Evolved Neurogenesis and Synaptogenesis for Robotic Control: The L-brain Model

Michael E. Palmer
Department of Biology
Stanford University
Stanford, CA 94305, USA
+1 650-384-0729
mepalmer@charles.stanford.edu

ABSTRACT
We have developed a novel method to "grow" neural networks according to an inherited set of production rules (the genotype), inspired by Lindenmayer systems. In the first phase (neurogenesis), the neurons proliferate in three-dimensional space by cell division, and differentiate in function, according to the production rules. In the second phase (synaptogenesis), axons emerge from the neurons and seek out connection targets. Part of each production rule is an augmented Reverse Polish Notation expression; this permits regulation of the applicable rules, as well as introduction of spatial and temporal context to the developmental process. We connect each network to a (fixed) robotic body with a set of input sensors and muscle actuators. The robot is placed in a physically simulated environment and controlled by its network for a certain time, receiving a fitness score according to its behavior (the phenotype). Mutations are introduced into offspring by making changes to their sets of production rules. This paper introduces the "L-brain" developmental method, and describes our first experiments with it, which produced controllers for robotic "spiders" with the ability to gallop, and to follow a compass heading.

Categories and Subject Descriptors
D.2.2 [**Software Engineering**]: Evolutionary Prototyping; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search – *heuristic methods*; I.2.8 [**Artificial Intelligence**]: Distributed Artificial Intelligence.

General Terms
Algorithms, Experimentation, Theory.

Keywords
Neurogenesis, synaptogenesis, L-system, L-brain, neural networks, robotics, generative, developmental, hexapod.

1. INTRODUCTION
A long-term goal of Evolutionary Computing is to evolve artificial artifacts that rival the complexity of naturally evolved artifacts. We agree with Stanley and Miikkulainen [1] that a sensible approach is the systematic identification and characterization of those features of a developmental process that can produce high evolvability in lineages that employ them. This implies a belief in general developmental mechanisms that aid evolvability in a broad range of contexts, if not in all contexts. Because these presumptive mechanisms are so broadly useful,

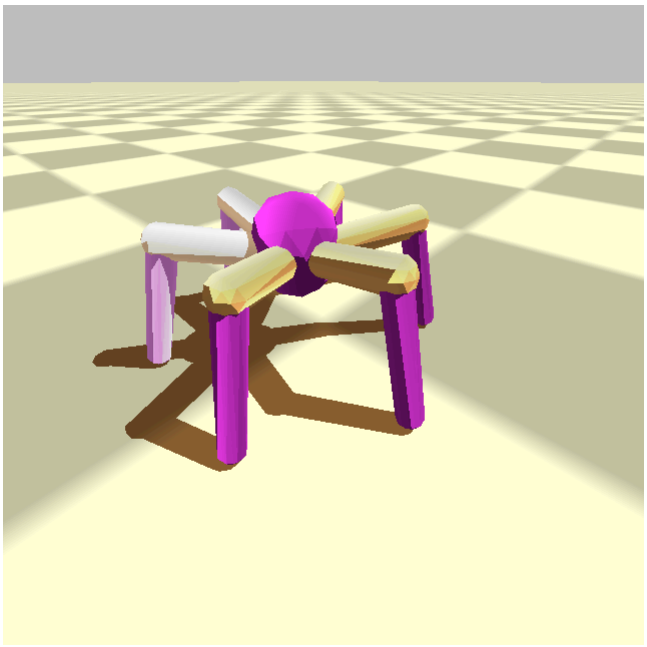

Figure 1: Six-legged "spider" robot body

they will function both in natural and artificial systems; thus we can look to nature for candidates. These mechanisms appear in living things today because they were selected over the long term for improving evolvability in the lineages that carried them (a hypothesis). Here, we introduce an artificial developmental system that intentionally mimics several features of biological development that we hypothesize aid evolvability. We show that the system can produce neural networks that solve complex control tasks in a physically simulated robot. In this introductory paper, due to space constraints, we do not perform "knock-out"

experiments to determine the relative importance of the components, nor do we compare other methods; this will be crucial future work. However, we do situate the system in the context of previous work, as follows.

There is a long heritage of the use of Lindenmayer systems (L-systems) [2] and other grammatical methods to produce neural networks [3-6]. Whereas cellular encoding [7] and edge encoding [8] consider the neural network as an abstract graph, geometrically-oriented methods consider the process of neurogenesis as unfolding in 2D [9] or 3D space. Others have added synaptogenesis situated in 2D [10] or 3D space. Allowing development to unfold in space helps to solve the problem suffered by graph encodings of "node creation-order connectivity bias" identified by Hornby [11]. The L-brain system, introduced here, incorporates neurogenesis and synaptogenesis phases. It was inspired by L-systems, but its production rules unfold directly into three dimensions, like natural neurogenesis and synaptogenesis, without a one-dimensional intermediary string. It is a marriage of grammatical methods, which provide recursive module reuse, with geometrically oriented methods, which provide a spatial and temporal context in which the developmental process unfolds. We introduce this context via conditional expressions that control when and where developmental rules apply.

2. BODY MODEL

Our neural networks will be scored for their ability to properly control a physically simulated robot model. For physical modeling, we use the Bullet open-source physics engine; the hexapod "spider" model is borrowed from one of the Bullet demo programs; see Figure 1. Each leg has three degrees of freedom (DOF): from the center of the head, looking outward along one of the upper leg segments, that segment can move left-right and up-down. The attached lower leg segment can move up-down only. Neither joint can twist. (All constraints are slightly springy.) Each joint axis has fixed limits to its range of motion. When an Output neuron outputs a value of +1, it is calling for its corresponding joint axis to be at its maximum range limit; a value of -1 calls for the minimum range limit. A simulated "spring" between the actual and requested positions generates a force on the joint axis.

3. BRAIN MODEL

3.1 Lindenmayer Systems

The simplest form of L-system consists of an ordered triplet $G = \langle V, \omega, P \rangle$, where: V denotes an alphabet of symbols; ω is a word (string of symbols) called the starting axiom, $\omega \in V+$ (V+ is the set of all strings of one or more symbols from V); and $P \subset V \times V^*$ is a finite set of production rules mapping from a single symbol in V to a word in V* (V* is the set of all strings of zero or more (finite) symbols from V). A production $(a, \chi) \in P$ is written as $a \rightarrow \chi$. The letter $a \in V$ and the word $\chi \in V^*$ are called the predecessor and the successor, respectively. Beginning with the axiom word ω, the production rules are applied to all individual symbols in parallel, replacing each predecessor symbol with a successor word, generating a new (concatenated) word, called the L-string. This rewriting process is iterated to produce a sequence of strings. For example, one might define an axiom word of "a", and define the production rules "a → ab" and "b → ba". The sequence of resulting strings, starting from the axiom word, would then be: a → ab → abba → abbabaab →abbabaabbaababba → ...

When a termination condition is met, the final string is (traditionally) interpreted by a predefined procedure to construct the phenotype. Parametric L-systems [2] include a conditional expression with each production rule: a rule with a matching predicate is applicable only if the conditional evaluates to true.

3.2 The L-brain Model

The "L-brain" model (named in honor of Lindenmayer) produces a connected neural network in two phases that unfold directly into three dimensions: 1) neurogenesis, and 2) synaptogenesis.

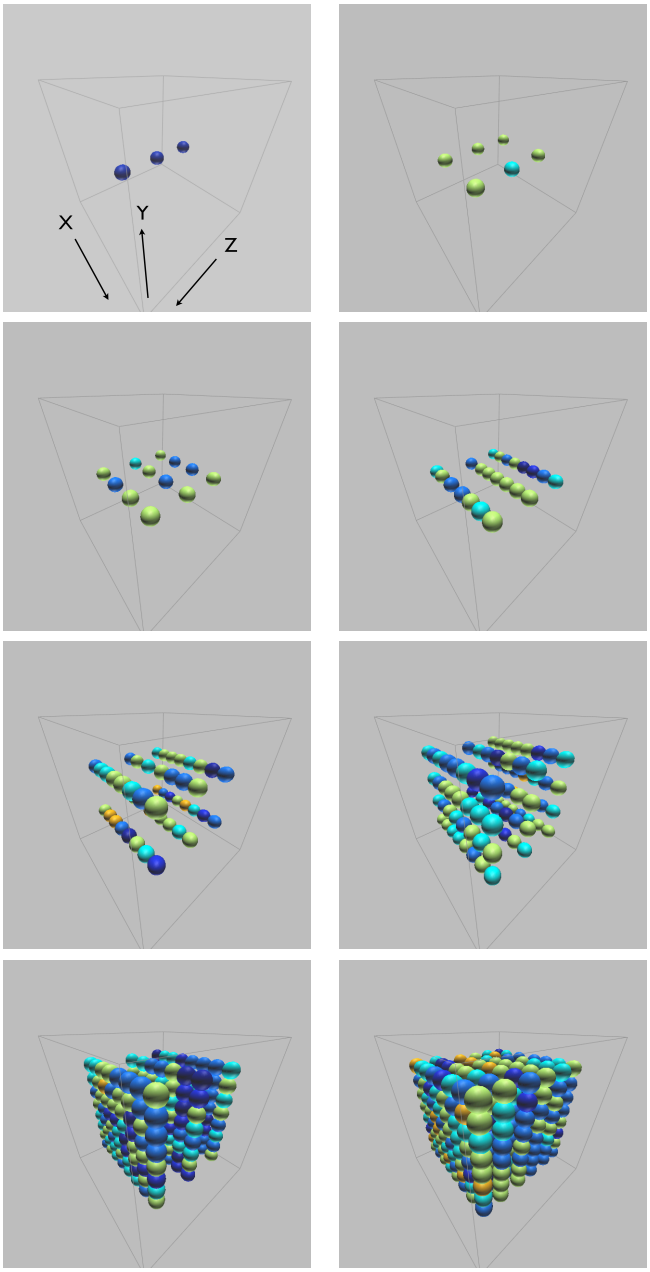

Figure 2: Neuron growth and differentiation; colors represent cell states.

3.2.1 Neurogenesis

In the neurogenesis step, an axiom (initial set) of several cells repeatedly divides in three dimensions. A cell has a three-dimensional position, and an integer type. In an L-brain production rule, the predicate is an integer, and the successor is a pair of integers. At each division of a parent cell, we search for production rules with a predicates the matching parent cell's type, with a conditional that evaluates to true (as detailed below). When a matching rule is selected, the parent is removed and two daughter cells are placed physically on either side of the parent's position in space; each is assigned a type from one of the two integers in the successor. By a process of repeated divisions of all cells, a set of initial cells produces a (usually) larger collection of cells, each with its position and type. At the top left of Figure 2, we have placed 3 "axiom" (initial) neurons along a line parallel to the Z axis; the blue color indicates that we initialized them all to the same cell type. In the second panel (top right), each cell has divided once in the X direction; the colors indicate the cell types of these 6 cells. In the third and fourth panels (second row), two more divisions in X occur. Three divisions in the Y (up) direction follow, and a final division is made in the Z direction. A full complement of 384 neurons has been produced, with varying cell types, as indicated by color.

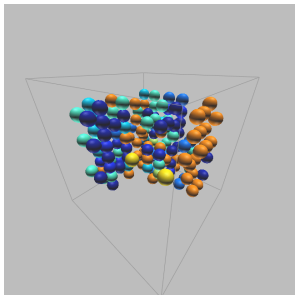

Figure 3: Irregular structures can result if not all cells divide.

If there is no production rule that possesses a matching predicate for a cell's type, or if the conditionals for all such production rules evaluate to false, the cell will not divide. This is a normal part of the process. As illustrated in Figure 3, this is a way to produce irregular structures of fewer than the maximum complement of neurons. On the other hand, if more than one rule is applicable, one is chosen stochastically.

3.2.1.1 The Multi-stack

Part of each production rule is a conditional expression, which is a sequence of tokens defining a Reverse Polish Notation (RPN) arithmetic expression operating on a set of stacks that we call the "multi-stack". Our description of this method, below, may initially seem complex and ad hoc. However, we had two design goals in mind: 1) the expressions should embody a rich and expressive analogy to biological gene regulation, and 2) they should behave "sensibly" under mutation and recombination.

Evaluation of an expression fills the multi-stack with values of several types. We use four independent stacks: a stack of floating point type; a stack of boolean type; and two separate stacks of integer type called the "want-in" and "want-out" stacks. (The latter two are used to compute the connections "desired" by axons during synaptogenesis; they contain integer neuron types.) Importantly, the boolean value on the top of the boolean stack after evaluation of the expression determines whether the conditional evaluates to true or false. Thus, the contents of the expression (in addition to the predicate) control which production rules apply. Special tokens introduce the temporal and spatial context of the cell into the evaluation of the expression, thus influencing development.

3.2.1.2 Token types

The token types are divided into several classes (Operator, Constant, Program Counter, and Special). As each token in the expression is evaluated in sequence, it may affect one or more of the stacks, as outlined in Table 1.

Table 1. Expression tokens and their effects on the multi-stack

	Token type	Pop float	Push float	Pop bool	Push bool	Push want-in	Push want-out
Operator	+, -, *, /	2	1	0	0	0	0
	Enter	1	2	0	0	0	0
	Roll	1	0	0	0	0	0
	Swap	2	2	0	0	0	0
	>, <, >=, <=, ==	2	0	0	1	0	0
	or, and, xor	0	0	2	1	0	0
	not	0	0	1	1	0	0
	enter (bool)	0	0	1	2	0	0
	roll (bool)	0	0	1	0	0	0
	swap (bool)	0	0	2	2	0	0
Constant	float *value*	0	1	0	0	0	0
	bool *value*	0	0	0	1	0	0
	want-in *value*	0	0	0	0	1	0
	want-out *value*	0	0	0	0	0	1
Program Counter	ifjump (jumps to next P.C. token)	0	0	1	0	0	0
	nop	0	0	0	0	0	0
	end (stops)	0	0	0	0	0	0
Special	xpos, ypos, zpos, division-axis, division-count	0	1	0	0	0	0
	is-synaptogenesis	0	0	0	1	0	0

Operator tokens may push or pop values to/from one or more stacks. For example, the "+" operator pops two values from the floating point stack, and pushes their sum back onto the same stack. In contrast, the "or" operator pops two booleans off the boolean stack, and pushes their *or* back onto the same stack. Some operators affect more than one stack: for example, the ">" operator pops two values from the floating point stack, pushing "true" onto the boolean stack if the second is greater than the first, and "false" if not. The "Enter" operator pops a value off the floating point stack, and pushes this value twice. The "Roll" operator simply pops one value from the floating-point stack. The "Swap" operator swaps the top two values on the floating-point stack. The boolean counterparts "enter", "roll", and "swap" apply the same operations to the boolean stack. (A detail: when a value is popped from an empty stack, the floating point stack returns

0.0, the boolean stack returns false, and the want-in and want-out stacks return -1.)

Constant tokens push a floating point, boolean, "want-in", or "want-out" value to the corresponding one of the four stacks.

There are several tokens that affect the Program Counter. The "ifjump" token pops a boolean; if it is true, the Program Counter skips tokens until it comes to one of "ifjump", "nop", or "end". The "end" token stops evaluation. Of course, "nop" has no direct effect on the multi-stack; however, it is a destination point for jumps. Thus the positions of Program Counter tokens in an expression can dramatically affect the final stack contents.

Our first design goal for the RPN expressions was that they permit the spatial and temporal context of the cell to regulate development. The Special tokens directly realize this function. The "xpos", "ypos", and "zpos" tokens push the X, Y, or Z position value of the parent cell onto the floating-point stack. The "division-axis" token pushes a floating-point value of 0, 1, or 2 if the current cell division is in the X, Y, or Z direction. The "division-count" token pushes a floating-point value that increments by one at each cell division, starting from zero. Finally, the "is-synaptogenesis" pushes a boolean value specifying whether development has entered the synaptogenesis phase.

As an example, if we evaluate the following expression,

(6, 5, Swap, Enter, t, f, t, 4, 1, >, and, +, 2, ifjump, 3, 3, nop, 3, end, not)

the floating point stack will subsequently contain: (5, 12, 2, 3); the boolean stack will contain: (t, f); and the ifjump did occur. The want-in and want-out stacks are not involved.

Our second design goal was for the expressions to not be "brittle" under mutation and recombination. We mutate the expressions by inserting, deleting, and replacing tokens. They may also be recombined sensibly (e.g., with crossover at Program Counter tokens), although the experiments shown here do not include sex.

3.2.2 Synaptogenesis

After neurogenesis is complete, the process of synaptogenesis, or the formation of neural connections, proceeds in three steps.

3.2.2.1 Placement of Input and Output Neurons

In the first phase of synaptogenesis, we place a fixed set of Input and Output neurons into the neural field. The 18 red spheres shown in Figure 4 are the Output neurons actuating the spider's 18 DOF. The neurons have abbreviated labels, as follows: the 3 right legs (from front to back on the body) are named R0, R1, R2; the three left legs (front to back) are L0, L1, L2. Each leg has two joints j0, and j1, corresponding to the upper and lower leg segments. Joint j0 has two moveable axes a0, a1 whereas joint j1 has only a0. For example, the actuator to move the upper segment of the front right leg up-and-down is enervated by *R0j0a0*.

The Input neurons are shown in green. The robot has nine inputs: *velX*, *velY*, and *velZ* are the current X, Y, and Z velocities of the spider's "head" in the world coordinates. Input *xFront* indicates the dot product of a vector pointing out the "front" of the spider with the world X axis (i.e., it is a compass that reads +1 when the spider is facing North (positive world X), and -1 for South.) Input *zFront* indicates the dot product of a vector pointing out the front of the spider with the world Z axis. (It reads +1 when the robot faces world East, and -1 for West.) Input *yUp* indicates the dot product of the spider's "up" with the world Y axis (+1 for upright). The rates of change per time step of the latter three inputs are provided as inputs in *xFrontVel*, *zFrontVel*, and *yUpVel* respectively.

All neurons of Input and Output class (and all other neurons) produce output values in the range [-1, 1]. Suitable scaling is applied to the values of angles, etc. relating to the robot body.

3.2.2.2 Assignment of Neuron Classes

After the Input and Output neurons are in place, we return to the set of cells produced in the neurogenesis phase; call these cells the "protoneurons". We perform one more application of the production rules to each protoneuron. (As developmental cues at this time, the token "is-synaptogenesis" returns "true"; "division-axis" and "division-count" return -1.) For each protoneuron, if a production applies, then a single neuron is generated at the protoneuron's position. The type of the neuron is set to the first of the pair of successor values (and the second is ignored). In addition to its integer *type*, the neuron is assigned one of the following *classes*: Sigmoid, Delay, Constant, or Oscillating.

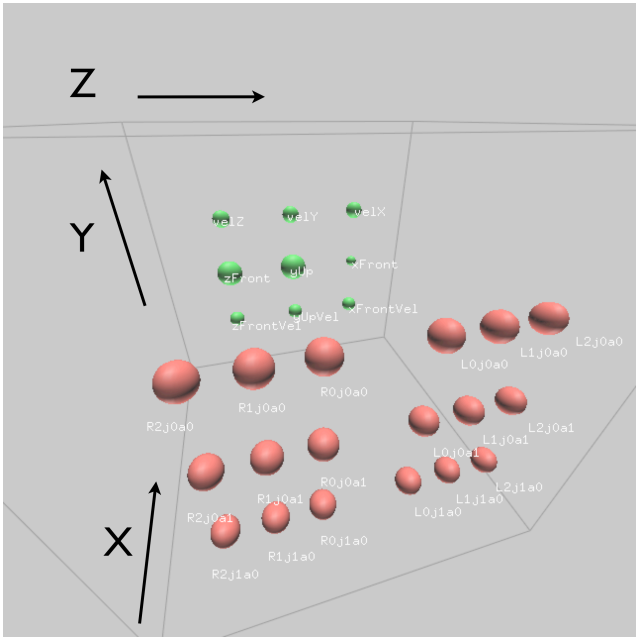

Figure 4: Placement of Input (green) and Output (red) Neurons.

Sigmoid neurons are familiar from the neural network literature: they sum their inputs plus a "bias" value, and apply a sigmoid normalization function to keep output in the range [-1, 1]. When a sigmoid neuron is produced, its *bias* and *a* (which controls the steepness of the sigmoid curve) parameters are required; we will obtain these (as described below) from the multi-stack. In the experiments shown here, all Sigmoid neurons have two inputs available for connection.

Constant neurons produce a constant output value; when initialized they require that value. They have no inputs.

Delay neurons take their input and buffer it for *delay-length* time steps in a FIFO queue, then output it. The queue is initially filled with zeroes. Delay neurons require a *delay-length* parameter. They have one input.

Oscillating neurons oscillate sinusoidally between -1 and 1 over a period of *period* time steps; they require this parameter. They have no inputs.

All the required values for each neuron type above are derived from the contents of the multi-stack after the last evaluation (i.e., when the final neuron type was produced from the protoneuron). Sigmoid neurons obtain their *a* and *bias* parameters from the floating point stack; these are normalized to the range [-1,1] for the bias, and (0, 1] for *a*. Constant neurons obtain their value normalized to the range [-1, 1]. Oscillating neurons obtain their *period* value by popping a sequence of values off the boolean stack, and using these booleans to select one from the following set of possible period values: {90, 60, 30, 20}. (Units are in simulation time steps.) Delay neurons are provided with a *delay-length* in a similar way from the set {5, 10, 20, 40}.

In the experiments described here, we included three neuron *types* of *class* Delay, three types of class Sigmoid, three types of class Oscillating, and none of class Constant. The *class* of a neuron determines its behavior (e.g., an Oscillating neuron oscillates) in the network, as well as the parameters it requires), whereas the *type* of a neuron is used to determine which axons seek to connect to that neuron. Each of the nine Input neurons receives a unique type. The 18 Output neurons each receive one of three types unique to their joint and axis of control (i.e., j0a0, j0a1, j1a0 each receive a unique type; see discussion of Figure 4). Thus, axons connecting to Output neurons must "seek" them (see below) both by their type, and by their position in the neural field.

3.2.2.3 Axon Searching

With all neurons in place, and having assigned the neuron types (and classes), we start to form synaptic connections. Consider a focal neuron. It produces a single axon from its output. That axon investigates each input of all other neurons that are downstream (i.e., toward the Output neurons; the networks are feed-forward) within a certain spatial neighborhood, to see if they are suitable for connection, in the following sense. We turn again to each protoneuron's multi-stack, which is queried for a set of "want-in" values to be assigned to each of a given neuron's inputs, and a single "want-out" value to be assigned to a given neuron's single output (all neuron classes have a single output). As the axons investigate the inputs of the downstream neighbors, we compute what we call the "happy factor" of each potential synaptic connection by adding 1 if the "want-out" of the focal neuron matches the *type* of the downstream neuron, and adding 1 if the "want-in" of the particular input of the downstream neuron matches the focal neuron's *type*. All axons investigating a particular input of a downstream neuron compete by this metric, and the best makes a final synaptic connection (ties are broken by shortest inter-neuron distance).

3.2.2.4 Two unevolved L-brain examples

Figure 5 shows renderings of two example L-brains (generated neural networks). A colored sphere indicates the position in space of each neuron. Input neurons are colored red, Output neurons green, Delay neurons light blue, Sigmoid neurons purple, and Oscillating neurons yellow. (The specific type of a neuron within its class is not rendered.) The size of each sphere indicates the current output value of the neuron: the largest spheres in the renderings indicate the highest possible output value (i.e, a value of 1), and the smallest spheres (still of small positive radius) indicate the lowest possible output value (i.e., a value of -1). Lines indicate the synaptic connections, with the line width indicating the absolute value of the connection weight. Black lines indicate positive weights and red lines indicate negative weights.

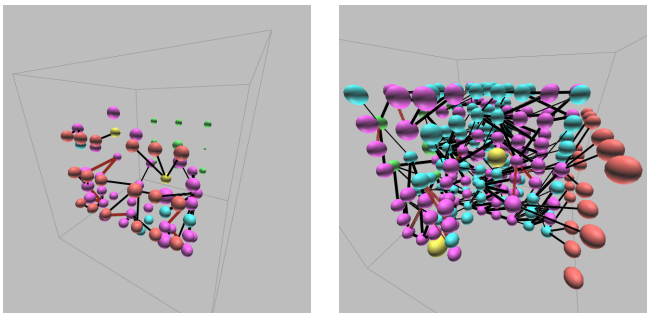

Figure 5: Two unevolved (randomly initiated) L-brains

For each of the two panels, we randomly generated a set of production rules (comprising one genotype), including random expressions, and then used it to produce an L-brain. This process is not necessarily deterministic if there are multiple matching productions at any point; thus each panel of the figure shows one possible phenotype for its genotype. The L-brain at left did produce robot movement: two Oscillating (yellow) neurons can be seen directly connected to several Outputs (red). In the L-brain at right, sequences of neurons including Sigmoid (purple) and Delay (light blue) neurons cascade from the Inputs (green); if we artificially introduce fluctuating signals onto the Inputs, they echo downstream through the Sigmoid and Delay neurons. Neither of these L-brains produces a proper gait.

4. EVOLUTIONARY MODEL

4.1 Mutation

The genotype of an individual is a set of production rules. We initialized all the individuals in the experiments below with 30 random production rules; the conditional expression of each production initially contained 40 random tokens.

Mutation is applied to an individual by considering each production rule in turn, and altering it with probability $\mu = 0.05$ (per rule per individual per generation). A production may be altered by changing its predicate (an integer) or one of its two successors (both integers), or by changing its conditional expression. If the expression is to be altered, tokens may be either: 1) mutated (the token is varied within its token class, e.g., an Operator token is replaced by a new random Operator token), 2) replaced (with a new random token of any class – a more drastic change than a token mutation), 3) added (a new random token is inserted in the expression), 4) deleted, or 5) two tokens may be swapped. It is also possible to add, delete, or clone production rules, but we did not do so in the experiments shown here.

4.2 Population structure

We conducted our evolutionary runs using metapopulations (sets of local populations, or "demes", connected by migration of individuals). We populated each of 16 demes with 50 individuals, with migration rate *mig* = 0.01. We use a metapopulation model because our previous experience [12] suggests that, if the migration rate is low enough, population subdivision protects rare, potentially evolvable genotypes, thereby increasing the rate of adaptation. In this way, a subdivided population effectively exploits a large population size better than a panmictic population.

4.3 Advancing one generation

A generation begins with a set of individuals in each deme. An L-brain phenotype is generated from the genotype of each individual. All 50 phenotypes in a deme are placed together in a physical simulation, randomly distributed in position and orientation in a square area on a large plane, as shown in Figure 6.

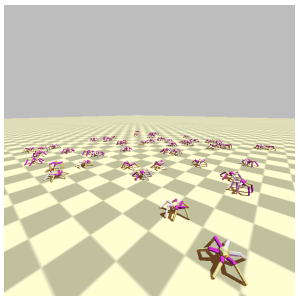

Figure 6: Fifty spiders are initially distributed randomly in a small square area on a large plane.

Each physical (simulated) robot body is enervated by its L-brain (neural network). The simulation is run for a certain time (2000 time steps in the experiments here; at 60 fps this is 33 seconds in simulation world time), and each individual is scored on its performance at a task that is fixed for that experiment (tasks described below). All individuals in a deme reproduce competitively, proportional to their scores, asexually producing 50 offspring individuals for the next generation. (Sexual recombination between sets of production rules is possible, but we did not apply it in the experiments described here.) Mutations are applied. When all demes have been updated, migration among the demes occurs, and a generation is completed.

5. RESULTS

5.1 Scoring

In the experiments below, we employ two scoring functions: one called "Go Far" which rewards the individuals for spending time far from their origin point; and another called "Go North" that rewards individuals for spending time at high absolute value of Z, and especially at positive X ("North") in world coordinates.

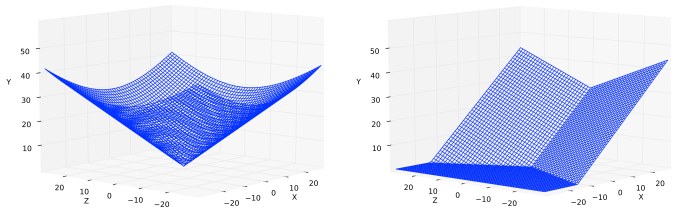

Figure 7: The "Go Far" (left) and "Go North" (right) per-generation scoring functions.

Figure 7 plots the "Go Far" and "Go North" per-generation scoring functions, which are $Y = \text{sqrt}(X^2 + Z^2)$, and $Y = X + 0.5 * \text{sqrt}(Z^2)$, respectively. The Y value corresponding to an individual's X-Z position is accumulated at each generation (starting from an initial score of zero) to produce its final score.

5.2 The "Go Far" task: acquired galloping

At the beginning of an experiment, most of the randomly initialized spiders stand passively with no motion. Some rhythmically tap a foot. Others may writhe disturbingly, or even flip themselves over. (See online video here [13].)

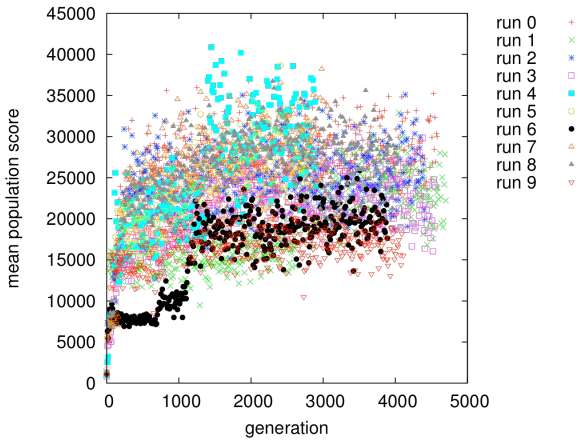

Figure 8: Mean pop. score on "Go Far", ten replicate runs.

However, adaptations soon arise in one deme or another, evolve, and spread to other demes. Figure 8 shows the increase in population mean score in ten independent replicate runs (16 demes of 50 individuals each) as the spiders begin to wiggle purposefully, and, after 1000 or so generations, to gallop.

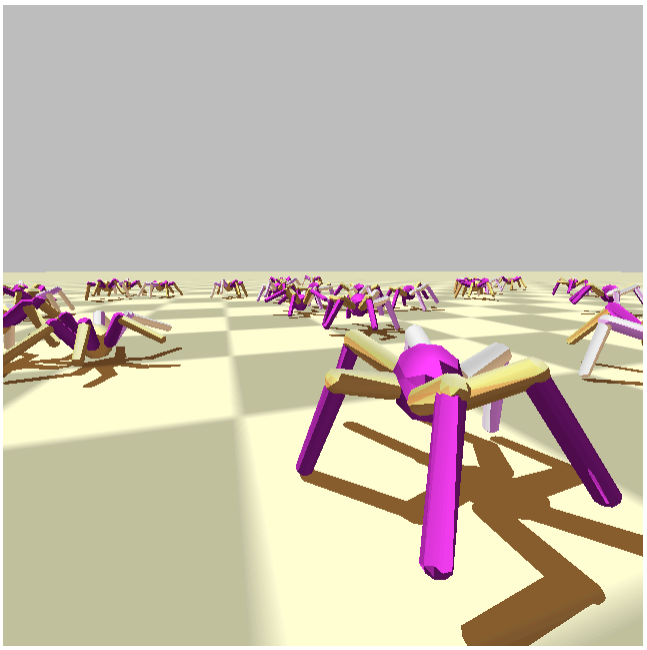

Figure 9: Evolution produces the ability to gallop.

Figure 9 shows a rendering of the physical simulation, with the camera placed in the path of some of the galloping spiders. (See online video here [13].) There were a variety of galloping gaits produced in the replicate runs; however, within a run, the best spiders at a given moment tend to be genetically related, so their gaits are often similar. In the figure, the "front" left and right legs (L0 and R0) are colored white. The closest spider is galloping toward the camera but "backward and sideways". This is unsurprising, since the score does not require "forward" motion.

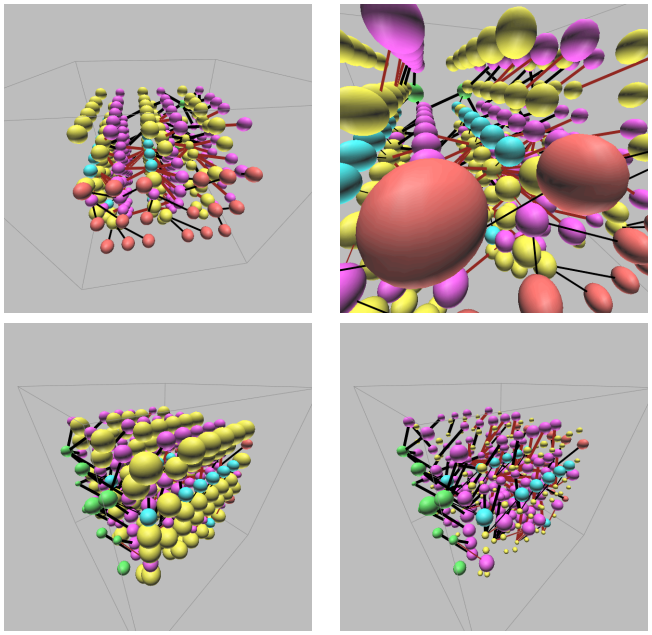

Figure 10: An evolved L-brain that produces a galloping gait.

Figure 10 shows four views of one of the evolved galloping L-brains. In the top left panel, we look up the X axis from the Outputs (red) to the Inputs (green); some regular spatial patterning of Oscillating (yellow), Sigmoid (purple), and Delay (light blue) neurons is apparent. In the top right panel we zoom in closer to the red Outputs; the green Inputs can be seen in the distance.

The two snapshots at the bottom left and right of Figure 10 are taken at different points in time. At bottom left, the yellow Oscillating neurons are large (indicating high output value); at bottom right, they are small (indicating low output value). These were oscillating at a period of about 1 second [13], producing downstream oscillating changes in the network, and ultimately in the outputs, to produce the galloping gait in the spider robot [13].

5.3 The "Go North" task: acquired steering by compass

We noticed that in many of the good solutions to the "Go Far" task, the Input neurons were unconnected. In fact, we could find no evolved "Go Far" brain in which the inputs had any apparent effect on the operation of the network; physically shaking the robots (which changes their sensor input values) produced no apparent change in the outputs. A fixed gait driven by Oscillating neurons, without influence by the inputs, was sufficient to succeed at the "Go Far" task. Thus, we were curious if the L-brains could be evolved to integrate their sensor inputs.

The "Go North" per-generation fitness function (right side of Figure 7) rewards robots for spending time at high absolute values of Z, and high positive values of X. Translation in X receives double the reward as translation in Z, per unit distance. Here, high fitness requires consultation of the input sensors.

Figure 11 shows the increase in population mean score in ten independent replicate runs. The problem appears to be harder than "Go Far", but many replicates still find an excellent solution.

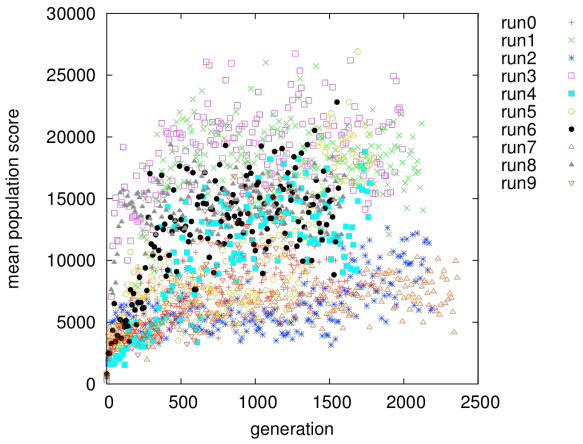

Figure 11: Mean pop. score on "Go North", ten replicates.

In Figure 12, we take one successful population of spiders and compare their initial distribution (left panel) on the plane with their distribution after they have been enervated for ~50 seconds (right panel). They have generally run North ("up" in the image). When the evolved robots are first placed on the plane, they begin to turn themselves to the North; as they face nearly northward, an oscillating galloping motion grows. If the experimenter disturbs a robot by turning its heading away from North, it again goes into turning mode, then begins to gallop again. (See video [13].)

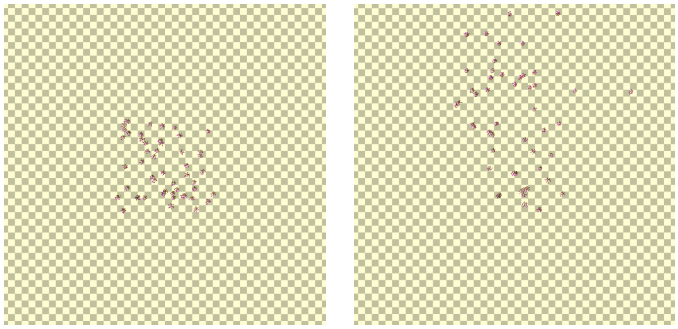

Figure 12: A successful run of "Go North".

Remarkably, one metapopulation found a "Go North" solution with a galloping gait that used no Oscillating neurons. One of the L-brains with this behavior is shown in Figure 13. It produces this behavior by involving the physical simulation in a feedback loop. The loop runs from the *velY* Input, through a chain of Delay neurons up the middle of the figure, connecting in turn to the actuators for the up-down axis of two of the legs (L0 and L1). When these two legs drive down (up), the front of the robot lifts (drops), raising (lowering) *velY*. Thus an oscillating signal propagates around the loop [13], making the robot gallop.

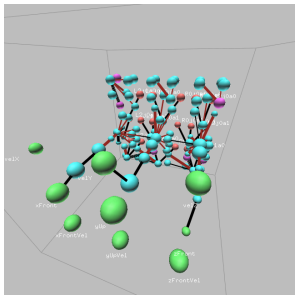

Figure 13: A successful "Go North" L-Brain with no Oscillating neurons.

6. DISCUSSION

A major long-term goal of Evolutionary Computing is to evolve artifacts that rival the complexity of living things. Living things became complex by accretive addition of mechanisms over macroevolutionary timescales. Therefore we wish to understand how to drive the accumulation of complexity in artificial evolution. Two factors will be required: 1) the developmental process must *admit the possibility* of accretive addition of mechanisms over macroevolutionary time periods; and 2) we must understand *how to encourage* (i.e., select for, directly or indirectly) such increase.

Gould [14] (p. 282) argues that selection for complexity was not necessary for living things to become more complex over time. Rather, life necessarily began at the "left wall" of minimal complexity, and "every once in a while, a more complex creature evolves and extends the range of life's diversity in the only available direction." (p. 214) Thus, if our artificial system first admits the possibility of the accretion of complexity, we can expect some increase in complexity due to drift, in the absence of selection for it. However, we may greatly accelerate the process by selecting for complexity, directly or indirectly. Moreover, when we practice artificial evolution, we need not start at the "left wall", i.e., with the simplest system possible. Rather, we can "take a short cut" by borrowing mechanisms (e.g., sexual reproduction, geometrically-oriented development) from living things, rather than evolving them entirely from scratch, if we believe (or, better yet, if we can demonstrate) that they aid the evolvability of complexity.

How do we "select for complexity"? When species are in competition, there will be an indirect pressure to invade new niches, where competitor species cannot follow due to lack of gene flow carrying the innovation that allows invasion of a new niche. (This force is present but weaker in intraspecies competition.) We hypothesize that, in a sufficiently complex environment, organisms with the ability to accrete complexity (which is a type of evolvability) will sometimes be able to invade niches that require unprecedented organismal complexity to exploit, and are thus currently empty. For example, when all life was unicellular, the complex physical world nonetheless admitted empty niches that could be – and eventually were – filled by multicellular organisms. Thus interspecies competition, which implies an indirect selection to diversify, will *in a sufficiently complex environment* also select for the accretion of complexity.

This first introductory paper describes a new platform designed to explore the issues posed above. The L-brain system is intended to "take a short cut" by borrowing mechanisms from biology that will permit further accretion of complexity. The choice of the pseudo-physical robotic control problem was also intentional: it will allow us to provide a sequence of tasks (niches) of increasing difficulty, including both difficult control tasks, as well as competitive and social behaviors. It is our intention to use the complete platform in the future to investigate by "knock out" experiments what components of the L-brain system are most important for the acquisition of complexity.

There are a number of online videos related to this paper [13]. We plan to release our source code in the future [15].

7. ACKNOWLEDGMENTS

We would like to acknowledge NIH grant GM28016 for funding for our computer cluster (160 cores). Thanks to JYLP, CRP, and CWP for much support.

8. REFERENCES

[1] Stanley, K. O. and Miikkulainen, R. A Taxonomy for Artificial Embryogeny. *Artificial Life*, 9, 2 (2003), 93-130.

[2] Prusinkiewicz, P. and Lindenmayer, A. *The Algorithmic Beauty of Plants*. Springer-Verlag, 1996.

[3] Hornby, G. S. and Pollack, J. B. Body-Brain Co-evolution Using L-systems as a Generative Encoding. in *Genetic and Evolutionary Computation Conference* (San Francisco, CA, 2001), 868-875.

[4] Kitano, H. Designing Neural Networks using Genetic Algorithms with Graph Generation System. *Complex Systems*, 4 (1990), 461-476.

[5] Mjolsness, E., Sharp, D. H. and Reinitz, J. A Connectionist Model of Development. *J. theor. Biol.*, 152 (1991), 429-453.

[6] Sims, K. Evolving 3D Morphology and Behavior by Competition. in *Artificial Life IV* (1994). MIT Press, 28-39.

[7] Gruau, F. Genetic synthesis of boolean neural networks with a cell rewriting developmental process. in *International Workshop on Combinations of Genetic Algorithms and Neural Networks* (Baltimore, MD, 1992), 55-74.

[8] Luke, S. and Spector, L. Evolving Graphs and Networks with Edge Encoding: Preliminary Report. in *Late Breaking Papers at the Genetic Programming 1996 Conference* (Stanford,CA, 1996), 117-124.

[9] Kodjabachian, J. and Meyer, J.-A. Evolution and Development of Neural Controllers for Locomotion, Gradient-Following, and Obstacle-Avoidance in Artificial Insects. *IEEE Transactions on Neural Networks*, 9, 5 (1998), 796-812.

[10] Nolfi, S. and Parisi, D. Growing neural networks. in *The Handbook of Brain Theory and Neural Networks* (1992). Addison-Wesley, 431-434.

[11] Hornby, G. Shortcomings with Tree-Structured Edge Encodings for Neural Networks. in *Genetic and Evolutionary Computation Conference* (Seattle, WA, 2004), 495-506.

[12] Palmer, M. E. and Feldman, M. W. Spatial Environmental Variation Can Select for Evolvability. *Evolution* (2011), in press.

[13] Several videos related to this paper are available at: http://www.youtube.com/user/geccospider

[14] Gould, S. J. *The Richness of Life: The Essential Stephen Jay Gould*. W. W. Norton & Company, 2007.

[15] We plan to release our source code in the future. Please check for updates at: http://www.mepalmer.net/geccospider