A comparison of gait learning methods on a quadruped robot

Anonymous Author
Some organization
1234 address
Somewhere
anony@mous.com

Anonymous Author
Some organization
1234 address
Somewhere
anony@mous.com

Anonymous Author
Some organization
1234 address
Somewhere
anony@mous.com

Anonymous Author
Some organization
1234 address
Somewhere
anony@mous.com

Anonymous Author
Some organization
1234 address
Somewhere
anony@mous.com

ABSTRACT

This paper presents an array of approaches to optimizing a quadrupedal gait for forward speed. We implement, test, and compare different learning strategies including uniform and Gaussian random hill climbing, policy gradient reinforcement learning, Nelder-Mead simplex, new predictive methods based on linear and support vector regression, and an evolved neural network (HyperNEAT). We compare results to a baseline random search method. Many of the methods resulted in walks significantly faster than previously hand-tuned gaits.

Edit: Write the abstract

Categories and Subject Descriptors

I.2.6 [**Artificial Intelligence**]: Learning

General Terms

Algorithms, Experimentation, Performance

Keywords

Evolving Quadruped Robotic Gaits, HyperNEAT, Machine Learning, Legged Robots

1. INTRODUCTION AND BACKGROUND

Gaits for walking robots are often designed with some explicit or implicit goal in mind. For some applications, the design criteria may be obvious — perhaps the robot needs to move as quickly or as efficiently as possible — but other times the objective is more complicated, requiring simultaneous optimization of several desired traits, each with its own relative importance. The different combinations of desired traits and the relative weight placed on each can produce drastically different gaits. For example, Honda's Asimo [2] and Boston Dynamic's Big Dog [10] both require gaits that are relatively quick, power efficient, and robust to changing terrain, but they vary widely in the importance placed on each attribute. Big Dog's gait is optimized for much more difficult terrain than Asimo's, resulting in a gait of a completely different form.

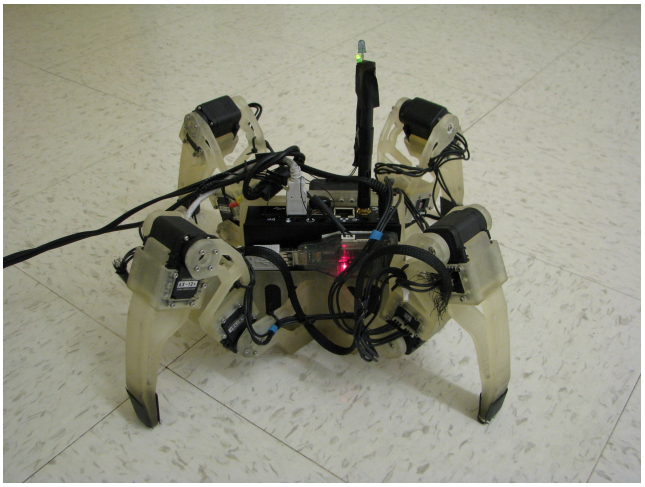

Figure 1: The quadruped robot. The translucent parts were created with 3D-printing technology.

Once the design goals are decided upon, gaits may be obtained by one of several methods. Gaits may be designed manually by an expert, as for Asimo and Big Dog [2, 10]. Alternately, they may be learned through repeated trial and evaluation. Learned gaits offer several advantages over manually designed gaits. Automatically learning gaits can save valuable engineering time and can allow customization of gaits to a particular robot and its unique actuators. Most importantly, in some cases learned gaits can outperform manually designed gaits [8, 17].

In this paper we compare the performance of three different methods of designing gaits: manual design, parametrized

GECCO 2011 Dublin, Ireland

gaits optimized with seven different learning methods, and gaits generated by evolving neural networks with the HyperNEAT generative encoding [14]. While some of these methods, such as HyperNEAT, have been tested in simulation [3, 6], we investigate how they perform when evolving on a physical robot (Figure 1).

Previous work has shown that quadruped gaits perform better when they are *regular* (i.e. when the legs are coordinated) [3, 6, 17]. For example, HyperNEAT produced fast, natural gaits in part because its bias towards regular gaits created coordinated movements that outperformed gaits evolved by an encoding not biased towards regularity [3, 6]. One of the motivations of this paper is to investigate whether any learning method biased towards regularity would perform well at producing quadruped gaits, or whether HyperNEAT's high performance is due to additional factors, such as its abstraction of biological development (described below). We test this hypothesis by comparing HyperNEAT to seven machine learning algorithms biased toward regularity.

A final motivation of this problem is to simply evolve effective gaits for a physical robot. Because HyperNEAT performed well in simulation, it is interesting to test whether it can produce a fast gait for a physical robot. It is additionally interesting to test how more traditional machine learning techniques compete with evolutionary algorithms when evolving in hardware.

1.1 Related Work

Various machine learning techniques have proven to be effective at generating gaits for legged robots. Kohl and Stone presented a policy gradient reinforcement learning approach for generating a fast walk on legged robots[9]. We experiment with this method to create a walk for our robot (called Policy Gradient Descent, described below). Others have evolved gaits for legged robots, producing competitive results [1, 8, 18, 3, 6, 4, 5, 16, 17]. In fact, an evolved gait was used in the first commercially-available version of Sony's AIBO robot [8]. Except for work with HyperNEAT [3, 6, 4, 5], the previous evolutionary approaches have helped evolution 'see' the regularity of the problem by manually decomposing the task. In other words, experimenters have to choose which legs should be coordinated, or otherwise facilitate the coordination of robotic legs. Part of the motivation of this paper is to see if the space of regularities can be explored automatically, which has previously performed well [17], and to perform a direct comparison on the same robot to HyperNEAT.

1.2 Outline of Sections

The remainder of the paper is organized as follows. In Section 2 we define more rigorously the problem of gait learning. Section 3 describes our experimental setup, including the hardware we used and our methods for evaluating fitness of a gait. Section 4 discusses the different gait generation and learning methods we tested, and Section 5 presents and discusses performance results. Finally, Section 6 concludes with possible future extensions to this work.

2. PROBLEM DEFINITION

The gait learning problem aims to find a *gait* that maximizes some performance metric.

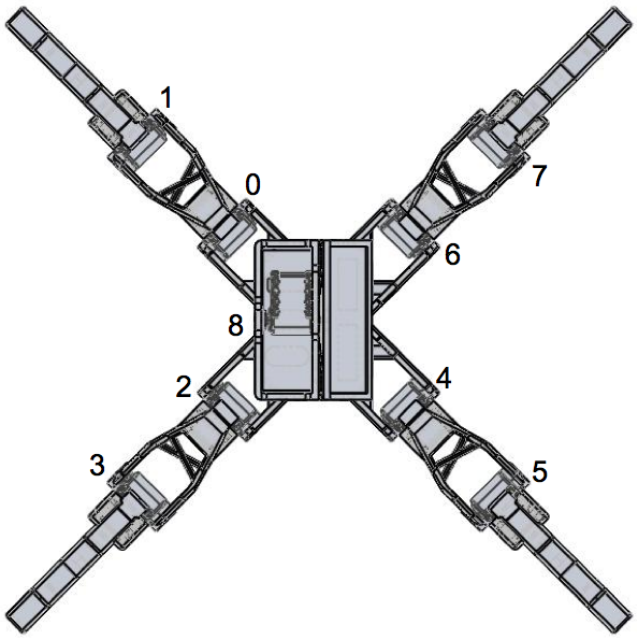

Figure 2: A figure of the robot from a top-down perspective, with servos labeled

Mathematically, we define a gait as a function that specifies a vector of commanded motor positions for a robot over time. We can write gaits without feedback — also called open-loop or feedforward gaits — as

$$\vec{x} = g(t) \tag{1}$$

for commanded position vector $\vec{x}$. The function depends only on time.

It follows that feedforward gaits are deterministic, producing the same command pattern each time they are run. While the commanded positions will be the same from trial to trial, the actual robot motion and measured fitness will vary due to the noisiness of trials in the real world.

For the system evaluated in this paper, we chose to use only feedforward gaits. An interesting extension would be to allow gaits to depend on the measured servo positions, loads, voltage drops, or other quantities.

The ultimate goal was to design gaits that were as fast as possible, so to optimize for speed, our performance metric was displacement over a period of motion lasting approximately 10 seconds. For details of exactly how this displacement was measured, see Section 3.2.

3. EXPERIMENTAL SETUP

Section 3.1 discusses the hardware platform used for the experiments presented in this paper, and Section 3.2 describes the method we used to measure a gait's fitness.

3.1 Platform details

The quadruped robot used in this study was assembled from parts printed on the Objet Connex 500 3-D Printing System. It weights 1.88 kg with the on-board computer and measures approximately 38 centimeters from leg to opposite leg in the crouch position depicted in Figure 1. The robot

Figure 3: A Nintendo Wii remote tracks the location of the robot, providing feedback about distance traveled, in pixels, through an infrared LED mounted on top of the robot.

is actuated by 9 AX-12+ Dynamixel servos: one inner joint and one outer joint servo in each of the four legs, and one servo at the center "hip" joint. This final unique servo allows the two halves of the robot to rotate with respect to each other. Figure 2 shows the positions and numerical designations of all nine servoes.

All of the computation for gait learning, fitness evaluation, and robot control was performed on the compact onboard CompuLab Fit-PC2, running Ubuntu Linux. All gait generation, learning, and fitness evaluation code, except HyperNEAT, is in Python and is available on our website [**?**]. HyperNEAT is written in C++. We controlled the servos with the `pydynamixel` library [7]. The robot connects to a wireless network on boot, which enabled us to control it via SSH.

To track the position of the robot and thus determine gait fitness, we mounted a Nintendo Wii remote on the ceiling and an infrared LED on top of the robot (Figure 3). The Wii remote contains an IR camera that can track and report the position of any IR sources in its image frame. The resolution of the camera was 1024 by 768 pixels, which produced a resolution of 1.7mm per pixel when mounted at a height of ***Edit:*** ??? m.

A separate tracking server, written in Python, ran on the robot and interfaced with the Wii remote via bluetooth using the CWiid library[12]. Our fitness testing code connected to this server via socket and requested position updates at the beginning and end of each run (see Section 3.2).

To run a gait on the robot, a *gait function* is provided, any Python function that accepts a single input — time starting at 0 — and outputs a list of nine commanded positions, one for each servo. To safeguard against limb collision with the robot body, the robot control code cropped the commands to a safe range. This range was [150, 770] for the inner leg servos, [30, 680] for the outer leg servos, and [392, 623] for the center hip servo.

3.2 Fitness Evaluation Details

Edit: Put this in there somewhere:

To be as fair as possible when obtaining the results presented in Section 5, we picked three random points — $\vec{\theta}_A$, $\vec{\theta}_B$, and $\vec{\theta}_C$ — and started all seven methods at these points...

Edit:
proofread. especially middle paragraph

The metric for evaluation of the designed gait was speed. To evaluate a set of parameters, the robot was sent the parameters and instructed to walk for a certain length of time. For each evaluation, the robot always started and ended in the same position in order to measure true displacement and not reward gaits the ended in a lean, since the LED would have moved a different distance the robot. More efficient parameters resulted in a faster gait, which translated into a longer distance walked, measured in pixels, and a better score.

The size of the Wii remote window is approximately 175 x 120 cm. A concern about this is that if the robot walks further than usual, it walks outside the viewable range and thus does not count the run. This possibility could bias the final results reported on the robot.

The only human intervention required during most learning was to occasionally move the robot back into the viewable area of the w

Each of the parameter ??? methods was run on 3 different initial parameter vectors, in order to fairly compare the algorithms. We allowed each run for the parameter ??? methods to continue until the results plateaued (no improvement for one third of the policies seen so far). Three runs of HyperNEAT were completed, each with a different initial seed and run for 20 generations.

4. GAIT GENERATION AND LEARNING

4.1 Parametrized Motion Models

There are a plethora of methods available for the creation of the vector motion generating function $g(t)$ introduced in Section 2, and we've chosen two for comparison in this study. The first method we evaluated was to use a family of parametrized gaits, described in this section and the next, and the second method was to use an evolved non-parametric gait, detailed in Section 4.3.

By a *parametrized gait*, we mean a gait produced by a parametrized function $g(t; \vec{\theta})$. Fixing the parameter $\vec{\theta}$ yields a deterministic motion function over time. We tried several parametrizations on the robot and, upon obtaining reasonable early success, settled on one particular parametrization. We dubbed this the *SineModel5*, as it has five parameters and employs a sine wave as its root pattern.

The five parameters in $\vec{\theta}$ are as follows:

Edit: make sure this text flows into the table

Intuitively, SineModel5 starts with 8 identically sine waves of amplitude α and period τ, multiplies the waves for all inner motors by m_o, multiplies the waves for all front motors by m_F, and multiplies the waves for all left motors by m_R.

Parameter	Description	Range
α	Amplitude	[0, 400]
τ	Period	[.5, 8]
m_O	Outer motor multiplier	[-2, 2]
m_F	Front motor multiplier	[-1, 1]
m_R	Right motor multiplier	[-1, 1]

Table 1: The parameters used for the *SineModel5* motion model.

To obtain the actual motor position commands, these waves are offset by appropriate fixed constants so that the base position (when the sine waves are at 0) is approximately a crouch (the position shown in Figure 1). Finally, in this motion model, the ninth "hip" motor is kept at a neutral position. Thus, the commanded position for all motors, as a vector function of time, is:

$$\vec{g}(t) = \begin{bmatrix} \alpha \cdot \sin(2\pi t/\tau) \cdot m_F & +C_I \\ \alpha \cdot \sin(2\pi t/\tau) \cdot m_O \cdot m_F & +C_O \\ \alpha \cdot \sin(2\pi t/\tau) & +C_I \\ \alpha \cdot \sin(2\pi t/\tau) \cdot m_O & +C_O \\ \alpha \cdot \sin(2\pi t/\tau) \cdot m_R & +C_I \\ \alpha \cdot \sin(2\pi t/\tau) \cdot m_O \cdot m_R & +C_O \\ \alpha \cdot \sin(2\pi t/\tau) \cdot m_F \cdot m_R & +C_I \\ \alpha \cdot \sin(2\pi t/\tau) \cdot m_O \cdot m_F \cdot m_R & +C_O \\ 0 & +C_C \end{bmatrix}$$

Here the fixed constants that determine the crouched position are C_I for inner motors, C_O for outer motors, and C_C for the center hip motor. The nine subscripted functions correspond to the nine labeled motors in Figure 2.

4.2 Learning Methods for Parametrized Motion Models

Using the SineModel5 parametrized motion model described in the previous section, along with the allowable ranges for each of the five parameters (shown in Table 1), the task becomes how to choose the combination of five parameters that results in the fastest motion, per the evaluation methods in Section 3.2.

If we choose a value for the five dimensional parameter $\vec{\theta}$, then a given physical trial gives us one measurement of the fitness of that parameter vector, or $f(\vec{\theta})$. Two things make learning difficult. First, each evaluation of $f(\vec{\theta})$ is expensive, taking 15-20 seconds of time on average. Second, the fitness returned by such evaluations has proved to be very noisy, with the standard deviation of the noise often being roughly equivalent to the size of the measurement.

For both of these reasons, an intelligent *learning algorithm* is needed. In this context, a learning algorithm is a method for choosing the next value of $\vec{\theta}$ to try, given a list of the $\vec{\theta}$ values tried so far and the fitness $f(\vec{\theta})$ measured for each.

We evaluated a total of seven subtly or not so subtly different learning methods for the parametrized motion models. All methods needed a way of picking a $\vec{\theta}$ for the first trial. In general this is done by selecting a random starting vector within the allowable domain. However, as discussed in Section 3.2, we decided to evaluate the different methods starting at one of several common starting points to reduce one possible contribution of noise.

The seven methods used, and their methods for choosing the next $\vec{\theta}$ are:

- *Random*: This method randomly generates parameter vectors in the allowable range for every trial. This strategy was used only as baseline for comparison.
- *Uniform random hill climbing*: This method begins by selecting a single random parameter vector. Subsequent iterations generate a neighbor by randomly choosing one parameter to adjust and replacing it with a new value chosen with uniform probability in the allowable range for that parameter. The neighbor is evaluated by running the robot with the newly chosen parameters. If this neighbor results in a longer distance walked than the previous best gait, it is saved as the new best gait. The process is then repeated, always starting with the best gait.
- *Gaussian random hill climbing*: This method works similarly to Uniform random hill climbing, except neighbors are generated by adding random Gaussian noise to the current best gait. This results in all parameters being changed at once, but the resulting vector is always fairly close to the previous best gait. We used independently selected noise in each dimension, scaled such that the standard deviation of the noise was 5% of the range of that dimension.
- *N-dimensional policy gradient descent*: In [9], Kohl and Stone document a method for local gradient ascent for gait learning with noisy fitness evaluations, and we have included this method in our evaluation. This strategy explicitly estimates the gradient for the objective function. It does this by first evaluating n randomly generated parameter vectors near the initial vector, each dimension of these vectors being perturbed by either $-\epsilon$, 0, or ϵ. Then, for each dimension, it groups vectors into three groups: $-\epsilon$, 0, and ϵ. The gradient along this dimension is then estimated as the average score for the ϵ group minus the average score for the $-\epsilon$ group. Finally, the method creates a new vector by changing all parameters by a fixed-size step in the direction of the gradient.
- *Nelder-Mead simplex method*: The Nelder-Mead simplex method [11] creates an initial simplex with $d+1$ vertices, where d is the dimension of the parameter space. The initial parameter vector is stored as the first vertex and the other five vertices are created by adding to one dimension at a time one tenth of the allowable range for that parameter. It then tests the fitness of each vertex and based on these fitnesses, it reflects the worst point over the centroid in an attempt to improve it. In general, the worst vertex is reflected; however, to prevent cycles and becoming stuck in local minima, several other rules are used. If the reflected point is better than the second worst point and worse than the best point, then the reflected point replaces the worst. If the reflected point is better than the best point, the simplex is expanded in the direction of the reflected point. The better of the reflected and the expanded point replaces the worst point. If the reflected point is worse than the second worst point, then the simplex is contracted away from the reflected point. If

the contracted point is better than the reflected point, the contracted point replaces the worst point. If the contracted point is worse than the reflected point, the entire simplex is shrunk [11].

- *Linear regression*: To initialize, this method chooses and evaluates five random parameter vectors. It then fits a linear model from parameter vector to fitness. In a loop, the method chooses and evaluates a new parameter vector generated by taking a fixed-size step in the direction of the gradient for each parameter, and fits a new linear model to all vectors evaluated so far, choosing the model to minimize the sum of squared errors.

- *SVM regression*: Similarly to linear regression, this model starts with several random vectors, but this time they are chosen in a small neighborhood about some initial random vector. These vectors (generally 8) are evaluated, and a support vector regression model is fit to the observed fitnesses. To choose the next vector for evaluation, we randomly generate some number (typically 100) of vectors in the neighborhood of the best observed gait, and select for evaluation the vector with the best predicted performance. We suspected that if we always chose the best predicted point out of 100, we may end up progressing along a narrow subspace, prohibiting learning of the true local fitness function. Put another way, we would always choose exploitation of knowledge vs. exploration of the space. To address this concern, we added a parameter dubbed `bumpBy` that added noise to the final selected point before it was evaluated.

  Such a method naturally has many tunable parameters, and we endeavored to select these parameters by tuning the method in simulation. To estimate the performance of the algorithm, we ran it against a simulation with a known optimum. The simulated function was in the same five dimensional parameter space, and simply returned a fitness determined as the height of a Gaussian with a random mean. The width of the Gaussian in each dimension was 20% of the range of each dimension, and the maximum value at the peak was 100. Figure 11 shows the learning results on this simulated model using the ultimately selected SVM parameters. Interestingly, a non-zero value of `bumpBy` resulted in better learning than noise free (exploration free) learning.

  Edit: JMC: This next paragraph sounds like it belongs in Results?

  Ultimately, however, the version of SVM tuned for simulation still did not show competitive performance on the real robot. We tried tuning some parameters on the real robot, but after some amount of tuning, the method still exhibited too little exploration and easily became stuck in local minima.

4.3 HyperNEAT Motion Model

Neuroevolution (Evolving Artificial Neural Networks with HyperNEAT):

HyperNEAT is an indirect encoding for evolving artificial neural networks (ANNs) that is inspired by the way natural organisms develop [14]. It evolves Compositional Pattern Producing Networks (CPPNs) [13], each of which is a genome that encodes an ANN phenotype [14]. Each CPPN is itself a directed graph, where each node is a mathematical function, such as sine or Gaussian. The nature of these functions can facilitate the evolution of properties such as symmetry (e.g., an absolute value or Gaussian function) and repetition (e.g., a sine function) [14, 13]. The signal on each link in the CPPN is multiplied by that link's weight, which can alter its effect.

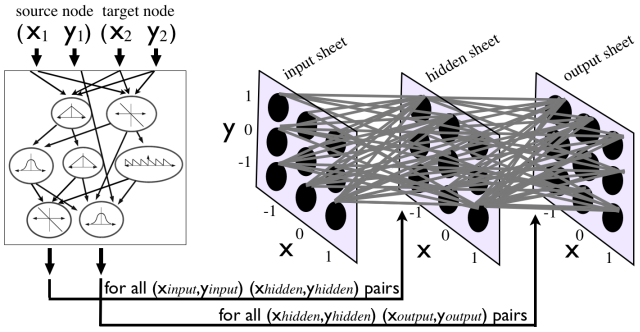

Figure 4: HyperNEAT Produces ANNs from CPPNs. ANN weights are specified as a function of the geometric coordinates of the source node and the target node for each connection. The coordinates of these nodes and a constant bias are iteratively passed to the CPPN to determine each connection weight. The CPPN has two output values, which specify the weights for each connection layer as shown.

A CPPN is queried once for each link in the ANN substrate to determine that link's weight. The inputs to the CPPN are the Cartesian coordinates of both the source (e.g., $x = 2$, $y = 4$) and target (e.g., $x = 3$, $y = 5$) nodes of a link and a constant bias value. The CPPN takes these five values as inputs and produces one or two output values, depending on the substrate topology. If there is no hidden layer in the substrate, the single output is the weight of the link between a source node on the input layer and a target node on the output layer. If there is a hidden layer, the first output value determines the weight of the link between the associated input (source) and hidden layer (target) nodes, and the second output value determines the weight of the link between the associated hidden (source) and output (target) layer nodes. All pairwise combinations of source and target nodes are iteratively passed as inputs to a CPPN to determine the weight of each substrate link.

HyperNEAT is capable of exploiting the geometry of a problem: because the link values between nodes in the final ANN substrate are a function of the geometric positions of those nodes, HyperNEAT can exploit such information when solving a problem [14, 5, 6]. In the case of quadruped locomotion, this property helped Hyper-NEAT produce gaits with front-back, left-right, and four-way symmetries [3, 6].

The evolution of the population of CPPNs occurs according to the principles of the NeuroEvolution of Augmenting Topologies (NEAT) algorithm [15], which was originally designed to evolve ANNs. NEAT can be fruitfully applied to CPPNs because of their structural similarity to ANNs. For example, mutations can add a node, and thus a function, to a CPPN graph, or change its link weights. The NEAT algorithm is unique in three main ways [15]. Initially, it starts

with small genomes that encode simple networks and slowly complexifies them via mutations that add nodes and links to the network, enabling the algorithm to evolve the network topology in addition to its weights. Secondly, NEAT has a fitness sharing mechanism that preserves diversity in the system and gives time for new innovations to be tuned by evolution before competing them against more adapted rivals. Finally, NEAT tracks historical information to perform intelligent crossover while avoiding the need for expensive topological analysis. A full explanation of NEAT can be found in [15].

The ANN configuration follows previous studies that evolved quadruped gaits with HyperNEAT in simulation [6, 3], but was adapted to accommodate differences in the physical robot. Specifically, the ANNs consists of three 3×4 Cartesian grids of nodes forming input, hidden, and output layers. Adjacent layers were completely connected, meaning that there were $(3 \times 4)^2 = 288$ links in each substrate. The inputs to the substrate were the current angles of each of the 9 joints of the robot (described below) and a sine and cosine wave (to facilitate the production of periodic behaviors). The sine and cosine waves were the following function of time (t) in milli-seconds: $\sin(t/12)\pi$. This function produces numbers in the range $[-\pi, \pi]$, which were then JASON FILL ME IN TO DESCRIBE HOW JOINT COMMANDS WERE SPECIFIED.

Edit:
are we mentioning that we gave HyperNEAT the center joint here or earlier?

Edit: jason: edit this section to reflect the post-processing...

The population size for HyperNEAT was 9 and runs lasted 20 generations. These numbers are small, but were necessarily constrained given how much time it took to conduct evolution directly on a real robot. The remaining parameters were identical to Clune et al. [6].

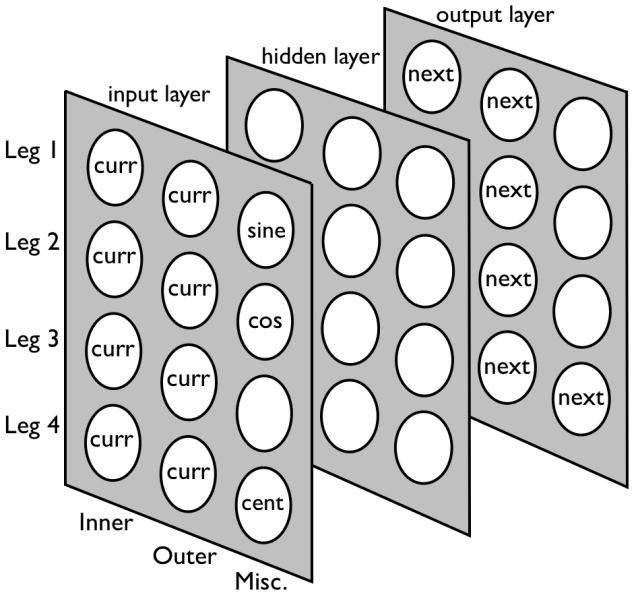

Figure 5: ANN Configuration for HyperNEAT Runs. The first two columns of each row of the input layer receive information about a single leg (the current angle of its two joints). The final column provides a sine and cosine wave to enable periodic movements and the angle of the center joint. Evolution determines the function of the hidden-layer nodes. The nodes in the output layer specify new joint angles for each respective joint. The unlabeled node in the input and output layer is ignored.

5. RESULTS AND DISCUSSION

Edit: proofread this.

5.1 Exploration of Parameter Space

In addition to running strategies to optimize parameterized gaits, we also wanted to investigate the space of possible gaits. To accomplish this, we selected a parameter vector that resulted in motion, but not an exceptional gait, and plotted performance along each dimension individually. In addition, we duplicated each measurement to be able to estimate the measurement noise at each point. Results for this exploration are shown in Figure 6 through Figure 10. As is shown in the figures, some dimensions are smoother than others, and some measurements are fairly noisy.

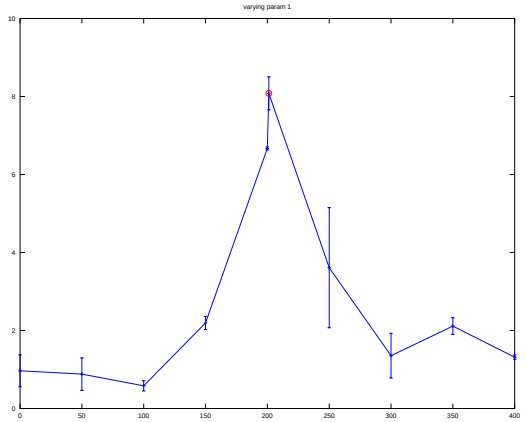

Figure 6: Fitness mean and standard deviation vs. dimension 1. The circle is a common point in Figure 6 through Figure 10

5.2 Maximizing Strategies

The maximizing strategies implemented include uniform random hill climbing, Gaussian random hill climbing, policy gradient descent, and Nelder-Mead simplex. As a group, these strategies produced results similar to gaits randomly generated using the SineModel5 model. None of these strategies consistently outperformed the others in the three trials; A different strategy generated the best gait in each trial. Uniform random hill climbing produced the best gait (11.37 body lengths/minute) in the first trial, with the next best

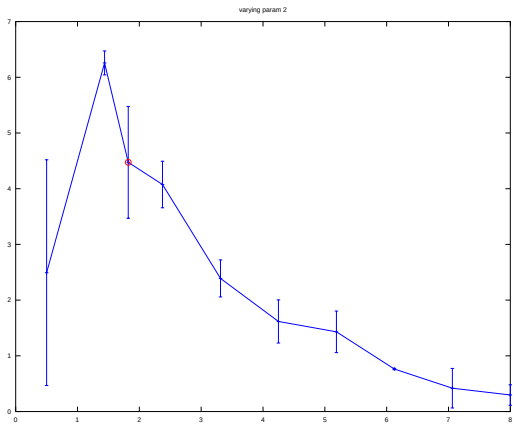

Figure 7: Fitness mean and standard deviation vs. dimension 2. The circle is a common point in Figure 6 through Figure 10

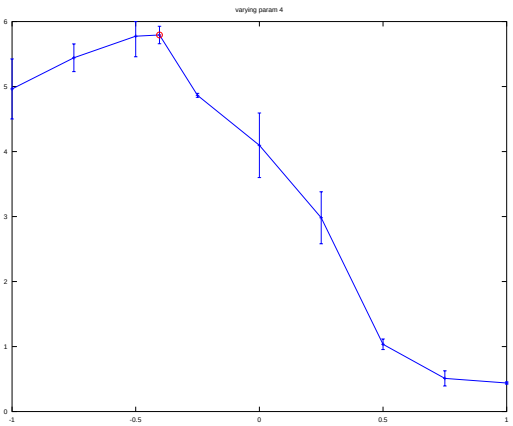

Figure 9: Fitness mean and standard deviation vs. dimension 4. The circle is a common point in Figure 6 through Figure 10

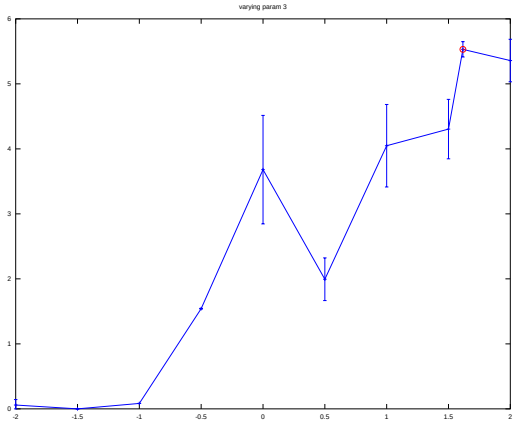

Figure 8: Fitness mean and standard deviation vs. dimension 3. The circle is a common point in Figure 6 through Figure 10

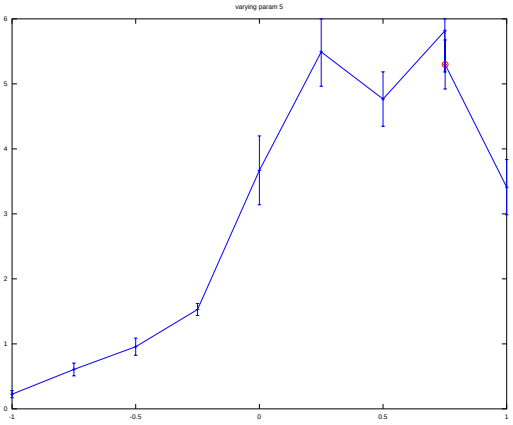

Figure 10: Fitness mean and standard deviation vs. dimension 5. The circle is a common point in Figure 6 through Figure 10

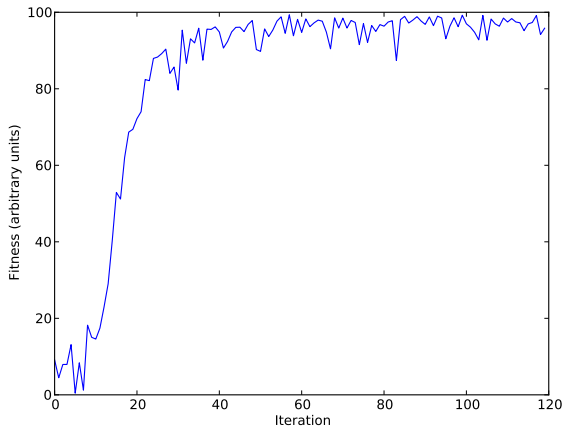

Figure 11: Results for the SVM regression strategy in simulation. This simulation was used to tune the SVM strategy's parameters before trying it on the physical robot. The strategy quickly approaches the maximum simulated fitness of 100.

performing strategy (Nelder-Mead simplex) resulting in a gait of 8.51 body lengths/minute. The best gait in the second trial was actually generated randomly (17.26 body-lengths/minute). Gaussian random hill climbing, policy gradient descent, and Nelder-Mead simplex all produced similar best gaits in the second trial, each around 14 body lengths/minute. In the third trial, Nelder-Mead simplex and Gaussian random hill climbing significantly outperformed the other maximizing strategies, resulting in gaits of 14.83 and 13.40 body lengths/minute respectively. Overall, Nelder-Mead simplex performed the best, producing an average best gait of 12.32 body lengths/minute. Gaussian random hill climbing followed, with an average best gait of 10.03 body lengths/minute.

5.3 Predictive Strategies

The predictive strategies used were linear regression and support vector regression. In the first trial, linear regression produced a gait (27.58 body lengths/minute) that was over three times faster than the best gait produced by a maximizing strategy (uniform random hill climbing – 11.37 body lengths/minute). The average of the best gait for the three trials of linear regression was 14.01 body lengths/minute, which was better than the average best gait for the best performing maximizing strategy (Nelder-Mead simplex – 12.32 body lengths/minute).

As mentioned above, the SVM strategy involved many tunable parameters. The version of SVM tuned for simulation did not show competitive performance on the real robot. We tried tuning some parameters on the real robot, but after some amount of tuning, the method still exhibited too little exploration and easily became stuck in local minima.

Edit: say something about tuning SVM and the sim figure

5.4 Evolved Neural Network Strategy

The evolved neural network strategy used was HyperNEAT. HyperNEAT produced the fastest gaits by far. Linear regression produced a gait in the first trial (27.58 body lengths/minute) that was on par with the best gaits generated by HyperNEAT, but no other maximizing or predictive strategy produced gaits close to those of HyperNEAT. Linear regression was the best performing maximizing or predictive strategy, but it resulted in an average best gait of only 14.01 body lengths/minute. HyperNEAT, on the other hand, resulted in an average best gait of 29.26 body lengths/minute. One issue encountered with HyperNEAT was that it generated gaits that were rougher on the robot than gaits generated by the other strategies. The gaits sometimes caused the motor servos to overheat, which caused the motor to behave differently than expected. Since an overheated motor produces a different walk that expected, a test was performed after each run to make sure all motors were working correctly. If a motor failed, the distance walked would be halved to penalize walks that were harmful to the robot.

Edit: take this out?

We have done complete runs of 3 different initial parameter vectors for random search, uniform random hill climbing, Gaussian random hill climbing, policy gradient descent, Nelder-Mead simplex, and linear regression. We also evaluated the SVM regression and HyperNEAT methods, but these methods were more experimental, and thus we do not yet have the same volume of data for these runs. We developed several gaits that were about 4 times faster than the original hand-coded gait. Results are shown in Table 2.

- For vector A, shown in Figure 12, linear regression worked significantly better than the other algorithms, resulting in a gait (27.58 body lengths/minute) that walked over twice as fast as the next best gait – uniform random hill climbing at 11.37 body lengths/minute – and 5.3 times better than the previous hand-coded gait.

- Vector B, depicted in Figure 13, resulted in similarly performing gaits with all the algorithms. The random method got lucky and produced the best gait (17.26 body lengths/minute) and uniform random hill climbing produced the worst (9.44 body lengths/minute). The remaining algorithms each produced a gait around 13 body lengths/minute.

- For vector C, shown in Figure 14, simplex and Gaussian random hill climbing each produced a gait that substantially outperformed the other algorithms. Simplex resulted in a gait of nearly 15 body lengths/minute and Gaussian random hill climbing produced a gait of just over 13 body lengths/minute, whereas the other algorithms returned gaits of less than 5 body lengths/minute.

Based on the three trials, as shown in Figure 15, linear regression worked the best, followed by simplex. Gaussian random hill climbing performed about as well as random, but it is unclear whether Gaussian would continue to improve and become much better than random search if allowed to run for more iterations. Uniform random hill climbing and policy gradient descent performed similarly to each other on average, but differed greatly on individual runs.

5.5 Discussion

Because only three trials were tested with each algorithm and no algorithm consistently outperformed the others, there

	A	B	C	Average
Previous hand-coded gait	–	–	–	5.16
Random search	6.04	17.26	4.90	9.40
Uniform Random Hill Climbing	11.37	9.44	2.69	7.83
Gaussian Random Hill Climbing	3.10	13.59	13.40	10.03
Policy Gradient Descent	0.68	14.69	3.60	6.32
Nelder-Mead simplex	8.51	13.62	14.83	12.32
Linear Regression	27.58	12.51	1.95	14.01
Evolutionary Neural Network (HyperNEAT)	24.27	36.44	27.07	29.26

Table 2: The best gaits found for each starting vector and algorithm, in body lengths per minute.

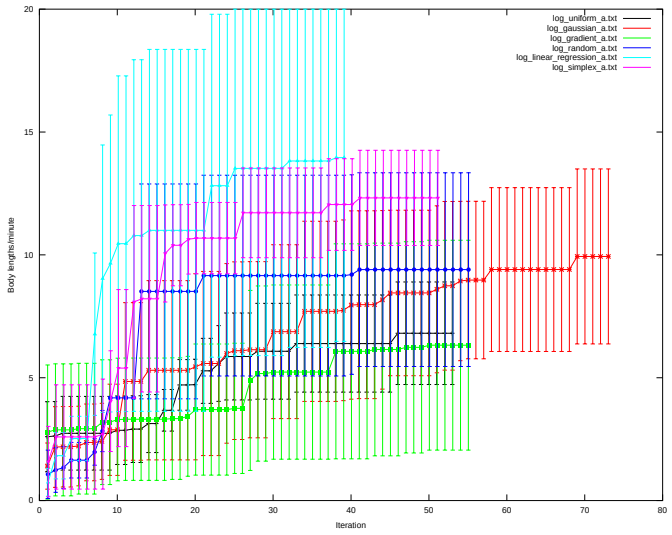

Figure 15: Average results for each method starting with parameter vectors A, B, and C. Error bars are plotted showing the standard error. Linear regression outperformed other methods, followed by simplex. Gaussian random hill climbing performed about as well as random, but it is unclear whether Gaussian would continue to improve and become much better than random search if allowed to run for more iterations. Uniform random hill climbing and policy gradient descent performed similarly to each other on average, but differed greatly on individual runs.

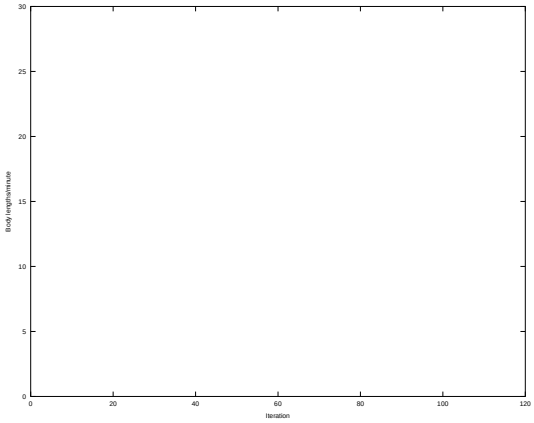

Figure 12: Results for runs beginning with vector A. Linear regression worked significantly better than the other algorithms, resulting in a gait (27.58 body lengths/minute) that walked over twice as fast as the next best gait – uniform random hill climbing at 11.37 body lengths/minute – and 5.3 times better than the previous hand-coded gait.

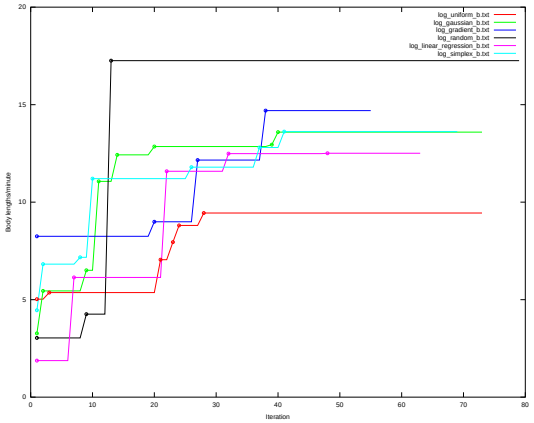

Figure 13: Results for run starting with vector B. All algorithms performed similarly. The random method actually got lucky this time and produced the best gait (17.26 body lengths/minute) and uniform random hill climbing produced the worst (9.44 body lengths/minute). The remaining algorithms each produced a gait around 13 body lengths/minute.

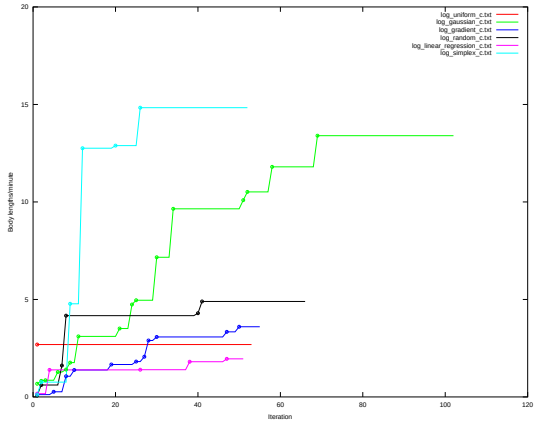

Figure 14: Results for runs starting with vector C. Simplex and Gaussian random hill climbing each produced a gait that substantially outperformed the other algorithms, with around 15 body lengths/minute and 13 body lengths/minute, respectively, whereas the other algorithms returned gaits of less than 5 body lengths/minute.

is a large standard error for each algorithm, as show in Figure 15. For this reason, it is unclear if any algorithm is superior to another in this application. More trials with each algorithm would be necessary to reach a definitive ranking of the performance of the algorithms.

However, each algorithm discovered at least one gait of over 10 body lengths/minute, including random. For this reason, we speculate that the motion model is at least as critical as the learning algorithm used.

6. CONCLUSION AND FUTURE WORK

Edit: write this, old section below

We have presented an array of approaches to optimizing a quadrupedal gait for forward speed. We have implemented and tested different learning strategies, including uniform

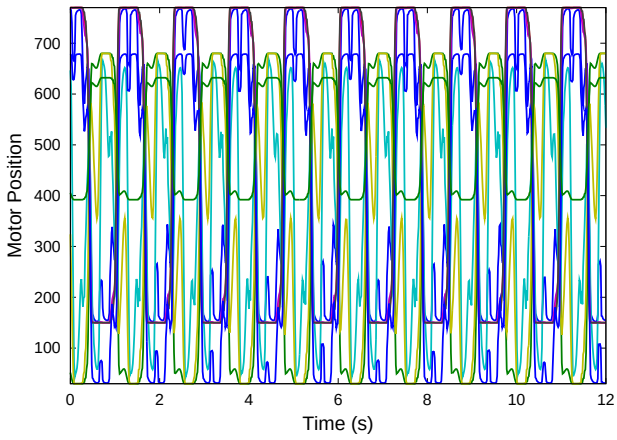

Figure 16: Caption here...???

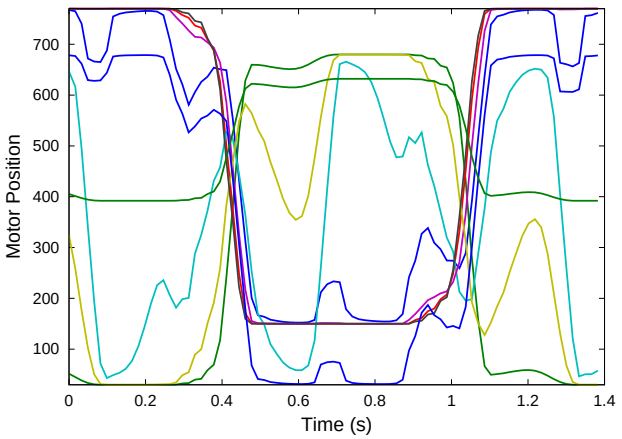

Figure 17: Caption here...???

and Gaussian random hill climbing, policy gradient reinforcement learning, Nelder-Mead simplex, several new predictive methods based on linear and support vector regression, and an evolved neural network (HyperNEAT). We have also compared these approaches to random search as a baseline. Many of the methods resulted in walks significantly faster than previously hand-tuned gaits.

Because only three trials were tested with each algorithm and no algorithm consistently outperformed the others, there was a large standard error for each method, as shown in Figure 15. Thus it was unclear if any algorithm was superior to another in this application. More trials with each algorithm would be necessary to reach a definitive ranking. Because each algorithm discovered at least one gait of over 10 body lengths/minute, including random search, we also conjectured that the motion representation for the robot is more integral to forward speed than the learning algorithm. How to learn the motion representation, in addition to its parameters, remains an open problem.

Edit: write this, old section below

There are several directions in which we could continue our work. First, the error margins for our runs were large, so reducing them by running more trials would lead to more conclusive data. It would also be insightful to run our algorithms on different motion models. We suspect that our choice of motion model influenced the results greatly, as even random choices in the space produced gaits that moved a significant fraction of the time ($> 5\%$). It would be interesting to see how learning methods would perform using a model that included a much higher percentage of unproductive gaits. We also intend to experiment further with SVM regression and evolutionary algorithms/HyperNEAT. Some parameter vectors resulted in the robot turning, as opposed to it walking long distances. This could be an interesting learning goal in future projects. To these ends, we propose the following additions and enhancements:

- More runs and/or longer runs
- Different motion representations
- Better tuning of SVM regression
- Evolutionary algorithms/HyperNEAT
- Learning how to turn

7. ACKNOWLEDGMENTS

Left blank for anonymity.

8. REFERENCES

[1] S. Chernova and M. Veloso. An evolutionary approach to gait learning for four-legged robots. In *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 3, pages 2562–2567. IEEE, 2005.

[2] J. Chestnutt, M. Lau, G. Cheung, J. Kuffner, J. Hodgins, and T. Kanade. Footstep planning for the Honda ASIMO humanoid. In *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, pages 629–634. IEEE, 2006.

[3] J. Clune, B. Beckmann, C. Ofria, and R. Pennock. Evolving coordinated quadruped gaits with the HyperNEAT generative encoding. In *Proceedings of the IEEE Congress on Evolutionary Computation*, pages 2764–2771, 2009.

[4] J. Clune, B. Beckmann, R. Pennock, and C. Ofria. HybrID: A Hybridization of Indirect and Direct Encodings for Evolutionary Computation. In *Proceedings of the European Conference on Artificial Life*, 2009.

[5] J. Clune, C. Ofria, and R. Pennock. The sensitivity of HyperNEAT to different geometric representations of a problem. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 675–682. ACM, 2009.

[6] J. Clune, K. Stanley, R. Pennock, and C. Ofria. On the performance of indirect encoding across the continuum of regularity. *IEEE Transactions on Evolutionary Computation*, 2011 (submitted).

[7] P. Goebel. Pydynamixel. `http://code.google.com/p/pydynamixel/`, 2010.

[8] G. Hornby, S. Takamura, T. Yamamoto, and M. Fujita. Autonomous evolution of dynamic gaits with two quadruped robots. *IEEE Transactions on Robotics*, 21(3):402–410, 2005.

[9] N. Kohl and P. Stone. Policy gradient reinforcement learning for fast quadrupedal locomotion. *IEEE International Conference on Robotics and Automation*, 3:2619–2624, 2004.

[10] M. Raibert, K. Blankespoor, G. Nelson, and R. Playter. Bigdog, the rough-terrain quaduped robot. *Proceedings of the 17th International Federation of Automation Control.(April 2008)*, 2008.

[11] S. Singer and J. Nelder. Nelder-mead algorithm. `http://www.scholarpedia.org/article/Nelder-Mead_algorithm`, 2009.

[12] D. Smith. Cwiid. `http://abstrakraft.org/cwiid/`, 2010.

[13] K. Stanley. Compositional pattern producing networks: A novel abstraction of development. *Genetic Programming and Evolvable Machines*, 8(2):131–162, 2007.

[14] K. Stanley, D. D'Ambrosio, and J. Gauci. A hypercube-based encoding for evolving large-scale neural networks. *Artificial Life*, 15(2):185–212, 2009.

[15] K. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

[16] R. Téllez, C. Angulo, and D. Pardo. Evolving the walking behaviour of a 12 dof quadruped using a distributed neural architecture. *Biologically Inspired Approaches to Advanced Information Technology*, pages 5–19, 2006.

[17] V. Valsalam and R. Miikkulainen. Modular neuroevolution for multilegged locomotion. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 265–272. ACM, 2008.

[18] V. Zykov, J. Bongard, and H. Lipson. Evolving dynamic gaits on a physical robot. *Proceedings of Genetic and Evolutionary Computation Conference, Late Breaking Paper, GECCO*, 4, 2004.