the output changes.

One way to address this problem is to add an abstraction layer that translates simple output vectors into periodic sequences of motor output. For example, if we have robust Aibo motor sequences that result in moving forwards, turning, and moving backwards, then we could use this abstraction layer to translate Khepera motor output to Aibo motor output, resulting in the same general locomotion. This leads us to the question we explore in this paper: where do we get these periodic movements?

There are a number of different approaches to this problem that have been used for real and simulated robots. The simplest method is to cycle through manually-defined poses and movements using a finite state machine. Another method is to use hand-coded or trained dynamic controllers. The final method, and the one that we explore, is to use genetic algorithms to create periodic movements that produce the most efficient locomotion.

1.1 Finite state machines

To design a walking movement for the Aibo, we could look at how real dogs walk, divide the movement into stages, and then implement it as a finite state machine: a directed graph of movement states. For example, it could move one leg at a time, and move each leg in three strages: bending the knee, rotating the leg forwards, and then extending the knee. This would result in a twelve-stage walking animation that is pretty easy to implement. It is not guaranteed to be robust or efficient, but it will

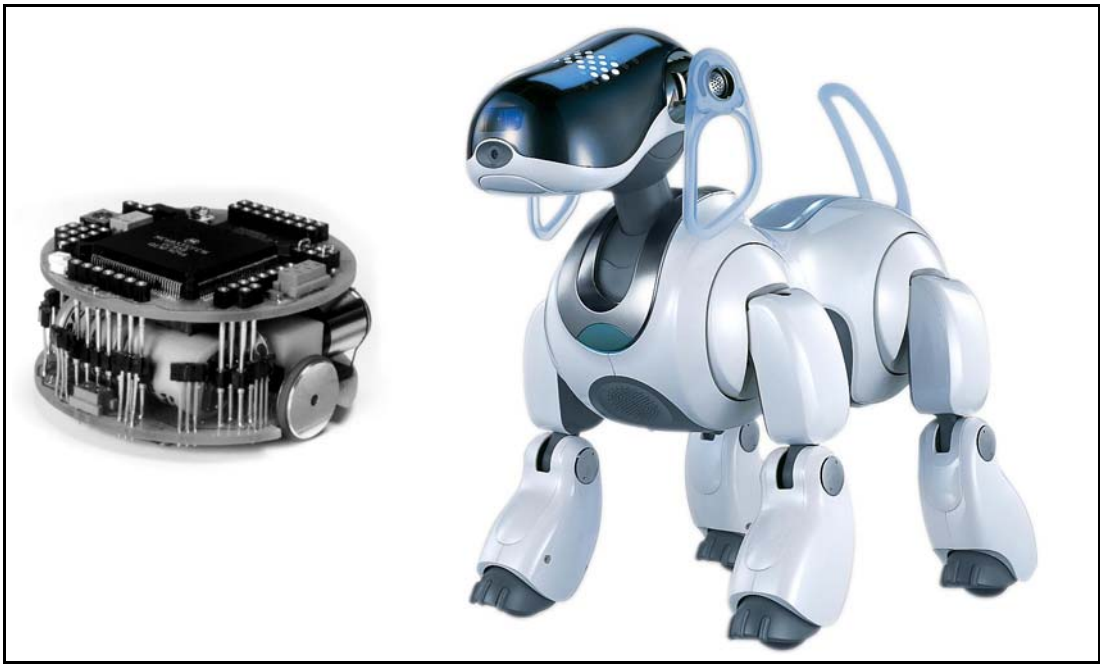

Figure 1: The wheeled Khepera robot(left) and articulated Aibo robot(right).

probably work. From observing the Aibo's movements, it appears that this is how its built-in walking behavior was created.

We can also create other finite state machines for movements such as running and turning, and add transitions between states when they can be performed safely[2]. For example, running might be able to transition to walking in states in which most of the feet are on the ground, and walking could transition to turning when all of the feet are on the ground. With enough periodic movements and transitions between them, the robot could move smoothly in many directions and at many speeds. Because this method offers the designer so much direct control, this technique is often used for 3D character animation. It produces decent results fairly quickly, and will not have to respond to unexpected conditions in a strictly-controlled virtual world.

The advantage of this method is that it is fast and simple. Assuming nothing goes wrong, it should be possible to get an articulated robot to move using this method within a couple hours. However, that is a big assumption. This brings us to the disadvantages of this method: it results in brittle behaviors. That is, if the walking movements are designed on a smooth, level surface, it is likely to fail on a surface that is rough or tilted. It is possible to expand the movement to detect and react to special cases, but it requires a great deal of time and thought to identify and implement a reaction to each additional case.

1.2 Dynamic controllers

A similar approach is to manually code movements that can perform dynamic actions. For example, in a room with many horizontal edges, we could use the Aibo's camera to detect how tilted it is, and use this input to design a movement that maintains an upright position. This kind of dynamic balance controller can then be combined with the finite state machine walking motion to result in a more robust behavior. Balance controllers are often combined with finite state machines to make those brittle behaviors more robust[9]. These kinds of controllers are effective if done correctly, but are difficult to implement. It is not quite as hard in simulation, but it is still mostly an unsolved

problem for biped robots in the real world.

Dynamic controllers can also be combined with finite state machines by including parameters that affect the states in various ways. For example, a swimmer's stroke[8], or a bird's wingbeat[1] can be performed at different angles and speeds to have different effects on the agent's trajectory. A robot can then use machine learning techniques to optimize the parameters for each movement to match a given trajectory. Conversely the robot can analyze motion capture data from real organic movements, and use that information to extract the parameters that result in the most similar movement[4].

Another kind of dynamic controller uses inverse kinematics to move specific parts of a robot to specific points in space [7]. If we know the position of the gripper of a robot arm, and its target position, we can use inverse kinematics to define a series of joint rotations that will bring the gripper to the target. This technique is useful to make robot arms or legs reach specific points, but often we do not have precise information about the spatial coordinates of the robot and its target positions. Also, this technique does not take into account the movement's effect on the dynamics of the rest of the body. For example, we could use inverse kinematics to plan the joint rotations required to move the foot from its current position on the ground to another position, but there is no guarantee that these rotations will not cause the robot to topple over.

1.3 Evolved movement

This last approach is to use genetic algorithms to create controllers. We can represent the controller as a neural network: a graph of sensor input nodes, hidden nodes, and motor output nodes. Now we can define the set of possible hidden nodes and mutations, and evolve a population of controllers to maximize a fitness function. The main advantage of this method is that it can find unexpected solutions that a human designer might not consider, and does not require supervision. On the other hand, it requires many, many trials, and thus may only be practical in simulations. Also, to achieve good results, it is important to choose fitness functions, hidden nodes, and mutations that result in a reasonable search space; a task that is not necessarily intuitive.

Karl Sims used a genetic algorithm to evolve the controllers and morphology of articulated block creatures[6], resulting in a number of interesting behaviors. This inspired us to experiment with evolving cyclic behaviors for robots with fixed morphology. This would allow us to generate locomotion behaviors that are unconstrained by our initial assumptions about how it should move. Similarly, it would greatly reduce the amount of work required to create new kinds of locomotion for special situations; we can just change the fitness function. It still requires some time to evolve, but the evolution requires no human supervision.

Similar research was performed by Larry Gritz, who used a genetic algorithm to evolve controllers for robots with fixed morphology for use in 3D animation[3]. Our approach differs from his in that his controllers were used for discrete actions such as jumping from one point to another or touching a character's hand to a certain point, and were given spatial information that is not available to most robots. On the other hand, our approach is focused on evolving cyclic movement behaviors that do not require any sensor input.

2 Experiment

The goal of this experiment is to learn what kinds of genetic algorithm are most effective at learning efficient cyclic locomotion movements, and how we can constrain the search space to make it easier. We chose to explore the evolution of movement of a humanoid articulated robot because the human body has so many joints with so many degrees of freedom, and thus creates a high-dimensional search space for the genetic algorithm. For this reason, it is a useful problem for testing different evolution parameters and constraints. It would also be difficult to manually design efficient cyclic motions, so it makes sense to approach this task indirectly using genetic algorithms.

There has been a lot of research on simulating or reproducing human walking movement, but we could find none on human locomation via other motions, such as crawling or rolling. It would be interesting to see what movements can be evolved using a humanoid robot without any preconceived information about how humans actually tend to move. When we manually design movements and behaviors for robots, we can forget how different their sensors and motors are from organic creatures, and thus design ineffective behaviors and inefficient movements. By using genetic algorithms, we can avoid most of these problems.

2.1 Simulation

We wrote a 3D robot simulator using C++, using OpenGL for rendering and Open Dynamics Engine (ODE) for physics simulation. The humanoid robot consists of 16 capsules (cylinders capped with spheres at each end), connected with hinge or hinge-2 joints (these work like two perpendicular hinge joints). These joints have motors that can exert torque along any of their degrees of freedom. The robots were subject to a realistic force of gravity, and the environment consisted of a flat plane with friction. Figure 2 shows a rendering of the robot in the simulator.

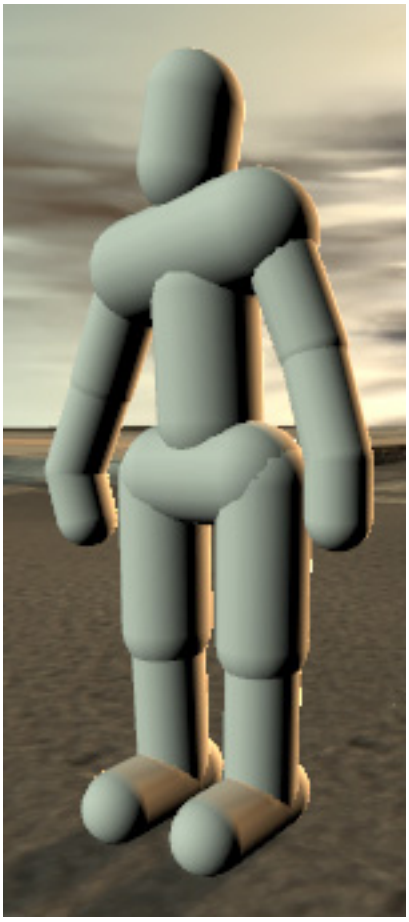

Figure 2: The simulated humanoid robot morphology

2.2 Controller

The controller graph for these robots had no input, only hidden nodes and motor output nodes. There was one motor output for each of the degrees of freedom of the joints, 20 in total (8 hinge joints and 6 hinge-2 joints). The hidden nodes were all oscillators with two parameters: frequency and offset. They were connected to output nodes by weighted edges. The output values were determined by adding together all of the connected hidden nodes multiplied by the weights of their respective edges. For example, if there was one oscillator node with frequency 2 hz, and it had one edge connected to the back-front axis of each shoulder joint, then the robot would flap its arms twice every second.

Possible mutations included changing one of the parameters of a hidden node, changing either end of an edge from one node to another, adding an edge between a hidden node and an output node, changing the weight of an edge, and adding a new hidden node (with a random edge attached). Whenever a graph was mutated, each edge and hidden node parameter had a 20% chance of being changed, and there was a 10% chance each that an edge would change target, that a new hidden node would be created, or that a new edge would be added.

2.3 Evolution

The evolution was always performed on populations of 100 controllers for 100 generations. Each trial lasted for 20 seconds, and then the fitness function was evaluated and stored, and the simulation was reinitialized for the next trial. The 20 survivors of each generation each went on to the next generation, along with four mutated offspring each. There was no crossover mutation. We did not experiment with different evolution parameters, because that could confound our experimentation with the fitness functions and search space constraints.

2.4 Variations

At first the fitness function was simply the distance moved from the starting position. The starting position was defined by the position of the center of mass of the character two seconds into the simulation, to reduce the effect of the initial fall. We soon decided to encourage efficiency by dividing the distance by the amount of energy used (plus a constant to discourage simply lying on the ground for infinite fitness). Next, we decided to implement simulated annealing to help avoid getting stuck at local maxima at first, and then to climb to the top of the global maximum. We did this by creating a temperature value that affected all of the mutation parameters. The temperature value started at 200% (doubling all mutation parameters and magnitudes) and decreased geometrically by 2% every generation, so by the 100th generation it was down to about 25%.

We then experimented with constraining the search space. First, we clamped the oscillator frequencies to the nearest integer so that they would always be multiples of some global movement frequency (in this case 1 Hz), and would thus combine into one global movement cycle. This was inspired by the observation that most movements in nature can be subdivided into discrete cycles such as footsteps, wingbeats, undulations or jumps. Without this restriction, one limb could be moving at 1.47 Hz and the other at 2.23 Hz, and their movement would not combine into an identifiable cycle.

The final restrictions we tried were symmetry and antisymmetry. That is, we would set the torque on the right knee joint to be the same, or the opposite, of the torque on the left knee joint. This restriction was inspired by the observation that most cyclic movements of creatures with bilateral symmetry are either symmetric or antisymmetric. For example, a bird's flight is symmetric, and a human walk or crawl is antisymmetric.

3 Results

The first experiment, with $fitness = distance$, resulted in a behavior that resembled breakdancing (Figure 3). The robots spun, leapt, and squirmed across the plane. The evolution was clearly working, as shown by the increasing fitness (Figure 4), but this movement did not appear to be very efficient. The robots were using a lot more energy (by applying torque at joints) than seemed necessary. Similarly, they would not always move in a straight line; they would sometimes move in a circle, or move in one direction and then back towards the start. This insired us to use a new fitness function: $fitness = \frac{distance}{energy+k}$. Energy was determined by summing the torque applied at each joint every timestep, and k was added to avoid a divide-by-zero error when energy is 0.

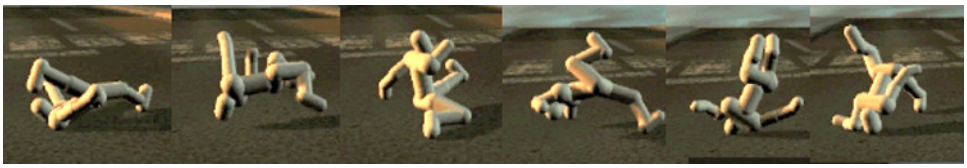

Figure 3: Breakdancing behavior

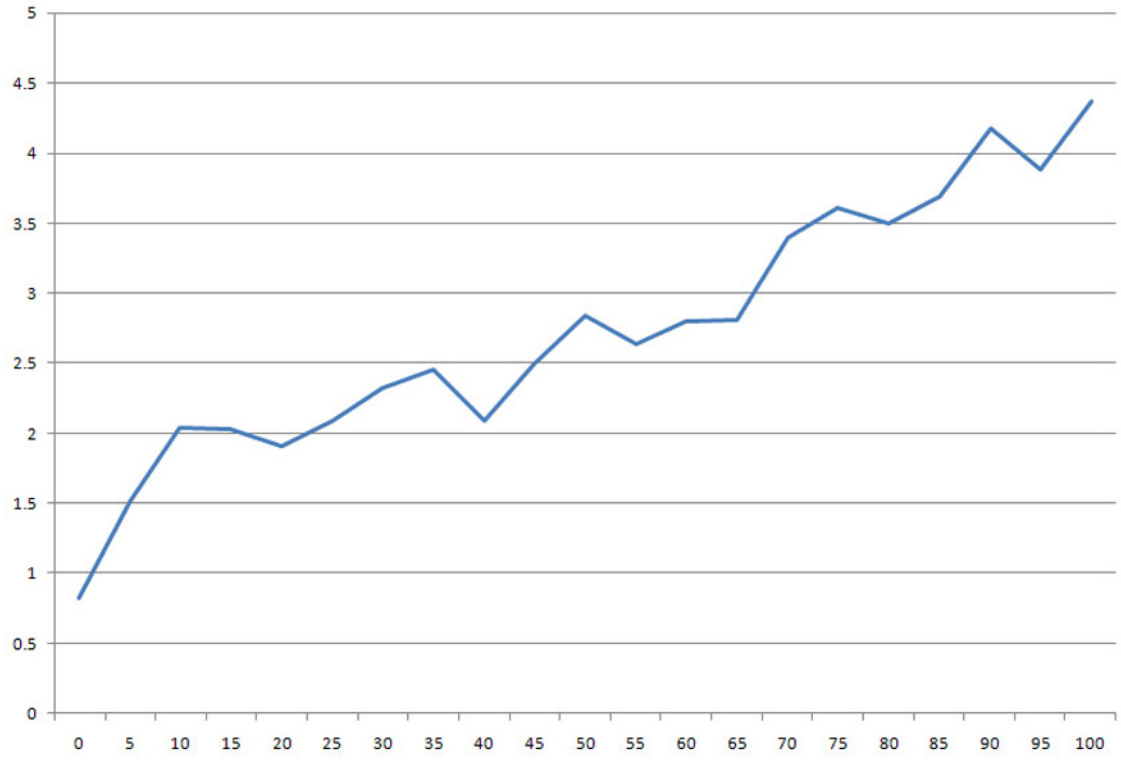

Figure 4: Fitness over generations for three runs with $fitness = distance$

The evolved behavior with the new fitness function looked much more efficient. It was still not a very coherent-looking movement but most of the movement seemed to be propelling the robot along the ground in a crawling motion (Figure 5), and it was much more likely to move in a relatively straight line. Looking at the graph of the average fitness of each generation (Figure 7, orange), it

looked like it may be finding small maxima and then falling off of them because the mutations are too big. For this reason we decided to implement simulated annealing: starting the evolution with large mutations and then decreasing their frequency and magnitude over time. This would help the evolution find a more global maximum and then not fall off of it.

Figure 5: Crawling behavior from first run with $fitness = \frac{distance}{energy+k}$

The simulated annealing seemed to prevent the fitness function from sliding backwards as much (Figure 7, red), with one exception: there is one sharp drop after generation 69 which was apparent in all of the further trials. As we did tests to determine why this is happening, we found that it was not caused by the evolution. When we loaded generation 70 and ran it again, it no longer showed this decrease in fitness. We spent a lot of time trying to figure out why this was happening, and it looks like there may be a bug in the physics engine which caused a change in some of the world parameters after enough objects were created. The drop in fitness therefore represents a sudden change in the environment, rather than a sudden change in the controllers.

The final generation of robots of the first run using simulated annealing lay on their back and kicked in the air as if riding a bicycle, and used the momentum from this kicking to shift their shoulders and move forwards. We noticed that they moved more efficiently when the shoulder movement was synchronized with the leg movement, and decided to try constraining oscillator frequencies to multiples of a fixed frequency to make it more likely for movements to synchronize consistently.

With the new frequency constraint, the evolution proceeded much faster, reaching an end result that was about twice as efficient as before (Figure 7, green). The final movement of the first run was a kind of side crawl. The arm and leg movements were always synchronized, which seemed to be what made it so much more efficient than any of the previous trials. One of the runs gave a result that looked like the crawling behavior from the first run with $fitness = \frac{distance}{energy+k}$, but much more controlled and efficient (Figure 6). On the other hand, the movement was asymmetric, which lead us to our final two constraints: symmetry and anti-symmetry.

Figure 6: One of the behaviors with the integral frequency constraint

The symmetric and anti-symmetric movements (Figure 7, purple, light blue, respectively) were clearly different from the less constrained periodic movements. The first symmetric movement was sort of a backwards hopping movement, and the antisymmetric movement was a scissor kick much like the earlier bicycle kick movement. These constraints seemed to help create clear forwards or backwards movements, but actually resulted in decreased efficiency.

There is a video of all of these movements online at http://www.wolfire.com/evolvedmovement.mov. All of the runs were repeated three times, and resulted in very different movements each time, but the final fitness and shape of the fitness graph were similar between runs of the same algorithm.

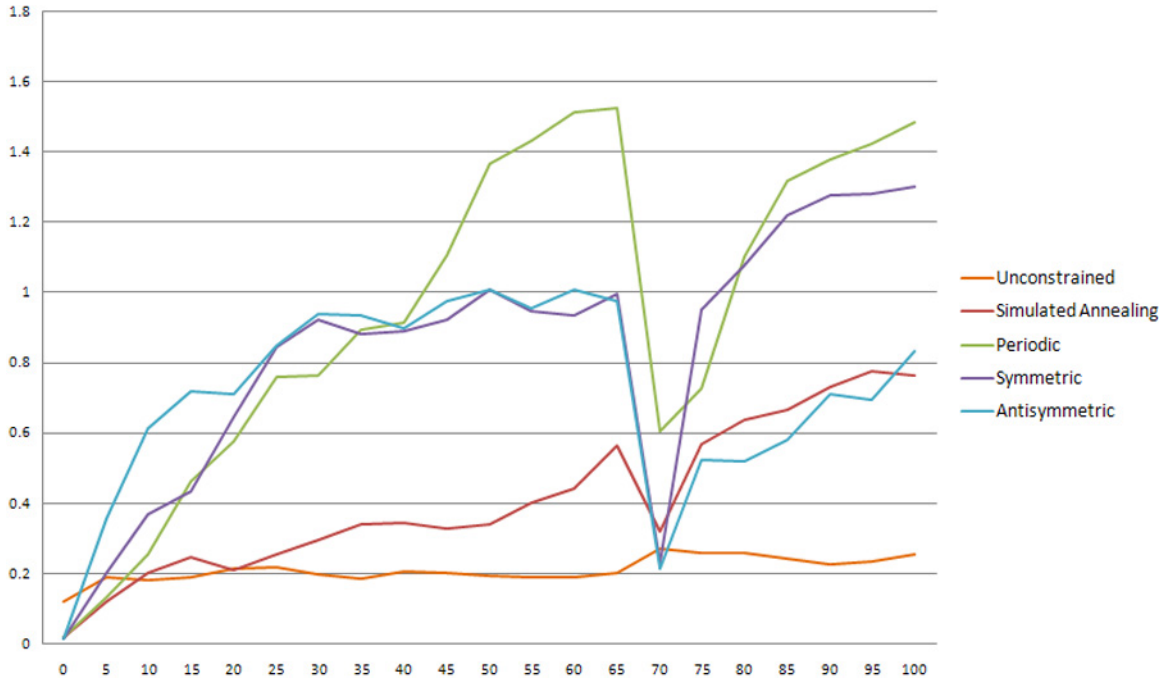

Figure 7: Fitness over generations for three runs of each variation with $fitness = \frac{distance}{energy+k}$

4 Discussion and Future Work

While the evolved movements seemed effective, and mostly possible for a human or humanoid robot to perform, they are probably not ones that we would actually use to move from one place to another. For more human-like movement we would need a more realistic anatomy simulation. Real muscles, and many motors for articulated robots, work by exerting a linear force pulling the points of muscle attachment together, rather than directly applying torque at the joint. Similarly, different muscles vary greatly in strength, while this simulated robot has equal strength at every joint. Also, humans and some walking robots are engineered such that specific motions use very little energy, and can be sustained almost entirely through angular momentum and spring constraints[5].

We would also have to add some consideration of damage to the denominator of the fitness function. Many of the evolved motions would result in injury, involving hard impacts and stress forces to the head and neck, as well as friction burns on the back. By penalizing injury in the fitness function, we could encourage safety as well as efficiency, which would result in movements that would be more useful for real creatures and robots.

To use evolved movements with real robots we would need a very accurate physics simulator. The one that we used gave visually plausible results, but they were not guaranteed to be physically accurate. As with any robot simulator, the learned behaviors will fail if the real environment is very different from the simulated one. This particular application is probably less susceptible to this problem than many because it is not relying on sensor input, but the differences between the simulator and reality can still cause problems.

Finally, this method does not yet create robust movements. It evolves very specific movements that cannot adapt to changes in the environment. However, this approach can give us efficient basic motions to work from and combine with another method, such as a dynamic balance controller. Alternately, we could change the evolution itself to generate more adaptible behaviors. For example, we could include relevant sensor input nodes with edges that amplify or damp the oscillator nodes,

and train the robots on more varied environments.

5 Conclusion

We evolved cyclic movements that allow a simulated, articulated, humanoid robot to locomote across a planar surface. Different runs with the same parameters resulted in very different movements with similar fitness, showing that this technique can be used to evolve a wide range of efficient cyclic movements. By altering the fitness function and oscillator constraints, we could indirectly change the kind of movement that was evolved.

By starting with unconstrained evolution and incrementally adding constraints, we could clearly see the effect of each new constraint. Using simulated annealing greatly improved the learning rate by allowing the robots to avoid getting initially stuck at mediocre solutions, and later to fine-tune the parameters to get closer to better ones. Clamping oscillator frequencies to the nearest integer also improved the learning rate by narrowing the search space to movements that work together more effectively. Symmetric and antisymmetric constraints actually resulted in slightly worse solutions, but they helped direct the motion forwards and backwards rather than sideways or diagonally.

Further work will be necessary to evolve movements that are more human-like, or to evolve movements that can adapt to changes in the environment. However, the movements that we evolved did result in efficient locomotion, and are interesting and varied enough to have some applications in 3D character animation. The fact that we evolved efficient cyclic locomotion movements with such a complex morphology implies that this technique could be a useful tool for locomation in articulated robots in general.

Acknowledgments

Thanks to Professor Lisa Meeden for teaching us about AI and robotics, and Lisa Spitalewitz for helping make this paper as clear as possible.

References

[1] Jia chi Wu and Zoran Popović. Realistic modeling of bird flight animations. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 888–895, New York, NY, USA, 2003. ACM.

[2] Petros Faloutsos, Michiel van de Panne, and Demetri Terzopoulos. Composable controllers for physics-based character animation. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 251–260, New York, NY, USA, 2001. ACM.

[3] Larry Israel Gritz. *Evolutionary controller synthesis for 3-d character animation.* PhD thesis, 1999. Director-James K. Hahn.

[4] C. Karen Liu, Aaron Hertzmann, and Zoran Popović. Learning physics-based motion style with nonlinear inverse optimization. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 1071–1081, New York, NY, USA, 2005. ACM.

[5] Tad McGeer. Passive dynamic walking. *Int. J. Rob. Res.*, 9(2):62–82, 1990.

[6] Karl Sims. Evolving virtual creatures. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 15–22, New York, NY, USA, 1994. ACM.

[7] Katsu Yamane, James J. Kuffner, and Jessica K. Hodgins. Synthesizing animations of human manipulation tasks. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 532–539, New York, NY, USA, 2004. ACM.

[8] Po-Feng Yang, Joe Laszlo, and Karan Singh. Layered dynamic control for interactive character swimming. In *SCA '04: Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 39–47, Aire-la-Ville, Switzerland, Switzerland, 2004. Eurographics Association.

[9] KangKang Yin, Kevin Loken, and Michiel van de Panne. Simbicon: simple biped locomotion control. In *SIGGRAPH '07: ACM SIGGRAPH 2007 papers*, page 105, New York, NY, USA, 2007. ACM.