Quadruped Robot Obstacle Negotiation via Reinforcement Learning

Honglak Lee*†, Yirong Shen‡, Chih-Han Yu††, Gurjeet Singh§ and Andrew Y. Ng*
* *Computer Science Department, Stanford University, Stanford, CA 94305*
† *Department of Applied Physics, Stanford University, Stanford, CA 94305*
‡ *Department of Electrical Engineering, Stanford University, Stanford, CA 94305*
†† *Division of Engineering and Applied Sciences, Harvard University, Cambridge MA 02138*
§ *Scientific Computing and Computational Mathematics Program, Stanford University, Stanford, CA 94305*

Abstract— **Legged robots can, in principle, traverse a large variety of obstacles and terrains. In this paper, we describe a successful application of reinforcement learning to the problem of negotiating obstacles with a quadruped robot. Our algorithm is based on a two-level hierarchical decomposition of the task, in which the high-level controller selects the sequence of foot-placement positions, and the low-level controller generates the continuous motions to move each foot to the specified positions. The high-level controller uses an estimate of the value function to guide its search; this estimate is learned partially from supervised data. The low-level controller is obtained via policy search. We demonstrate that our robot can successfully climb over a variety of obstacles which were not seen at training time.**

I. Introduction

While wheeled vehicles (such as cars and trucks) are very fuel efficient, legged robots can, in principle, climb over much larger obstacles relative to the size of the robot, and thereby access significantly more difficult terrain. In this paper, we apply reinforcement learning to develop a novel controller for a quadruped robot so as to enable it to negotiate a wide variety of obstacles, including ones that had not been previously seen during training.

In related work, some other researchers have focused on clever mechanical design of legs that automatically adapt to terrains [1], [2]. Bai et al. [3] proposed a rule-based algorithm for generating gaits to step onto (and over) small obstacles, and demonstrated them in simulation. There has also been work on adaptive gaits for quadruped robots based on “central pattern generators” [4]–[8]. However, these gaits have mainly been applied to traversing rough surfaces, rather than climbing over large/discrete obstacles. Moore et al.’s hexapod (six-legged) RHex robot is capable of climbing up steps, but does so using ingenious mechanical design in which large, circular, “wheel-like” legs flail at the obstacles, rather than through careful balance and coordination [9], [10]. There are also several robots that rely on hopping to climb up steps, e.g., Raibert’s biped robot [11] and Poulakakis et al.’s galloping robot [12]. Bretl et al.’s vertical climbing robot successfully applied motion planning to robot climbing [13]. Their algorithm relied on computing statically stable poses for vertical climbing, and does not immediately apply to more general obstacle negotiation tasks. Learning algorithms have been successfully applied to a few legged locomotion problems, for example [14]–[16]. But these methods have been demonstrated to work only on flat terrain in which the same (periodic) gait could be repeated without taking into account possible obstacles in the path. Kim et al. [17] proposed an algorithm for walking and climbing in 3d environments, but specifically for a robot whose feet had strong suction pads. Chestnutt et al. [18] used hierarchical planning to deal with obstacle avoidance, but considered only 2D path planning and 3D obstacle avoidance (“stepping” over small obstacles), and not the more difficult task of climbing over obstacles.

Our method is based on hierarchical reinforcement learning [19]–[22] using a two-level hierarchical decomposition of the task. Given an obstacle (such as a step) that the robot needs to climb over, the high-level planner selects a sequence of “target foot placement positions” for the robot, one foot at a time. It is then the low-level controller’s task to move the feet to these targets in order, while keeping the robot balanced and preventing the legs from hitting any obstacles.

The low-level controller operates on a very short temporal scale, in a problem which requires fine coordination and balance, but does not require careful multi-step reasoning. Policy search algorithms [23] with simple parameterized policies apply well to such problems; we use it to learn the low-level controller.

In contrast, the high-level controller has to plan a sequence of foot placement positions, and must reason on significantly longer timescales. We build a high-level controller using value function approximation and beam search. In detail, a value function indicates the relative “desirability” of different states in the problem. We learn a value function using a novel reinforcement learning algorithm, one that also makes use of some supervised information. We then apply beam search with the learned value function to find the sequence of foot placements that maximizes the reward. This sequence is then executed by the low-level controller.

Hierarchical reinforcement learning algorithms have been applied to a number of other problems, for example various grid-world variants (e.g., [22]). To our knowledge, it has not previously been successfully applied to any continuous state-space problems of comparable size and complexity to ours.

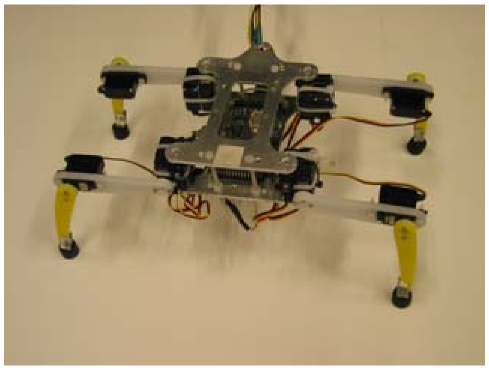


Fig. 1. Quadruped robot

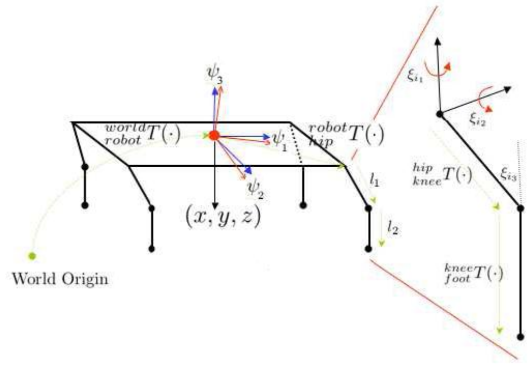


Fig. 2. Quadruped state variables and forward kinematics

II. Quadruped Robot Model

The quadruped robot used in this work is shown in Figure 1. The robot is approximately 10cm tall, 28cm wide, and 12cm long, and weighs 500 grams. Each leg has three servomotors, two at the hip joint for rotating the upper segment of each leg forward/backward or up/down; and one at the knee joint for rotating the lower segment of each leg inward/outward. The servos have a maximum torque 3.0 kg-cm, and may fail to move to the commanded position if the torque is insufficient. The task is to send the right sequence of commands (target angles) to these servos that enable the robot to climb up an obstacle.

The state of the robot is completely determined by its position (x, y, z); orientation (roll ψ_1, pitch ψ_2, yaw ψ_3); the twelve joint angles ξ_1, ... , ξ_{12}, and the corresponding velocities and angular velocities $\dot{x}$, $\dot{y}$, $\dot{z}$, $\dot{\psi}_1$, $\dot{\psi}_2$, $\dot{\psi}_3$, $\dot{\xi}_1$, ..., $\dot{\xi}_{12}$. This gives a 36 dimensional state space. However, our low-level controller will choose commands only as a function of the variables (x, y, z, ψ_1, ψ_2, ψ_3, ξ_1, ... , ξ_{12}), which span an 18 dimensional subspace (cf. state abstraction in reinforcement learning [19]–[22]). To simplify our notation, we will sometimes write these 18d state variables at time t as $\Omega_t = [\omega_1, ..., \omega_{18}]^T$.

Figure 2 shows the kinematic model of our robot and the coordinate transformations used to compute the location of its joints. Standard forward kinematics can be used to calculate all the foot positions given Ω_t [24]. We write foot i's position as follows:

$$u_t^i = ^{world}_{foot} T(x, y, z, \psi_1, \psi_2, \psi_3, \xi_{i_1}, \xi_{i_2}, \xi_{i_3}). \qquad (1)$$

where u_t^i is the foot i's position at time t; and i_1, i_2, i_3 are the indices of the three joints on foot i. The term $^{world}_{foot}T(\cdot)$ is obtained from simple kinematic computations.

III. Low-level controller

The job of the low-level controller is to output a sequence of servo commands that will move a single foot to a given target position while keeping the other 3 feet stationary. Our low-level controller uses a policy (controller) that maps from the current state, the index of the moving foot, and the target position to commands for the 12 servos. We take a policy search approach to learning the low-level controller. Thus, as is standard practice in policy search, we will begin by specifying the policy class (i.e., the parameterization of the low-level controller). The low-level controller has to generate foot motions that, during the motion to the target position, do not hit any of a variety of possible obstacles. Potential fields [25] seem well suited to this task. More precisely, we choose a policy parameterization based on potential fields, and use policy search to automatically learn the potential field parameters.

Potential fields [25] are frequently used in robot motion planning to find a sequence of actions to move towards a goal without violating certain constraints; for example, to generate a path to a goal while avoiding obstacles. Informally, we will place an "attractive potential" at the goal (this potential is represented by a function over the state space that decreases as we approach the goal); and "repulsive potentials" at the obstacles (represented by functions that increase as we approach an obstacle). The overall potential function is the sum of all the attractive and repulsive potentials. On each step, the robot moves by taking the steepest descent direction over the overall potential function.

In detail, we formulate a low-level controller using a potential field composed of three functions: (1) **Goal potential**: an attractive potential field which encourages the robot to move a single leg toward the specific goal position generated by the high-level planner. (2) **Surface potential**: a repulsive potential that keeps the moving foot away from the ground and obstacle surfaces. (3) **Posture Potential**: a potential function that encourages "good" posture (cf. [26]) such as having the center of gravity (CG) within the triangle formed by the three supporting feet and minimizing yaw and roll of the robot body.

For the task of moving foot i from the current position u_t^i to a goal foot placement position u_g^i, we construct the following potential:

$$U_t(\Omega_t, u_t^i, u_g^i) = U_t^g(u_t^i, u_g^i) + U_t^s(u_t^i, u_g^i) + U_t^p(\Omega_t), \qquad (2)$$

where $U_t^g(\cdot)$, $U_t^s(\cdot)$, and $U_t^p(\cdot)$ represent the goal, floor, and posture potential functions respectively. We defer to the Appendix the exact functional forms of these potential functions. To execute a foot motion, at each instant in time the robot computes the negative gradient, evaluated at the current state, of the potential function:

$$\tilde{g}_{t,j} = -\nabla_{\omega_j} U_t(\Omega_t, u_t^i, u_g^i), \quad j = 1, ..., 18 \qquad (3)$$

The robot then tries to change its joint angles in the direction specified by $\tilde{g}_t$.[1]

This idea needs two additional refinements. Naively following the gradient of the potential function will only work if there are no local minima in the potential, other than a global minimum at the goal position. However, near the front of obstacles, it is possible for the attractive goal potential and the repulsive surface potential to cancel, causing local minima to form in the potential function. To deal with this problem, we add a vortex field term [27]; this is a vector field that goes around the obstacles. Our use of the vortex field is motivated by the observation that the moving foot must not only avoid colliding with obstacles, but must also go around obstacles in order to reach the goal position. So, near the front of obstacles where the attractive goal potential and the repulsive surface potentials can nearly cancel each other, the vortex field will cause the moving foot to move upwards over the obstacle. Details on the vortex field can be found in the Appendix.

One additional refinement to the algorithm is necessary to keep the three supporting feet from moving. At every step, we informally think of the robot as trying to change its state in some direction $\hat{g}_t$ (given by the sum of the gradient of U_t and the vortex field vector). In general, following the direction of $\hat{g}_t$ exactly would also change the position of the supporting (non-moving) feet $u_t^{s_1}$, $u_t^{s_2}$, and $u_t^{s_3}$, where s_1, s_2, and s_3 are the indices of the robot's three supporting feet at time t. In order to avoid this (since we want to move only one leg at a time), following [28] we will project $\hat{g}_t$ into the subspace of motions which keeps the positions of the supporting feet constant. More precisely, we define $\Phi_t \in \mathbb{R}^{18\times 9}$ to be the matrix whose columns are the gradients of the components of $u_t^{s_1}$, $u_t^{s_2}$, and $u_t^{s_3}$ with respect to the 18 state variables. The change in the positions of the supporting feet due to a small change in the state variables $\delta\Omega_t$ is approximately $\Phi_t^T \cdot \delta\Omega_t$. Hence, in order to keep the supporting feet stationary, we should only move in directions that are in the nullspace of Φ_t^T. Now, let $\hat{g}_t{}^*$ be the projection of $\hat{g}_t$ onto the nullspace of Φ_t^T. Finding $\hat{g}_t{}^*$ can be posed as the following minimization problem:

$$\begin{array}{ll} \min & ||\hat{g}_t{}^* - \hat{g}_t||_2 \\ \text{s.t.} & \Phi_t^T \cdot \hat{g}_t{}^* = 0 \end{array} \qquad (4)$$

The closed form solution for $\hat{g}_t{}^*$ is:

$$\hat{g}_t{}^* = (I - \Phi_t \cdot (\Phi_t^T \cdot \Phi_t)^{-1} \cdot \Phi_t^T) \cdot \hat{g}_t \qquad (5)$$

Thus, we change the joint angles in the direction of $\hat{g}_t{}^*$.

In choosing the form of the potential function, we were thus able to encode a significant amount of prior knowledge about the task of moving a single foot. Specifically, minimizing the combination of the three potential functions will tend to cause the robot to move the foot toward the goal while maintaining balance and avoiding collisions with obstacles or with the ground. Further, the projection onto the nullspace of Φ_t prevents the supporting feet from moving. All this prior knowledge, encoded into the low-level controller's policy class, helps it to solve the high-dimensional search problem associated with moving a single foot to a new position.[2] That we can encode so much prior knowledge into the policy parameterization is one of the strengths of policy search. However, the policy class still has many free parameters which are difficult to choose by hand. For example, even though it is easy to specify the form of a potential that repulses a foot from an obstacle, it is hard to decide by hand exactly what the *magnitude* of this repulsion should be. In our approach, we learn the parameters of the potential functions automatically, as discussed in the next section.

A. Policy search

The low-level controller has 20 parameters that govern various trade-offs between its attractive and repulsive potentials. To learn these parameters, we began by building a dynamical simulator[3] of the robot following the specifications of the real robot. Using the simulator, we then applied PEGASUS policy search [23], implemented with locally greedy hill-climbing search, to optimize the parameters on a set of predefined training tasks. Each of the training tasks involves moving a single foot to a new location. During learning, we use a reward function that penalizes undesirable behaviors such as taking a long time to complete the foot movement, passing too close to the vertical surface of an obstacle, or failing to move the foot to the desired goal location. The policy search algorithm tries to choose parameters so as to maximize the rewards. The learned parameters are then fixed, giving us the low-level controller that will be used by the high-level planner.

IV. High-level planner

Given the low-level controller for moving a single leg, we now need a high-level planner for generating the sequence of target foot positions. Finding this sequence of target foot positions represents a difficult search problem because a bad choice of foot position early in the sequence could lead the robot to a bad state (for example, one with very bad posture), from which it may be extremely difficult to make further progress. Moreover, performing exhaustive search over all possible sequences of foot positions is computationally intractable, because of the high branching factor and large search depth involved.

In order to avoid performing exhaustive search, we need a way of measuring how "good" a state is. For example, in A^* search, a heuristic cost-to-go function gives estimates of the optimal future cost from any given state to the goal. However,

[1]Only 12 of the components of $\tilde{g}_t$ correspond to robot joint angles, and are executed on the servomotors. The other 6 components are ignored.

[2]The action space used by the low-level controller is 12d and not 3d, since even if the current step requires only moving foot 1, we may still wish to move the servomotors in the other legs to maintain balance.

[3]We used Open Dynamics Engine (ODE) to build our dynamical simulator. See http://ode.org for details on ODE.

it is often difficult to hand-specify a good heuristic cost-to-go function in complex problems, since it requires estimating the entire sequence of unknown future costs. In reinforcement learning (RL) [29], the "goodness" of a state s is measured by the optimal value function $V^*(s)$. Informally, this is the "expected optimal future rewards" starting from the state s. Using reinforcement learning, we will automatically learn an approximate value function.

A. Reinforcement learning preliminaries

This section gives a brief overview of the reinforcement learning formalism. For more details, see, e.g., [29].

In reinforcement learning, we model a system to be controlled as having a set of possible states S, and a set of possible actions A. Further, we are given a reward function $R : S \times A \rightarrow \mathbb{R}$, so that $R(s,a)$ is the reward for taking action a in state s. The system's dynamics are given by a state transition function $F : S \times A \rightarrow S$, defined so that $s' = F(s,a)$ is the state reached upon taking action a in state s.[4] A policy is any function $\pi : S \rightarrow A$ mapping from the states to the actions. We say we are executing a policy π if whenever we are in state s, we take action $a = \pi(s)$. The value function $V^\pi(s_0)$ for a policy π is defined as the sum of discounted rewards upon starting in state s_0 and taking actions according to π:

$$V^\pi(s_0) = R(s_0,a_0) + \gamma R(s_1,a_1) + \gamma^2 R(s_2,a_2) + \dots$$

where $a_i = \pi(s_i)$, $s_{i+1} = F(s_i,a_i)$, and $\gamma \in [0,1)$ is the discount factor. The discount factor causes rewards in the distant future to be weighted less than rewards in the near future. The optimal value function is defined as

$$V^*(s) = \max_\pi V^\pi(s).$$

This is the best possible sum of discounted rewards that can be attained using any policy. The optimal value function satisfies the Bellman equation:

$$V^*(s) = \max_a R(s,a) + \gamma V^*(s'), \tag{6}$$

where s' is the state reached upon taking action a in state s. In other words, the maximum possible value starting from the current state s is given by maximizing the sum of the immediate one-step reward $R(s,a)$ and the future rewards from the next state s'. Once $V^*(\cdot)$ is known, finding the policy that gives the maximum sum of discounted rewards is easy. The optimal policy $\pi^* : S \rightarrow A$ is given by

$$\pi^*(s) = \arg\max_a R(s,a) + \gamma V^*(s'). \tag{7}$$

In our high-level planner, taking an action a will correspond to choosing a new position for one of the feet. The reward function we used gave positive rewards for (1) movement of the robot's CG towards the goal; small negative rewards for (2) the time it took to execute the movement; and very large negative rewards for (3) failure of low-level controller to execute the action.

[4]Although the general RL framework is most commonly applied to problems with stochastic dynamics, here we will use a simplified version with only deterministic dynamics.

B. Value function approximation

There is a large number of existing RL algorithms for exactly learning the optimal value function. However, most of the exact algorithms apply only to problems with small, finite, state-spaces. For problems with large, continuous state spaces, it is generally not possible to obtain $V^*(\cdot)$ exactly, since even explicitly storing $V^*(s)$ for every $s \in S$ is impossible. In such cases, an approximate representation for $V^*(\cdot)$ must be used. Following fairly standard practice in RL, we will approximate $V^*(s)$ as a linear combination of a set of features of the state s. Specifically, we approximate $V^*(s)$ as:

$$\hat{V}(s) = \theta^T \phi(s),$$

where $\phi(s)$ is a vector of features of the state s, and θ is a parameter vector that will be learned.

In our approach, $\phi(s)$ consisted of the following features of the state[5]: (1) Distance from robot's CG to the goal; (2) Average distance from the feet to the goal; (3) Orientation of the body, expressed as $(1-\cos(\psi_1), 1-\cos(\psi_2), 1-\cos(\psi_3))$; (4) Maximum knee angle; (5) A feature capturing surface roughness underneath each foot on the ground[6]; (6) Surface curvature[7]; (7) Difference between maximum and minimum heights of the feet; (8) Area of supporting triangle (formed by the three stationary feet); (9) Radius of the largest circle that inscribes the supporting triangle; (10) The distances between each pair of feet.

C. Reinforcement learning algorithm

To learn an approximation to the value function, we developed a new partially supervised learning algorithm, inspired by Support Vector Machines [30], for learning an approximation to the value function. Our algorithm is based on the observation that in order to have $\hat{\pi}(s)$—the action chosen using our learned value function $\hat{V}$—to be the same as the optimal action $\pi^*(s)$, we need only for the following to hold:

$$R(s,\pi^*(s)) + \gamma \hat{V}(F(s,\pi^*(s))) > R(s,a) + \gamma \hat{V}(F(s,a)). \tag{8}$$

for all actions $a \neq \pi^*(s)$. Thus, if we have a "training set" consisting of 3-tuples $(s_i, a_i^*, a_i), i = 1,\dots,m$ where $s_i \in S$, $a_i^* = \pi^*(s_i)$ and $a_i \neq \pi^*(s_i)$, then one might try to find an approximation $\hat{V}$ to the value function that satisfies the equation above for all tuples in the training set. Since $\hat{V}(s) = \theta^T \phi(s)$ is linear in the parameters θ, this simply gives a set of linear constraints, and in fact any number of standard linear classification algorithms (such as softmax

[5]We fixed the foot movement order as in the conventional trot gait (left-front, right-rear, right-front, left-rear), and so our state actually also includes information about which foot is to be moved next.

[6]This is computed using a (gaussian weighted) sum of squares of the local gradient of the terrain height, over a small patch centered at each foot on the ground.

[7]Computed as a (gaussian weighted) sum of the second derivatives of the terrain height function, over a small patch centered at each foot on the ground.

regression [31], and support vector machines [30]) can be used to learn the parameters.

However, in our experiments, one additional refinement to this idea caused it to work significantly better. Specifically, under mild conditions, it can be shown that the optimal value function V^* is the unique solution to the Bellman equations (Eq. 6). Thus, one possible approach to approximating V^* is to find the parameters θ that come as close as possible to solving the Bellman equations. More precisely (following [32]), we can solve for θ so as to minimize the squared Bellman error:

$$\min_\theta \sum |\hat{V}(s_i) - R(s_i, a_i^*) - \gamma \hat{V}(F(s_i, a_i^*))|^2, \qquad (9)$$

where s_i and a_i^* are from our training set,[8] as described above, and $\hat{V}(s_i) = \theta^T \phi(s_i)$.

Putting together the two ideas described above—supervised learning to try to satisfy the constraints given in Eq. 8, as well as minimizing the squared Bellman error as in Eq. 9—we obtain the following optimization problem:

$$\begin{array}{ll} \max_{\theta,\delta} & \delta - \beta \sum |\hat{V}(s_i) - R(s_i, a_i^*) - \gamma \hat{V}(F(s_i, a_i^*))|^2 - \alpha ||\theta||_2^2 \\ \text{s.t.} & R(s_i, a_i^*) + \gamma \hat{V}(F(s_i, a_i^*)) \\ & -R(s_i, a_i) - \gamma \hat{V}(F(s_i, a_i)) \geq \delta \quad i = 1, \ldots, m. \end{array} \qquad (10)$$

Here, α and β are constants that determine the relative weighting of the different terms in the optimization objective. This optimization problem is a quadratic program, which can be readily solved [33]. Note that the constraints in the optimization are exactly demanding that Eq. 8 holds for all tuples in the training set (assuming $\delta > 0$), and moreover that the left-hand-side be larger than the right-hand-side by a gap or "margin" of at least δ.[9]

If $\beta = 0$, then the objective function reduces to only trying to optimize the constraints given in Eq. 8. (In fact, the resulting algorithm would be extremely similar to using a Support Vector Machine [30] with parameters θ to try to learn a classifier to identify optimal actions a_i^* vs. suboptimal actions a_i.) Conversely, for small or zero α, the optimization objective instead emphasizes minimizing the squared Bellman error, as in Eq. 9. However, by combining both of these objective functions together, we obtain a better algorithm than either approach alone.

In our experiments, we used a training set of 400 tuples of states, optimal actions (target foot positions) and suboptimal actions.[10] Out of these 400 training examples, there were only 16 distinct states s_i and (assumed) optimal actions a_i^*, which corresponded to a sequence of 16 footsteps that successfully climbed a single 2.6cm step. The suboptimal actions for these 16 steps were generated by manually labeling a number of foot positions that were clearly suboptimal or inappropriate for the current state s_i.

[8] We sum over only distinct s_i's.

[9] Following the L_1 norm soft margin SVM [30], we also considered adding slack parameters to address the case of linearly inseparable data. However, this turned out to be unnecessary for our application, and even without the soft margin, the algorithm attained good performance.

[10] We note that generating these training examples was fairly easy, since it is not difficult for this robot to climb the (relatively short) 2.6cm step. Thus, we could quite easily find a sequence of actions which succeeded in climbing this step; the actions in this sequence then formed our (assumed optimal) a_i^* in the training set. A set of (assumed suboptimal) actions a_i were similarly quickly generated.

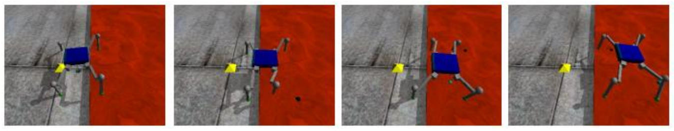


Fig. 3. Simulated robot executing a sequence of controls to climb up a single 4cm step.

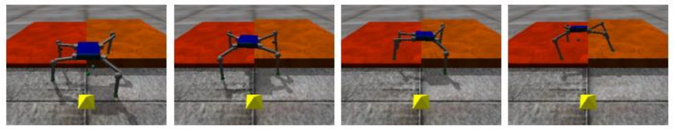


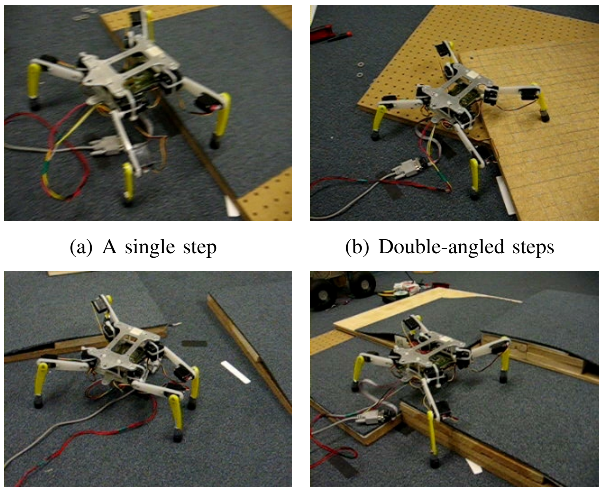
Fig. 4. Simulated robot climbing up an irregular step; the right portion of the step is 6cm and the right portion of the step is 3 cm.

We also tried learning a value function using temporal difference (TD) learning [29], which is one of the best known, standard, reinforcement learning algorithms. Unlike our algorithm, TD does not make use of a labeled training set during learning. But despite devoting significant efforts to tuning TD, including trying out different features, different minor modifications to the learning algorithm and so on, in every instance it still failed to produce a good value function even after running for more than 10 hours.

Having learned the parameters θ, we are now ready to generate actions using the high-level planner. Given $\hat{V}$, Eq. 7 gives a straightforward way to choose actions. However, as in [34] we can do even better by performing a multiple-step lookahead. More precisely, we used beam search to look multiple steps ahead, and to try to find a sequence of actions to move us towards the goal, or more formally, to maximize the total sum of discounted rewards. During the beam search procedure, we used our (deterministic) simulator for the robot to expand different search nodes, and used the learned $\hat{V}(s)$ as the heuristic function with which to guide the search algorithm. Our implementation discretized the space of possible actions a (possible foot placements) into 15 different values, and used a beam width of 10.

V. Experimental Results

All learning was done using our simulator of the robot. After learning the parameters for both the low- and the high-level controllers, we tested the resulting hierarchical policy, in simulation, on a variety of obstacles. In the first experiment, we asked the robot to climb up a 4cm tall step. Figure 3 shows the resulting sequence of robot motions. We also tested the robot on an "irregular" step. (See Figure 4.) In this example, the right half of the step is 6cm high, and the left half is 3cm high. We believe that it is extremely difficult to hand-code controllers for climbing asymmetric steps like these, since they require somewhat difficult, asymmetric (and, we find,


(a) A single step (b) Double-angled steps

(c) Double-cornered steps (d) Four interleaved steps

Fig. 5. Real robot executing learned sequence of controls to climb up obstacles.

somewhat non-intuitive) gaits. The learned controller successfully climbed this obstacle. In addition to these examples, the learned controller also appeared quite robust, and was able to climb a large variety of different obstacles, including steps at various heights (0-6cm); ones oriented at different angles relative to the robot; and irregular steps (with high and low portions) of different heights.

We also successfully demonstrated these gaits on the real robot, for a large variety of obstacles. For example, Figure 5 shows the real robot executing the gait shown in Figure 3, as well as gaits for several other obstacles.[11] Videos of the simulated and actual robot executing the learned gaits are available online, at `http://ai.stanford.edu/~yirong/quadruped/videos`

Note that our experiments used a training set (specifying optimal and sub-optimal target foot positions) based on climbing only a 2.6cm step. However, our learned gaits readily generalized to a large range of obstacles that were not seen in the training set, and were indeed significantly more complex and difficult than the single 2.6cm step seen during training.

For comparison, we also implemented the Rapidly exploring Random Trees (RRT) algorithm [35]. This is a commonly-used nonholonomic motion planning algorithm, that in principle can be applied to tasks such as ours. We spent significant effort adjusting RRT's parameters, but in all versions, even after running for more than twenty hours, it was never able to find control sequences for negotiating a single 4cm step.

[11] So far, our description has assumed a closed-loop controller that chooses the current action as a function of the state s. In our initial experiments, we used a 3-camera array and an IMU to estimate the robot state s, so as to execute a closed-loop controller. However, somewhat to our surprise, just by generating an open-loop sequence of actions (by executing the controller in closed-loop in simulation, and taking the resulting sequence of servomotor angles executed), we were also able to execute these gaits (comprising small numbers of steps) open-loop on the robot. We note also that the obstacle shown in Figure 3d, which comprises surfaces at four different heights, seemed particularly difficult. Our algorithm successfully found a gait for it using a beam width as small as two, but neither the Bellman-error only nor the supervised learning only ($\beta = 0$) algorithm was able to do so.

VI. Conclusions

We described the application of a hierarchical reinforcement learning algorithm, one that uses a two-level hierarchical task decomposition, to a quadruped robot. The algorithm was able to learn a controller for negotiating a large variety of different obstacles, including ones not seen during training. We believe that hierarchical reinforcement learning holds rich promise for negotiating obstacles and rough terrain with biped, quadruped and hexapod robots.

Acknowledgments

We thank Sebastian Thrun and Greg Lawrence for helpful conversations. The robot hardware used in this work was based on a design by Lynxmotion, and portions of it were built by Doron Tal. Support from the DARPA Learning Locomotion program (under contract number FA8650-05-C-7261) is gratefully acknowledged.

References

[1] S. Hirose, A. Nagabuko, and R. Toyama, "Machine that can walk and climb on floors, walls, and ceilings," in *International Conference on Advanced Robotics*, Pisa,Italy, 1991.

[2] S. Hirose, K. Yoneda, and H. Tsukagoshi, "Titan vii: Quadruped walking and manipulating robot on a steep slope," in *International Conference on Robotics and Automation*, Albuquerque, New Mexico, USA, 1997.

[3] S. Bai, K. H. Low, G. Seet, and T. Zielinska, "A new free gait generation for quadrupeds based on primary/secondary gait," in *International Conference on Robotics and Automation*, 1999, pp. 1371–1376.

[4] H. Kimura and Y. Fukuoka, "Adaptive dynamic walking of the quadruped on irregular terrain - autonomous adaptation using neural system model," in *International Conference on Robotics and Automation*, 2000, pp. 436–443.

[5] H. Kimura, Y. Fukuoka, Y. Hada, and K. Takase, "Three-dimensional adaptive dynamic walking of a quadruped - rolling motion feedback to cpgs controlling pitching motion," in *International Conference on Robotics and Automation*, 2002, pp. 2228–2233.

[6] Y. Fukuoka, H. Kimura, Y. Hada, and K. Takase, "Adaptive dynamic walking of a quadruped robot 'tekken' on irregular terrain using a neural system model," in *International Conference on Robotics and Automation*, 2003, pp. 2037–2042.

[7] S. Peng, C. Lani, and G. Cole, "A biologically inspired four legged walking robot," in *International Conference on Robotics and Automation*, 2003, pp. 2024–2030.

[8] Z. Zhang, Y. Fukuoka, and H. Kimura, "Adaptive running of a quadruped robot on irregular terrain based on biological concepts," in *International Conference on Robotics and Automation*, 2003, pp. 2043–2048.

[9] E. Z. Moore and M. Buehler, "Stable stair climbing in a simple hexapod," in *International Conference on Climbing and Walking Robots*, Karlsruhe, Germany, 2001, pp. 603–610.

[10] E. Z. Moore, D. Campbell, F. Grimminger, and M. Buehler, "Reliable stair climbing in the simple hexapod rhex," in *International Conference on Robotics and Automation*, vol. 3, Washington, D.C., U.S.A, 2002, pp. 2222–2227.

[11] M. H. Raibert, *Legged Robots that Balance*. Cambridge, MA: MIT Press, 1986.

[12] I. Poulakakis, J. A. Smith, and M. Buehler, "Experimentally validated bounding models for scout ii quadrupedal robot," in *International Conference on Robotics and Automation*, New Orleans, LA, USA, 2004, pp. 825–830.

[13] T. Bretl, S. Rock, J. Latombe, B. Kennedy, and H. Aghazarian, "Free-climbing with a multi-use robot," in *9th Int. Symp. on Experimental Robotics*, Singapore, 2004.

[14] J. Morimoto and C. Atkeson, "Minimax differential dynamic programming:an application to robust biped walking," in *Neural Information Processing Systems Conference*, 2002.

[15] J. Bagnell, S. Kakade, A. Ng, and J. Schneider, "Policy search by dynamic programming," in *Neural Information Processing Systems Conference*, Vancouver, Canada, 2003.

[16] N. Kohl and P. Stone, "Machine learning for fast quadrupedal locomotion," in *National Conference on Artificial Intelligence*, San Jose, California, USA, 2004.

[17] H. Kim, T. Kang, V. G. Loc, and H. R. Choi, "Gait planning of quadruped walking and climbing robot for locomotion in 3d environment," in *International Conference on Robotics and Automation*, 2005.

[18] J. Chestnutt, J. Kuffner, K. Nishiwaki, and S. Kagami, "Planning biped navigation strategies in complex environments," in *International Conference on Humanoid Robots*, 2003.

[19] R. Sutton, D. Precup, and S. Singh, "Intra-option learning about temporally abstract actions," in *International Conference on Machine Learning*, 1998, pp. 556–564.

[20] M. Hauskrecht, N. Meuleau, C. Boutilier, L. P. Kaelbling, and T. Dean, "Hierarchical solution of markov decision processes using macroactions," in *Conference on Uncertainty in Artificial Intelligence*, 1998.

[21] R. Parr and S. Russell, "Reinforcement learning with hierarchies of machines," in *Neural Information Processing Systems Conference*, 1998.

[22] T. G. Dietterich, "Hierarchical reinforcement learning with the maxq value function decomposition," *Journal of Artificial Intelligence Research*, vol. 13, pp. 227–303, 1999.

[23] A. Y. Ng and M. I. Jordan, "Pegasus: A policy search method for large mdps and pomdps," in *Conference on Uncertainty in Artificial Intelligence*, 2000, pp. 406–415.

[24] J. Craig, *Introduction to Robotics: Mechanics and Control*, 2nd ed. Addison- Wesley Longman Publishing, 1989.

[25] O. Khatib, "Real-time obstacle avoidance for manipulators and mobile robots," *International Journal of Robotics Research*, vol. 5, no. 1, pp. 90–98, 1986.

[26] O. Brock, O. Khatib, and S. Viji, "Task-consistent obstacle avoidance and motion behavior for mobile manipulation," in *International Conference on Robotics and Automation*, Washington, USA, 2002.

[27] C. D. Medio and G. Oriolo, "Robot obstacle avoidance using vortex fields," in *Advances in Robot Kinematics*, S. Stifter and J. Lenarcic, Eds. Springer-Verlag, 1991, pp. 227–235.

[28] O. Khatib, "Inertial properties in robotics manipulation: An object-level framework," *International Journal of Robotics Research*, vol. 14, no. 1, pp. 19–36, February 1995.

[29] R. S. Sutton and A. G. Barto, *Reinforcement Learning*. MIT Press, 1998.

[30] V. N. Vapnik, *Statistical Learning Theory*. John Wiley & Sons, 1998.

[31] P. McCullagh and J. A. Nelder, *Generalized Linear Models (second edition)*. Chapman and Hall, 1989.

[32] L. C. Baird, "Residual algorithms: Reinforcement learning with function approximation," in *International Conference on Machine Learning*, 1995.

[33] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.

[34] S. Davies, A. Y. Ng, and A. Moore, "Applying online-search to reinforcement learning," in *National Conference on Artificial Intelligence*, 1998.

[35] S. M. LaValle and J. J. Kuffner, "Randomized kinodynamic planning," in *International Conference on Robotics and Automation*, Detroit, Michigan, USA, 1999.

APPENDIX
LOW-LEVEL CONTROLLER DETAILS

Here, we describe the potential functions and the vortex field in detail. Our coordinate system is defined with the robot's direction of travel being the positive x direction, with up being the positive z direction, and y being given by the right-hand rule. We will use the following notation:

t	is the time since the start of the task (of moving a single leg)
Ω_t	is the state vector at time t
i	is the index of the moving foot
u_t^j	$u_t^j = (u_{t,x}^j, u_{t,y}^j, u_{t,z}^j)$ is the position vector of foot j at time t
u_g	$u_g = (u_{g,x}, u_{g,y}, u_{g,z})$ is the goal position for the moving foot
N_{obs}	is the number of obstacles in the environment. Our current implementation permits only obstacles that are rectangular boxes. Note that the ground is also considered as an obstacle in the definitions below.
$\xi(\Omega_t)$	is a measure of how far the center of gravity(CG) of the robot is inside the triangle formed by the 3 supporting legs. It is calculated by projecting the positions of the 3 supporting feet and the CG into the xy plane and measuring the minimum distance between projection of the CG to the sides of the triangle formed by the projected feet positions. $\xi(\Omega_t)$ is positive if the CG is within the triangle, and negative if it is outside the triangle.

The low-level controller has 20 parameters: $\rho_1, \rho_2, \ldots, \rho_{16}, \lambda_1, \lambda_2, \tau_1, \tau_2$. The ρ_i's control the shape and size of the various potential and vortex fields. The λ_i's and τ_i's determine which potentials are turned on based on the position of the CG and the amount of time elapsed.

At the beginning of each task, the robot starts in a centering phase, during which the goal and surface potentials are off while the CG potential is on. This allows the robot to shift its CG into the new supporting triangle before it tries to take a new step. The initial centering phase will last a minimum of τ_2 time steps and will terminate when either the CG is sufficiently inside the supporting triangle (i.e., $\xi(\Omega_t) \le \lambda_1$) or when the maximum number of time steps for the initial centering phase τ_2 is reached. The goal and surface potentials are also turned off whenever the center of mass is near or outside the boundary of the supporting triangle ($\xi(\Omega_t) \le \tau_1$); this in effect stops the motion of the active foot and allows the robot to move its CG into a stable position before continuing with moving the active foot.

A. Goal Potential

The goal potential attracts the moving foot to the goal location u_g.

If we are in the initial centering phase or if the CG is near the boundary of the supporting triangle (i.e., $\xi(\Omega_t) \le \lambda_1$), then $U_t^g = 0$. Otherwise,

$$\begin{aligned} U_t^g &= \rho_1 ||u_t^i - u_g||_2 - \rho_2 \exp(-\rho_3 ||u_t^i - u_g||_2) \\ &\quad + \rho_4 (u_{t,y}^i - u_{g,y})^2 \end{aligned}$$

The third term puts a heavier emphasis on the y coordinate to reduce small oscillations of the moving foot in the y direction.

B. Surface Potential

Away from the goal position, the surface potential repels the moving foot from the surfaces of obstacles and the ground. The strength of this potential decays to 0 as one approaches the goal position.

If we are in the initial centering phase or if the CG is near the boundary of the supporting triangle (i.e. $\xi(\Omega_t) \le \lambda_1$), then

$U_t^s = 0$. Otherwise,

$$U_t^s = (1 - \exp(-\rho_7 ||u_t^i - u_g||_2^2)) \sum_{k=1}^{N_{\text{obs}}} \rho_5 \exp(-\rho_6 d_k)$$

where d_k is the distance from the moving foot to obstacle k.

C. Posture Potential

The posture potential is defined as the sum of three other potentials

$$U_t^p = U_t^c + U_t^o + U_t^f$$

where U_t^c, U_t^o and U_t^f are the CG, orientation, and support feet constraint potentials respectively.

1) CG Potential: The CG potential encourages the the CG of the robot to stay within the triangle formed by the three supporting feet. It is always on during the initial centering phase of a new task. It is also on when the CG is near or outside the boundary of the supporting triangle.

If we are not in the initial centering phase and $\xi(\Omega_t) \geq \lambda_2$ then $U_t^c = 0$. Otherwise,

$$U_t^c = \exp(-\rho_9 \min(\xi,\ \rho_8))$$

2) Orientation Potential: The orientation potential discourages large roll and yaw of the robot body and is defined as

$$U_t^o = \rho_{10}(1 - cos(\psi_1)) + \rho_{11}(1 - cos(\psi_3)),$$

where ψ_1 and ψ_3 are respectively the roll and yaw of the robot body.

3) Constraint Potential: The constraint potential tries to keep the supporting feet at the same position as they were at the beginning of the task and is defined as

$$U_t^f = \rho_{12} \sum_{j=1, j\neq i} ||u_t^j - u_0^j||_2^2$$

Although it may seem that the constraint potential is redundant since it accomplishes the same purpose as the nullspace projection described by 4, it is nonetheless necessary because of errors in the linearization used by the nullspace projection.

D. Vortex Field

The vortex field encourages the moving foot to move up along the vertical surfaces of obstacles. The vortex field is used to help ensure that the moving foot does not get stuck in regions where the potential function has nearly zero gradient (e.g., places where the goal and surface potential gradients point in nearly opposite directions). The strength of the vortex field of an obstacle decays with distance to the obstacle.

The total vortex field is the sum of the vortex fields induced by all the obstacles. Let $\tilde{v}_k$ denote the vortex field induced by obstacle k. Let $p_k = (p_{k,x}, p_{k,y}, p_{k,z})$ be the nearest point on obstacle k to the position of the moving foot u_t^i. If $u_{t,x}^i = p_{k,x}$ then the moving foot is already above a flat region of obstacle k and we set $\tilde{v}_k = 0$. If the moving foot is higher than and relatively close to the goal position (i.e., $u_{t,z}^i > u_{g,z}$) and $|u_{t,x}^i - u_{g,x}| < \rho_{16}$, then we expect the goal potential will be able to pull the foot in and so we set all the vortex fields to 0 in this case. In all other cases, $\tilde{v}_k$ is calculated as

$$\tilde{v}_k = \frac{\rho_{13}}{1 + \exp(\rho_{14}(||u_t^i - p_k||_2 - \rho_{15}))} \cdot \frac{\text{sign}(u_{t,x}^i - u_{g,x})\vec{v}_k}{||\vec{v}_k||_2}$$

where $\vec{v}_k$ is the vector cross product

$$\vec{v}_k = (u_t^i - p_k) \times \hat{y}$$

and $\hat{y}$ is the unit vector in the y direction.

We note that $\tilde{v}$ is given in terms of the 3d coordinates of the moving foot and we need to convert it into the 18d state space of the robot. This is done by multiplying by the Jacobian made up of the partial derivatives of the components of the moving foot's position with respect to the state variables.