# Learning To Reach and Reaching To Learn

## A Unified Approach to Path Planning and Reactive Control through Reinforcement Learning

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera Italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Mikhail Alexander Frank, M.Sc.

under the supervision of
Prof. Jürgen Schmidhuber and Dr. Alexander Förster

September 2014

Dissertation Committee

| | |
|---|---|
| **Prof. Rolf Krause** | Università della Svizzera italiana, Switzerland |
| **Prof. Kai Hormann** | Università della Svizzera italiana, Switzerland |
| **Prof. Giorgio Metta** | Italian Institute of Technology, Genova, Italy |
| **Prof. Ben Kuipers** | University of Michigan, Michigan, USA |

Dissertation accepted on 30 September 2014

| Research Advisor | Co-Advisor |
|---|---|
| **Prof. Jürgen Schmidhuber** | **Dr. Alexander Förster** |

PhD Program Director

**Prof. Igor Pivkin**

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Mikhail Alexander Frank, M.Sc.
Lugano, 30 September 2014

# Abstract

The next generation of intelligent robots will need to be able to plan reaches. Not just ballistic point to point reaches, but reaches around things such as the edge of a table, a nearby human, or any other known object in the robot's workspace. Planning reaches may seem easy to us humans, because we do it so intuitively, but it has proven to be a challenging problem, which continues to limit the versatility of what robots can do today.

In this document, I propose a novel intrinsically motivated RL system that draws on both *Path/Motion Planning* and *Reactive Control*. Through *Reinforcement Learning*, it tightly integrates these two previously disparate approaches to robotics. The RL system is evaluated on a task, which is as yet unsolved by roboticists in practice. That is to put the palm of the iCub humanoid robot on arbitrary target objects in its workspace, starting from arbitrary initial configurations. Such motions can be generated by *planning*, or searching the configuration space, but this typically results in some kind of trajectory, which must then be tracked by a separate controller, and such an approach offers a brittle runtime solution because it is inflexible. Purely *reactive* systems are robust to many problems that render a planned trajectory infeasible, but lacking the capacity to search, they tend to get stuck behind constraints, and therefore do not replace motion planners.

The planner/controller proposed here is novel in that it deliberately plans reaches without the need to track trajectories. Instead, reaches are composed of sequences of reactive motion primitives, implemented by my Modular Behavioral Environment (MoBeE), which provides (fictitious) force control with reactive collision avoidance by way of a realtime kinematic/geometric model of the robot and its workspace. Thus, to the best of my knowledge, mine is the first reach planning approach to simultaneously offer the best of both the *Path/Motion Planning* and *Reactive Control* approaches.

By controlling the real, physical robot directly, and feeling the influence of the constraints imposed by MoBeE, the proposed system learns a stochastic model of the iCub's configuration space. Then, the model is exploited as a multiple query path planner to find sensible pre-reach poses, from which to initiate reaching actions. Experiments show that the system can autonomously find practical reaches to target objects in workspace and offers excellent robustness to changes in the workspace configuration as well as noise in the robot's sensory-motor apparatus.

# Contents

# Chapter 1

# Introduction

While the neuroscience community tries to unravel the inner workings of the brain, the Artificial Intelligence (AI), Machine Learning (ML), and Developmental Robotics communities try to create computational/algorithmic problem solvers, which are inspired by the learning process observed in biological organisms. Despite considerable success on certain problems, such as chess, which people tend to view as *hard*, AI and ML approaches have encountered significant difficulties in solving other problems, such as manipulating the chess pieces, which conventional wisdom tells us should be *easy*. In fact, after some sixty years of AI and Robotics research, the average five year old is still *far* better at manipulating chess pieces than today's most advanced robots.

Why should object manipulation be so difficult? After all, we adults intuitively manipulate all kinds of different objects in different ways, constantly, without even thinking about it. Let us consider the problem from the standpoint of programming a robot to do some basic object manipulation, similar in complexity to what we see small children doing with blocks; for example, picking and placing at arbitrary locations on some kind of work surface, sorting, maybe even stacking and knocking over.

The essence of the problem is that like us, robots that are designed to manipulate objects (manipulators) usually have a lot of joints, or degrees of freedom (DOF). Therefore, what looks to the casual observer like a simple and intuitive reach is actually the result of highly coordinated motion involving 6 to 10 motors. Finding such high-dimensional trajectories (path planning) is a difficult problem, even when the objects in the robot's immediate environment (workspace) are assumed to be static. However, consider that by definition, *manipulation* means to move objects around the workspace. Therefore, to manipulate objects in practice, a robot must be able to plan trajectories around

those objects, even though they are not always in the same places. Furthermore, manipulating objects requires that plans be executed faithfully, typically by some kind of feedback controller. Therefore, plans must respect the robot's dynamic constraints, which drastically increases the complexity of the planning problem. Nevertheless, plans must somehow be generated quickly, and re-planning must be possible at any moment, in order to cope with noisy sensory-motor apparatuses as well as environmental changes that result from external influence, like a human co-worker.

Clearly, even simple instances of the *object manipulation problem* require elements of *sensing*, *motion planning*, and *reactive control*. Each of these elements constitutes its own body of literature, however the work proposed here integrates key aspects of deliberate planning and reactive control in a realtime reinforcement learning (RL) system. The result is a tightly integrated behavioral control apparatus, which allows a robotic manipulator to intelligently and autonomously learn to reach objects in its workspace.

## 1.1 On Humanoid Robots

Currently available industrial robots are employed to do repetitive work in structured environments, and their highly specialized nature is therefore unproblematic, or even desirable. However the next generation of robot helpers is expected to tackle a much wider variety of applications, working alongside people in homes, schools, hospitals, offices, city streets, war zones, disaster areas, spacecraft and places we haven't even thought of yet.

The hardware exists already. State-of-the-art humanoid robots such as the National Aeronautics and Space Administration / General Motors (NASA/GM) Robonaut 2 [Diftler et al., 2011], the Willow Garage Personal Robot 2 (PR2) [Cousins, 2010], the iCub from the Italian Institute of Technology (IIT) [Metta et al., 2008], and Toyota's Partner Robots [Takagi, 2006; Kusuda, 2008] are technologically impressive, and sometimes eerily anthropomorphic machines. Two complex, high-DOF hands/arms allow them, in principal anyway, to interact with a wide variety of objects of different sizes and shapes. This encourages the belief that a humanoid robot has the potential to adapt to a much wider variety of circumstances than its industrial counterparts ever could.

In addition to the potential versatility of humanoid robots, they pose interesting challenges to currently prevalent methodology. In contrast to traditional, industrial manipulators, humanoids typically exhibit a great deal of kinematic redundancy, and vision is provided by stereo cameras that move, not only as a

pair but often with respect to one another. Such anthropomorphic design elements conspire to violate many of the assumptions found in the literature, which provides an opportunity to extend the state-of-the-art.

The iCub humanoid embodies all of the above characteristics, making it an ideal platform upon which to study approaches to reaching, and eventually object manipulation. Moreover it is relatively inexpensive, widely used (at least in Europe), and it is the first commercially available, open source robot, which gives a researcher and developer such as myself the opportunity to collaborate with others and contribute to their efforts in concrete ways both now and in the future. It is for these reasons that I have chosen to use the iCub robot as my primary research platform.

## 1.2 Motivation for a Developmental Approach

Physically speaking, humanoids should be capable of doing a much wider variety of jobs than their industrial ancestors. Behaviors, however, are still programmed manually by experts and the resulting programs are generally engineered to solve a particular instance (or at best a few related instances) of a task. Consequently, these advanced robots are endowed with relatively few, highly specialized control programs, and their versatility remains quite limited.

In order to realize the potential of modern humanoid robots, especially with respect to service in unstructured, dynamic environments, we must find a way to improve their adaptiveness and exploit the versatility of modern hardware. This will undoubtedly require a broad spectrum of behaviors that are applicable under different environmental constraints/configurations. At the highest level, the planner/controller must solve a variety of different problems by identifying relevant constraints and developing or invoking appropriate behaviors. However we, as engineers and programmers, are not likely to be able to explicitly and accurately predict the wide range of constraints and operating conditions that will be encountered in the real world, where the next generation of robots should operate. This motivates the *developmental approach* to robotics, which focuses on systems that adaptively and incrementally build a repertoire of actions and/or behaviors from experience.

To effectively *learn* from experience, a planner/controller must *explore*, and that idea is where my research began. But before I go into that, I will cover some pre-requisites from the robotics literature, which correspond to critical sub-problems implied by terms like *reaching* and *object manipulation*.

## 1.3 Path Planning

We are all intimately familiar with the path planning problem. Should I sit in this traffic jam, or should I chance a longer route in hopes that I can maintain a higher average speed? Is it quicker to go over the mountain or around it? Just how does one get into a wetsuit, or a ski boot, or get the oil filter out from inside that engine without burning one's forearm on the exhaust manifold? These are pertinent life questions, which we all deal with every day, and regardless of the problem domain, a *plan* consists of a sequence of temporally dependent *actions*.

A particularly illustrative example of the path planning problem is navigating in the mountains, when a storm cloud develops. There is no shelter available nearby, and one wants to get down. It could be that the smart thing to do is to resist the urge to descend, instead traversing a ridge line for a while, or even climbing a little bit, in order to descend later, but it is hard to know based only on one's immediately observable environment. A contrasting approach would be to simply look around, find the steepest navigable slope, and proceed downward. The problem is that this approach may lead deeper into the mountains, or one might get stuck in a valley and end up having to climb a lot more, but it does offer the benefit that it took almost no time to put the naive plan into action.

A good planner, upon seeing the thunder cloud, pulls out his or her trusty map and used this global knowledge to determine the *best path* to get out of the mountains quickly and efficiently. An even better planner also has the capacity to run for shelter if need be.

## 1.4 The Path Planning Problem for Robots

In robotics, the *Path Planning Problem* is to find motions that pursue goals, usually robot positions, while deliberately avoiding constraints, usually obstacles. The ability to solve the path planning problem in practice is absolutely critical to the eventual deployment of complex/humanoid robots outside of carefully controlled industrial environments. For a serial manipulator[1], the path planning problem is formalized as follows: The workspace,

$$W \subset \mathbb{R}^3 \tag{1.1}$$

contains a robot composed of $n$ joints/links. Each link, $A_i$, where $i \in \{1, 2, \ldots, n\}$, is represented by a semi-algebraic model. The vector of joint

[1]For illustrative purposes, I have chosen to formalize the path planning problem for a serial kinematic chain. The formalism can be extended straightforwardly to handle branching.

positions

$$q \in C \subset \mathbb{R}^n \tag{1.2}$$

denotes the *configuration* of the robot, and kinematic constraints yield a proper functional mapping $q \rightarrow A(q)$ defining its *pose*:

$$A(q) = \bigcup_{i=1}^{n} A_i(q) \subset W \tag{1.3}$$

Furthermore, there exist $m$ obstacles, $B_i \subset W$, where $i \in \{1, 2, \dots m\}$, which are also expressed as semi-algebraic models, and together they define the obstacle region:

$$B = \bigcup_{i=1}^{m} B_i \subset W \tag{1.4}$$

The set of configurations that cause the robot to collide with these obstacles can be expressed:

$$C_B = \{q \in C \mid A(q) \cap B\} \tag{1.5}$$

Analogously, the set of configurations that cause self-collisions can be expressed:

$$C_A = \bigcup_{\{j,k\} \in S} \{q \in C \mid A_j(q) \cap A_k(q)\} \tag{1.6}$$

where $S$ is a set of pairs of indices, $\{j, k\} \in \{1, 2, \dots, n\}$, with $j \neq k$, corresponding to two links $A_j$ and $A_k$, which should not collide. Thus, the set of all configurations, which are infeasible due to collisions can be expressed:

$$C_{\text{colliding}} = C_B \cup C_A \tag{1.7}$$

To find feasible motions, we must disambiguate the feasible ($C_{\text{free}}$) and infeasible ($C_{\text{colliding}}$) regions of the configuration space, which are complimentary:

$$C_{\text{free}} = C \setminus C_{\text{colliding}} \tag{1.8}$$

The path planning problem is essentially to find a trajectory $Q(t)$ such that:

$$\{q_i, q_g\} \subset Q(t) \subset C_{\text{free}} \tag{1.9}$$

In other words, $Q(t)$ interpolates initial and goal configurations, $q_i$ and $q_g$, while not intersecting $C_{\text{colliding}}$.

The path planning problem is probably NP Hard. Therefore, building real-world solutions is difficult, and approaches are usually confined to the lab and evaluated according to some standard algorithmic analysis in terms of criteria such as convergence guarantees, soundness and (resolution) completeness for an infinite time horizon. Such analysis requires that the problem does not change while the algorithm is running, and consequently most (if not all) proper planning algorithms require a static workspace in order to function properly, which is of course impractical for many applications.

## 1.5 Planning Algorithms

There exists a vast literature on *Path Planning* or *Motion Planning* algorithms, and the text book 'Planning Algorithms' [LaValle, 2006] provides an excellent overview. Here I focus on those algorithms that scale to the high dimensional configuration spaces of complex/humanoid robots.

Sampling based motion planning algorithms probe the configuration space with some sampling scheme. The samples $q$ are mapped to poses $A(q)$, which are in turn used to do collision detection computations, revealing whether $q \in C_{free}$ or $q \in C_{infeasible}$. The samples are interpolated, and in this way, feasible motions are constructed piece-by-piece. This can either be done on an as-needed basis, which is known as single query planning and exemplified by the Rapidly Exploring Random Trees (RRT) algorithm [LaValle, 1998; Perez et al., 2011], or the results of queries can be aggregated and stored such that future queries can be fulfilled faster, which is known as multiple query planning and exemplified by the Probabilistic Road Map (PRM) algorithmm [Latombe et al., 1996; Li and Shie, 2007]. Consider in broad terms the benefits and drawbacks of these two approaches.

The strength of single query planning algorithms is that they directly and effectively implement exploration by searching for feasible motions through trial and error. Some algorithms, such as RRT [LaValle, 1998], and its many descendants, even offer probabilistic completeness, guaranteeing a solution in the limit of a dense sampling sequence, if one exists. Moreover, since these algorithms answer each query by starting a search from scratch, they can readily adapt to different $C_{free}$ and $C_{infeasible}$ from one call to the next. It is however important to realize that $C_{free}$ and $C_{infeasible}$ must remain constant during the course of planning, which can take a considerable amount of time. The primary drawback of single query algorithms is their high complexity, which is $O(m^n)$, where $m$ is linear sampling density and $n$ is the dimensionality of the configuration space.

A recent state-of-the-art algorithm, Ball Tree RRT (BT-RRT) [Perez et al., 2011], requires 10 seconds to find a feasible solution to a relatively easy planning problem, wherein two arms (12 DOF) must circumnavigate the edge of a table to reach a cup. Moreover, the initial solution, the result of stochastic search, is quite circuitous, and BT-RRT requires an additional 125 seconds to smooth the motion by minimizing a cost function in the style of optimal control.

Practically speaking, a latency of tens to hundreds of seconds with respect to a robot's response to commands is rarely acceptable. This is the primary motivation for multiple query planning algorithms, which typically utilize a *roadmap* data structure in the form of a graph $G(V, E)$, where $V = \{v_1, v_2, ...v_n\} \in C_{free}$ and $E$ is a set of pairs of indices $(j, k) \in V$ such that $j \neq k$ and $j, k \in \{1, 2, ...n\}$. With each member of $E$ is associated a verified collision free trajectory $T(E_i) \subset C_{free}$.

The roadmap approach reduces each query from a search in $\mathbb{R}^n$ to graph search, which can be carried out by Dijkstra's shortest path algorithm [Dijkstra, 1959], A* [Hart et al., 1968], or similar. Importantly, the roadmap graph represents a natural crossroads between motion planning as an engineering discipline and the field of artificial intelligence, as it is a special case of a Markov Decision Process (MDP) [Puterman, 2009], where *states* are configurations $q \in C_{free}$, *actions* are trajectories $q(t) \subset C_{free}$, and all state transition probabilities are equal to one.

Early versions of the roadmap approach, such as PRM, first constructed the map offline, then queried it to move the robot [Latombe et al., 1996]. Whereas more recent versions can build the map incrementally on an as-needed basis by extending the current map toward unreachable goal configurations using single query algorithms [Li and Shie, 2007]. The ability of roadmap planners to quickly satisfy queries, even for complex robots with many DOF, makes them an appealing choice for practical application in experimental robotics. Moreover, when roadmaps are constructed incrementally by single query algorithms, the resulting system is one that builds knowledge from experience gained through exploratory behavior. As the roadmap grows over time, it becomes more competent at navigating the regions of the configuration space in which the planner has operated in the past.

Although the incrementally learned roadmap [Li and Shie, 2007] makes significant steps toward the autonomous development/acquisition of reusable behaviors, it is plagued by one very unrealistic assumption, namely that $T(E_i) \subset C_{free}$ for all time. In other words, neither the obstacle region $B$ nor the robot itself $A$ can change in a way that might have unpredictable consequences with respect to the roadmap $G(V, E)$. Therefore, the roadmap approach implicitly prohibits the robot from grasping objects, which would change $A$, and even if

objects could be moved without grasping, that is anyway prohibited also, as it would change $B$.

In summary, single query planning algorithms such as RRT effectively implement exploration and can readily adapt to changes in the environment from one query to the next, however they produce circuitous motions that require smoothing, and they are not fast enough to be applied online in practice. Multiple query roadmap based algorithms are quite fast, as they reduce the planning query to graph search, and when the map is constructed incrementally by single query algorithms, the resulting system clearly aggregates knowledge from experience through exploration. The drawback of the incrementally constructed roadmap is that it requires a static environment, which is also not practical in practice.

## 1.6 Reactive Control

An alternative to planning feasible actions preemptively is to adopt some heuristic, like *go straight*, and react to impending constraints/collisions as they are detected. This can either be done by interpreting the sensory data directly [Brooks, 1991], or by using a robot/workspace model.

### 1.6.1 The Subsumption Architecture

A pioneer focused on autonomy and robustness, Rodney Brooks built behaviors for robots by hand, according to his *Subsumption Architecture* [Brooks, 1991]. His 'Critters' were predominantly simple mobile robots, but they operated with considerable autonomy in real-world settings.

The Subsumption Architecture is based on asynchronous networks of Finite State Machines (FSM) and one of its defining characteristics is that the it does not maintain a robot/workspace model. Instead, sensors are connected directly to actuators via the FSM network. Brooks argues that the world is its own best model, and the claim is well demonstrated in the domain of mobile robots. However, I am interested in developing manipulation behaviors for humanoids, and this poses a different set of problems than does the control of a mobile robot.

Consider for a moment the relationship between the sensory and action spaces of mobile robots and humanoids. Mobile robots have a few controllable DOF, and are confined to move on a planar surface. They typically carry a number of cameras or range finding sensors, arranged radially about the robot and facing outward. Such a sensor array gives a natural representation of obstacles and free space around the robot, and behavioral primitives, such as *go forward*,

*stop* and *turn left/right*, can therefore be designed conveniently in that same planar space.

Although a humanoid robot has a similar sensory system to a mobile robot, an array of cameras and/or range finders, which capture 2 and 3D projections of the workspace, in most cases such information is not adequate to characterize the state of the robot.

A humanoid has a very large number of controllable DOF, and often cannot see most of its body. Shoulders and elbows for example are critical for executing reaches, but their states are not directly observable through vision most of the time. Instead, the state of the robot must be understood though proprioceptive information, consisting of joint positions, and perhaps higher order information.

Thus, in order to develop a complete picture of the state of the robot/world system, some kind of model is required in order to merge/fuse sensory data perceived in the workspace (vision) with that perceived in configuration space (proprioception).

### 1.6.2 The Potential Field Approach

Perhaps the earliest and most elegant approach to model-based reactive control was originally known as *real time obstacle avoidance* [Khatib, 1986; Kim and Khosla, 1992], however it has become widely known as the *potential field* approach, and is formulated as follows in terms of the notation from section 1.4 as follows: Consider a point

$$x \in A_i(q) \subset W \tag{1.10}$$

and let

$$U_B(x) = \sum_{i=1}^{m} U_{Bi}(x) \tag{1.11}$$

be a repulsive potential field function, which represents the influence of $m$ obstacles in the workspace on $x$. Let each $U_{Bi}$ be a continuous, differentiable function, defined with respect to an obstacle region $B_i \subset W$, such that $U_{Bi}$ is at a maximum in the neighborhood of the boundary of $B_i$ and goes to zero far from $B_i$. Khatib suggests:

$$U_{Bi}(x) = \begin{cases} \frac{1}{2}\eta(\frac{1}{\rho(x)} - \frac{1}{\rho_0})^2 & : \rho(x) \leq \rho_0 \\ 0 & : \rho(x) > \rho_0 \end{cases} \tag{1.12}$$

where $\rho$ is the shortest distance from $x$ to $B_i$, $\rho_o$ controls the (geometric) size of the potential field, and $\eta$ controls its maximum value. The defining characteristic of the potential field approach is that the robot is controlled such that $x$ descends the gradient of the potential field, $U_B$, and it can be done equally well for mobile robots and manipulators alike.

For a manipulator, the control input is computed as follows: The influence of the repulsive potential field $U_B$ on the robot is first computed as a force $f$ that acts on the robot at $x$:

$$f = -\frac{\partial U_B}{\partial x} = -\sum_{i=1}^{m} \frac{\partial U_{Bi}}{\partial x} \tag{1.13}$$

The force $f$ is then projected into the configuration space via the Jacobian matrix to yield joint torques $\tau$:

$$\tau(x) = J^T(q)\Lambda(x)f(x) \tag{1.14}$$

where $\Lambda(x)$ is a quadratic form, a kinetic energy matrix that captures the inertial properties of the end effector.

Reactive control assumes that timeliness is more important than algorithmic guarantees, and takes a heuristic approach. It seeks to compute a *good* control command at each *instant* in time. The command need not be the *optimal* in any sense, nor must there be any guarantees of what will happen over time. What is important is that the problem, meaning the workspace constraints, must be allowed to change, and that the controller must react in an intuitively appropriate manner to the workspace dynamics it is expected to encounter.

In practice, Reactive control is very effective with respect to quickly generating evasive motions to keep the manipulator away from obstacles, and it therefore excels in a dynamic workspace. Potential fields can even be defined to bring the end effector to some goal position, however this offers a very brittle solution to the motion planning problem, as non-convex potential functions, which arise from superposition, often create local minima in which the controller gets stuck.

Subsequent work has reformulated the potential field approach to improve the robustness of the implied global plan. For example, [Kim and Khosla, 1992] uses harmonic potential functions, which guarantee that no local minimum exists other than the global minimum, or alternatively, that point $x$ above, if treated as a point robot, will always be pushed to the goal from any initial condition. Still though there may exist structural local minima, configurations where non-point robots will not be able to proceed although they are being forced by the potential gradient.

Another reformulation is known as *attractor dynamics* [Schöner and Dose, 1992], wherein the robot does not descend the gradient of the potential function, but rather moves with constant velocity, adjusting its heading according to a dynamical system that steers toward the goal, but away from obstacles that lie in the robot's path. The method was developed for mobile robotics, however it has also been applied to manipulators [Iossifidis and Schöner, 2004, 2006]. And again, although attractor dynamics improves robustness over the original potential field approach, this time by keeping the state of the robot in the neighborhood of a stable attractor, it is still a heuristic planner that bases decisions on local information only, and it can therefore get stuck.

A third reformulation called *elastic strips* [Brock and Khatib, 2000] combines the local reactivity of the potential field approach with the more global framework of a roadmap planner. The edges of the roadmap graph are trajectories that are parameterized in such a way as to be deformable under the influence of a potential field. Again, this approach does improve robustness with respect to global planning, however it still suffers from structural local minima, and the elastic graph edges may not be able to circumvent certain obstacles. Failure to circumvent an obstacle while traversing an elastic edge causes the roadmap planner to fail exactly as its non-elastic counterpart would. Since re-planning is limited to local deformation of the current trajectory, the approach cannot cope with topological changes in the roadmap. Worse yet, after failure, the configuration of the robot does not lie on any of the nodes of the roadmap graph, nor on its undeformed edges. Therefore, a single query planner must be invoked to find a feasible path back to a node of the roadmap graph, and this could be problematic in a dynamic environment.

All approaches based on the potential field idea use local information from the workspace, and transform it into motor commands according to some heuristics. It is therefore not surprising that these approaches excel at fast, reactive obstacle avoidance while they have trouble with global planning tasks. Accordingly, potential field approaches have become popular in the context of safety and human-robot interaction [De Santis et al., 2007; Dietrich et al., 2011; Stasse et al., 2008; Sugiura et al., 2007]. In these applications a potential filed to attract the robot to the goal, is not defined. Instead, in the absence of influence from obstacles and joint limits, some other planner/controller system is allowed to operate freely.

To relate this discussion back to the topic of adaptively building knowledge of feasible actions from exploratory behavior: Reactive control may seem contrary to exploration, however it is in fact complimentary. An autonomous planner/controller will inevitably find danger in the form of unwanted collisions or

encounters with other unforeseen constraints, and if it should be robust to such occurrences, the capacity to react appropriately is required.

## 1.7 Reinforcement Learning

Path planning and reactive control are predicated on two different sets of assumptions, which conflict to some extent. Nevertheless, they are in many ways complementary, and in this work, I propose combining the two within the Reinforcement Learning (RL) framework [Sutton and Barto, 1998; Kaelbling et al., 1996], which allows an *agent* in an *environment* to learn a *policy* to maximize some sort of reinforcement or *reward*. These concepts are sufficiently abstract and general that RL can be implemented in terms of different representations and applied to different problem domains.

### 1.7.1 Model-Based RL

Early RL systems relied on a stochastic model of the dynamics of the environment [Bellman, 1952, 1957; Sutton, 1990; Bertsekas and Tsitsiklis, 1989, 1996], comprising *states*, $s$ and *actions*, $a$. Each state captures some relevant properties of the environment, and the actions are defined such that they cause the environment to transition from one state to another. The state transition from $s$ to $s'$ given the execution of an action $a$ is typically expressed as $(s, a, s')$, and with each such state transition is associated the probability, $P^a_{ss'}$, of observing $s'$ after being in $s$ and executing $a$ as well as the reward, $R^a_{ss'}$, received as the result of that occurrence. Each state, $s$, or state-action pair $(s, a)$ can be thought of as having a *value*, $V(s, a)$, defined as cumulative, future, discounted reward by the well known Bellman equation (eq. 1.15).

$$V(s,a) = \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma \underset{a'}{\text{argmax}} V(s', a')] \tag{1.15}$$

The parameter $0 \leq \gamma \leq 1$ controls the extent to which future rewards are discounted, and the notation $max_{a'} V(s', a')$ means the most valuable state-action pair available to the agent when the environment is in $s'$. This recursively defined value function is typically computed using dynamic programming (DP), and the optimal policy simply ascends its gradient. In practice, model-based RL is of particular interest when the dynamics of the environment are known a priori.

### 1.7.2 Model-Free RL

More recently, variants of RL have been developed [Sutton, 1988; Watkins, 1989; Watkins and Dayan, 1992], which can learn policies directly from the reward signal without the need to first model the environment. Such approaches, known as *model-free* or *temporal difference* (TD) methods, retrace the agent's steps, propagating value along the recent state-action history. The update for one-step Q-Learning (eq. 1.16) is a simple example of this idea, which like the above model-based update (eq. 1.15) places value on state-action pairs.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[R_{t+1} + \gamma \underset{a}{\text{argmax}} Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{1.16}$$

Here, $Q$ is the value function, equivalent to $V$ above, $t$ is a discrete time index, and $0 \leq \alpha \leq 1$ is the learning rate, which must be tuned to the application at hand and can cause instability. Such TD methods essentially learn the dynamics of the environment and the policy simultaneously, and accordingly they typically require more exploration on the part of the agent than do model-based methods. Updates are only performed locally in the regions of the state-action space, which have actually been explored by the agent. Thus, TD methods can learn effective policies that exploit only part of the environment, which is useful if the complete environment is intractably large. However, since the learned environmental dynamics are embedded in the policy, they cannot be reused. Thus, when applied to a new RL problem in the same environment, TD methods must start their extensive exploration process again, from scratch.

### 1.7.3 Policy Gradient Methods

The notion of a value function is not essential to RL. Policy gradient methods [Williams, 1992; Sutton et al., 2000; Peters et al., 2003; Peters and Schall, 2008], which have also been studied extensively here at IDSIA [Rückstieß et al., 2008a,b; Sehnke et al., 2008, 2010a,b; Grüttner et al., 2010; Wierstra et al., 2010], optimize some kind of a parametric policy representation with respect to the reinforcement/reward signal. Importantly, states and actions are not explicitly defined, and thus policy gradient methods avoid some of the problems commonly associated with value function based methods, such as partial observability. Moreover, the policy is typically optimized according to some kind of gradient ascent/decent, which offers better convergence guarantees than do the value function based RL algorithms.

Policy gradient methods for RL are applicable wherever a parametric policy can be defined, such that gradient decent is likely to work. One excellent example of such a problem domain is so called *imitation learning*, where a prototype motion such as a baseball swing is demonstrated to the robot (for example by forcing the manipulator externally and recording the joint angles), and the RL optimizes the motion to maximize say, the speed with which the baseball is batted away.

Despite being among the classes of learning algorithms most successfully applied in robotics [Peters et al., 2003; Peters and Schaal, 2008b], policy gradient methods are ill-suited to planning motions for a manipulator. Due to the highly non-linear mapping between the configuration space, $C$ and the workspace, $W$, it is exceedingly difficult to design a single parametric policy that works well over large regions of $C$ and $W$. Additionally, the constraints imposed by obstacles in $W$ create infeasible regions of $C$, altering its topology in ways that are counterintuitive and difficult to compute. For both of these reasons, a single parametric policy makes a poor path planner, even after it is refined through RL.

### 1.7.4 Artificial Curiosity

An RL agents needs to *explore* its environment. Undirected exploration methods [Barto et al., 1983], rely on randomly selected actions, and do not differentiate between already explored regions and others. Contrastingly, directed exploration methods can focus the agent's efforts on novel regions. They include the classic and often effective *optimistic initialization*, go-to the least-visited state, and go-to the least recently visited state.

Artificial Curiosity (AC) refers to directed exploration driven by a world model-dependent value function designed to direct the agent towards regions where it can learn something. The first implementation [Schmidhuber, 1991a] was based on an *intrinsic reward* inversely proportional to the predictability of the environment. A subsequent AC paper [Schmidhuber, 1991b] emphasized that the reward should actually be based on the *learning progress*, as the previous agent was motivated to fixate on inherently unpredictable regions of the environment. Subsequently, a probabilistic AC version [Storck et al., 1995] used the well known Kullback-Leibler (KL) divergence [Lindley, 1956; Fedorov, 1972] to define non-stationary, intrinsic rewards reflecting the changes of a probabilistic model of the environment after new experiences. In 2005 Itti & Baldi [Itti and Baldi, 2005] called this measure *Bayesian Surprise* and demonstrated experimentally that it explains certain patterns of human visual attention better than previous approaches.

Over the past decade, robotic applications of curiosity research have emerged in the closely related fields of Autonomous Mental Development (AMD) [Weng et al., 2001] and Developmental Robotics [Lungarella et al., 2003]. Inspired by early child psychology studies [Piaget and Cook, 1952], they seek to learn a strong base of useful skills, which might be combined to solve some externally posed task, or built upon to learn more complex skills.

Curiosity-driven RL for developmental learning [Schmidhuber, 2006] encourages the learning of appropriate skills. Skill learning can be made more explicit by identifying learned skills [Barto et al., 2004] within the option framework [Sutton et al., 1999]. A very general skill learning setting is assumed by the PowerPlay framework, where skills actually correspond to arbitrary computational problem solvers [Schmidhuber, 2013; Srivastava et al., 2013].

High-dimensional sensory spaces, such as vision, can make traditional RL intractable, however recent work has demonstrated that AC can help RL agents cope with large sensory spaces by helping them to explore efficiently. One such curious agent, developed recently at IDSIA, learns to navigate a maze from visual input [Luciw et al., 2011] by predicting the consequences of its actions and continually planning ahead with its imperfect but optimistic model. Similarly, the Qualitative Learner of Action and Perception (QLAP) [Mugan and Kuipers, 2012] builds predictive models on a low-level visuomotor space. Curiosity-Driven Modular Incremental Slow Feature Analysis [Kompella et al., 2012] provides an intrinsic reward for an agent's progress towards learning new spatiotemporal abstractions of its high-dimensional raw pixel input streams. Learned abstractions become option-specific feature sets that enable skill learning.

### 1.7.5 Developmental Robotics

Developmental Robotics [Lungarella et al., 2003] seeks to enable robots to learn to do things in a general and adaptive way, by trial-and-error, and it is thus closely related to Autonomous Mental Development (AMD) and the work on curiosity-driven RL, described in the previous section. However, developmental robotic implementations have been few.

What was possibly the first AC-like implementation to run on hardware [Huang and Weng, 2002] rotated the head of the SAIL robot back and forth. The agent/controller was rewarded based on reconstruction error between its improving internal perceptual model and its high-dimensional sensory input.

AC based on learning progress was first applied to a physical system to explore a playroom using a Sony AIBO robotic dog. The system [Oudeyer et al., 2007] selects from a variety of pre-built behaviors, rather than performing any

kind of low-level control. It also relies on a remarkably high degree of random action selection, 30%, and only optimizes the immediate (next-step) expected reward, instead of the more general delayed reward.

Model-based RL with curiosity-driven exploration has been implemented on a Katana manipulator [Ngo et al., 2012], such that the agent learns to build a tower, without explicitly rewarding any kind of stacking. The implementation does use pre-programmed *pick and place* motion primitives, as well as a set of specialized pre-designed features on the images from an overhead camera.

A curiosity-driven modular reinforcement learner has recently been applied to surface classification [Pape et al., 2012], using a robotic finger equipped with an advanced tactile sensor on the fingertip. The system was able to differentiate distinct tactile events, while simultaneously learning behaviors (how to move the finger to cause different kinds of physical interactions between the sensor and the surface) to generate the events.

The so-called hierarchical curiosity loops architecture [Gordon and Ahissar, 2011] has recently enabled a 1-DOF LEGO Mindstorms arm to learn simple reaching [Gordon and Ahissar, 2012].

Curiosity implementations in developmental robotics have sometimes used high dimensional sensory spaces, but each one, in its own way, greatly simplified the action spaces of the robots by using pre-programmed high-level motion primitives, discretizing motor control commands, or just using very, very simple robots. We are unaware of any AC (or other intrinsic motivation) implementation, which is capable of learning in, and taking advantage of a complex robot's high-dimensional configuration space.

Some methods learn internal models, such as hand-eye motor maps [Nori et al., 2007], inverse kinematic mappings [D'Souza et al., 2001], and operational space control laws [Peters and Schaal, 2008a], but these are not curiosity-driven. Moreover, they lack the generality and robustness of full-blown path planning algorithms [LaValle, 1998; Perez et al., 2011; Latombe et al., 1996; Li and Shie, 2007].

## 1.8 Approach - A Curious Confluence

In this thesis, I introduce a curiosity-driven reinforcement learner for complex manipulators, which autonomously learns a powerful, reusable solver of motion planning problems from experience controlling the actual, physical robot.

The application of reinforcement learning to the path planning problem (or more precisely the process of embodying the agent at a sufficiently low level of

control) has allowed two approaches to be incorporated, deliberate motion planning and reactive control, which for the most part have been treated separately by roboticists until now. The integrated system benefits from both approaches while avoiding their most problematic drawbacks, and I believe it to be an important step toward realizing a practical, feasible, developmental approach to real, nontrivial robotics problems. Furthermore, the system is novel in the following ways:

1. In contrast to previous implementations of artificial curiosity and/or intrinsic motivation in the context of developmental robotics, our system learns to control many DOF of a complex robot.

2. Planning algorithms typically generate reference trajectories, which must then be passed to a controller, but my RL-based planner/controller, learns control commands directly, while still yielding a resolution complete planner. This greatly simplifies many practical issues that arise from tracking a reference trajectory and results in a lighter, faster action/observation loop.

3. Rather than relying on reactive control to generate entire motions, I only use it to implement actions. The RL agent (planner) composes sequences of such actions, which interpolate a number of states distributed throughout the configuration space. Thus the resolution completeness of the motion planning is preserved, and its robustness is improved by the added capacity of each action to react to unforeseen and/or changing constraints.

### 1.8.1 RL for Path Planning

I propose using RL to extend the state of the art in motion planning for manipulators. The environment is the robot itself, subject to the constraints imposed by its workspace, and the policy is a solution to a particular path planning problem. The proposed RL system should be able to cope with many different instances of the path planning problem. It is therefore advantageous to model the environment, such that the system can reuse information from one problem instance to the next. I have therefore selected model based RL as the framework of the learning system. The notion of state is based on the configuration of the robot. Actions are sequences of control commands, intended to move the robot from one configuration to another using a reactive control approach. Thus I build reactiveness into the control system at a low level, which introduces stochasticity into the path planning process. Then to cope with this stochasticity, an intelligent agent based on RL is responsible for computing policies, which solve

motion planning problems and comprise sequences of these stochastic, reactive actions.

A critical advantage of model based RL with respect to motion planning is that a policy can always be computed over the entire environment, given some reward matrix. This means that once the agent finds a reward, the learning system can plan motions to get that reward at any time, irrespective of the initial state of the robot. Thus, a single policy/plan can in principal take the robot from any pose to the target pose, via a sequence of feasible intermediate poses. A model based RL system can therefore be a reasonable path planning solution.

### 1.8.2 Why Has No One Done This?

The direction of research that I have proposed thus far begs the question: Why have path planning and reactive control remained separate in the robotics literature for so long? In my opinion, the answer is that they are predicated on two different sets of assumptions, and are therefore not very compatible with one another.

Path planning is essentially a difficult search problem, and as such, in my opinion, it belongs to the field of theoretical computer science. Approaches to path planning are formulated in nice, tidy vector spaces, which are free from noise and uncertainty, and they are typically evaluated in kinematic simulation in terms of algorithmic concepts like complexity, soundness, and (resolution) completeness. The final answer (a trajectory) is of primary importance, and the transient behavior of the algorithm, which may require a great deal of runtime, is not usually considered.

Contrastingly, robotic control (particularly 'reactive control') is much more applied. All of the physical constraints of the robot must be respected, including its dynamics and its noisy, uncertain sensory-motor apparatus. Time is therefore of the essence, and the transient response of the robot is critical. In fact, control approaches are evaluated almost entirely according to the properties of that transient response, such as rise time, settling time, and smoothness.

My insight in the work presented here, is that these two different ways of thinking about robotic motion, while not compatible with one another directly, are absolutely complimentary. Moreover, it is RL that affords the abstract formalism necessary to integrate these two, previously disparate approaches to motion synthesis.

### 1.8.3 What Does It All Mean?

My approach, using RL to plan motions as sequences of actions implemented by dynamical systems facilitates unprecedented robustness.

All previous approaches to path planning for robotics could fail in the sense that a planned trajectory, which was thought to be feasible, turns out not to be. The robot must be stopped and left stationary in some intermediate position, while the expensive search for a feasible motion begins anew. In this condition, avoidance of a moving obstacle is impossible.

My planner, on the other hand, cannot fail as described above. Because the robot is constantly under the control of reactive actions, it can always avoid/pursue dynamic obstacles/goals regardless of how well the current plan achieves its intended goal. Motion planning and re-planning take place naturally, on an as-needed basis, as the actual, physical robot moves about in its workspace trying to achieve goals.

My work represents a conceptual departure from the prevalent thinking in robotic motion planning. It is the first approach I am aware of that encompasses the capacity to both plan and react. Therefore I was unable to devise a direct comparison to either more traditional path planners or purely reactive controllers. Such comparisons are usually made on particularly tricky instances of motion synthesis problems, and my work is not aimed at solving particular hard problem instances, but rather at solving (or attempting to solve) arbitrary problem instances robustly. Therefore, in this dissertation, I do my best to explain the ways in which my approach differs from the established state of the art, and I present a number of novel experiments, in which the iCub humanoid learns motion planners through an autonomous developmental process of intrinsically motivated experimentation, which can last for days of run time. These are the first such experiments i am aware of in developmental robotics, and they are only made possible by the unprecedented robustness of my RL based planner/controller.

## 1.9 Related Work

Reaching is a complex and multifaceted topic. To thoroughly review all of the related literature would require thorough consideration of both child development and neuroscience, which is beyond the scope of this dissertation. I will however briefly comment on both as they relate to my work.

### 1.9.1 Child Development

My approach to robotic reaching is decidedly developmental, and its inspiration (as the name suggests) comes from biology. A fair amount of attention has been paid to reaching in human infants over the years [Thelen et al., 1993; Berthier et al., 1999; Needham et al., 2002; von Hoftsen, 2004; Hespos et al., 2009; Berthier, 2011], and I had the good fortune to be exposed to some of that work throughout the course of the EU project, Intrinsically Motivated Cumulative Learning Versatile Robots (IM-CLeVeR). As an engineer I find such work interesting, as it gives some insight into *what* my control system should do. However, I do not find the work in this field, at least that with which I am familiar, to be particularly instructive as to *how* it should be done. For me, biological research on reaching remains a sort of nebulous source of inspiration and a benchmark, which sets the bar very high, for what future generations of robotic controllers should be capable of.

In my view the single feature exhibited by infants in reach learning, which is most deficient in robotics, is robustness. My intuition tells me that for any of the clever algorithms in the AI literature to ever work the way hollywood seems to think they should, they will need run robustly in the real world for a long time. That is my inspiration from child development. The rest of my approach is entirely motivated by and predicated on technical considerations.

My efforts toward creating a practical, developmental approach to reaching on a humanoid robot is among the first ever tried, and I think the field of humanoid robotics' lack of maturity limits the extent to which it can feed information back to the biological side of reaching. However one way that it may be able to do so already is in terms of experiment design. The difficulties we encounter in humanoid robotics may be able to help developmental psychologists to better identify/specify particular aspects of human behavior, which merit further investigation.

### 1.9.2 Neuroscience

The idea is popular in the neuroscience community that the central nervous system in humans uses some kind of forward model to (help) generate motions [Miall and Wolpert, 1996]. This has led to a great many approaches in robotics, by which various kinematic models and mappings are learned from experimentation [Schenck et al., 2003; Sun and Scassellati, 2004; Nori et al., 2007; Chinellato et al., 2009, 2011] that is sometimes called 'motor babbling.' Some model learning approaches are 'biologically plausible' and yield simple models, which

are typically limited to simple kinematics. Most of those cited above however, employ strong machine learning techniques, which are not biologically plausible but are capable of learning the kinematics of complex manipulators. None such model learning methods that I am aware of however, are able to do motion planning in the sense of obstacle avoidance. Therefore such model learning approaches are not really comparable with the planner/controller I develop in this thesis.

My work assumes that a complete forward kinematic model of the robot is known, along with its geometry. I demonstrate that this is a good assumption in chapter 2 by showing that such a model is easy to define and cheap to compute. The only practical benefit to learning the kinematic robot model (that I am aware of) is to account for differences between an analytical model and actual hardware. However this comes at the price that the learned model is not complete.

For example, a hand-eye model, learned by a humanoid from vision, does not typically account for the elbow, which is hard for the robot to see and can easily collide with the hip. During learning the range of motion of the arm must be carefully constrained to avoid elbow/hip collisions. Then the resulting learned model/map only covers a subset of the hand's true range of motion.

For my purposes (coarse motion planning with obstacle avoidance), differences between the analytical kinematic model and the actual hardware were negligible. In the future however, implementations such as mine might benefit from learning the kinematic model online. Still, it would be difficult if not impossible to avoid (self) collisions while still learning a complete kinematic model.

## 1.10 Summary/Outline

The structure of the remainder of this dissertation reflects my engineering approach to the research I have conducted. It begins by motivating and describing the core of the experimental setup, my kinematic robot model. Then it moves into some early applications of that setup, in terms of offline motion planning, and online re-planning via a switching controller. This experimental work, or more precisely, the shortcomings thereof, motivates a departure from the switching controller in favor of a more robust solution, formulated around second order dynamical systems. This results in a robust low-level control system, which allows traditional roadmap based motion planning to be reformulated in terms of RL.

In the latter part of the dissertation, results of two different batches of RL experiments are presented. The first set of experiments deals with learning a stochastic version of a roadmap planner, while the second set exploits such a planner to learn reaches to arbitrary workspace targets. Following, I will briefly describe the contents of each chapter, highlighting my novel contributions.

### Chapter 2 - An Egocentric Robot Model

The need for a robot model is motivated by the claim that robots are by nature distributed systems and therefore, some kind of mechanism to estimate/represent their complete state is quite useful. The functional requirements of such a model are then outlined according to my chosen application, manipulation. Then, I describe my implementation of a kinematic robot model, which is novel in its ability to be applied online to facilitate reactive control as well as offline for preemptive motion planning. Finally I present several experiments, demonstrating the robustness and scalability of the model.

### Chapter 3 - Task Relevant Roadmaps

My kinematic model is applied to offline roadmap planning in a collaborative project [Stollenga et al., 2013], which I undertook with Marijn Stollenga and Leo Pape. The novel aspect of the work is the application of Natural Evolution Strategies (NES) [Glasmachers et al., 2010], a powerful black box optimization algorithm, recently developed at IDSIA, to the construction of roadmap graphs for motion planning. Some of the resulting motions can be viewed in our short film, which won *Best Student Video* at the Association for the Advancement of Artificial Intelligence (AAAI) video competition in 2013 (`http://www.youtube.com/watch?v=N6x2e1Zf_yg`).

### Chapter 4 - Real Time Collision Response

My kinematic model is applied online, to protect the iCub from colliding with things in its workspace, including itself, stationary objects such as the work table, and other robots that move. Collision response is implemented by a switching controller, which turns out to be rather impractical, still among the experiments presented in this chapter are to my knowledge the first ones to learn/maintain a roadmap data structure for motion planning from a real piece of hardware exploring in realtime [Frank et al., 2012a,b], as well as the first experiments in which a precise industrial robot 'teaches' spatial perception to an imprecise humanoid robot in a shared workspace [Leitner et al., 2012b, 2013c].

Chapter 5 - MoBeE 2.0

Problems with the switching controller, related to noise and hysteresis made the approach fairly impractical, and this motivated a major reworking of the MoBeE system. The 2.0 implementation, which is based on dynamical systems facilitates robust collision avoidance. Moreover, it introduces a richer stochasticity to the low level control, as the dynamical systems can in principal settle anywhere. Thus, while the collision avoidance of MoBeE 2.0 is not terribly novel in and of itself, it does indeed represent the state of the art, and it facilitates a much richer and more interesting RL for planner learning than was possible under the control system in chapter 4.

Chapter 6 - An RL Agent for MDP Roadmap Planning

The robust, low-level control provided by the MoBeE 2.0 implementation allows roadmap planning to be reformulated around a Markov Decision Process (MDP) as opposed to the usual graph. This reformulation is my primary theoretical/methodological contribution to the robotics community [Frank et al., 2013], and perhaps the 'bread and butter' of my doctoral dissertation. This chapter lays out the details of learning an MDP based motion planner using RL, including 'Kail' divergence, my novel reformulation of the well-known Kullbach-Leibler (KL) divergence.

Chapter 7 - Model Learning Experiments

Several sets of experimental results are presented. They begin with small state-action spaces to prove the concept and validate the implementation, and move to more larger, more ambitious experiments, which approach the scale necessary for real world deployment. The efficacy of artificial curiosity is demonstrated. Importantly, these are the first developmental learning experiments I am aware of which are capable of running robustly and autonomously for days on end, as a real physical robot learns to navigate its configuration space. Additionally, a multi-agent experiment is presented, which produces an interesting emergent behavior. The iCub becomes interested in touching the work table, purely through the effects of intrinsic motivation.

Chapter 8 - Learning To Reach

Finally, the MDP motion planner is exploited to learn to reach to arbitrary workspace targets in a cumulative way. In contrast to previous planning approaches, the

system can recall how to circumnavigate targets/obstacles seen previously in order to put the hand in a sensible pre-reach position and eventually execute a dynamic reach. Although the performance of the integrated reach planning system is not yet perfect, it unequivocally does the following:

1. Run robustly and autonomously for arbitrary long periods of time.
2. Learn incrementally and cumulatively about different reaching problem instances as they are presented, one by one.
3. Produce many 'good' reaches, often pulling the hand out from behind the target in order to access a sensible pre-reach position.
4. Locate the best pre-reach poses in the state space over the whole set of reach problems.

Chapter 8 - Conclusion

The final chapter of my dissertation contains some of my more conceptual conclusions regarding motion synthesis for complex manipulators. Additionally, I reflect on my accomplishments during my time at IDSIA and discuss the direction of future work, which could improve the performance of my reach learning approach.

# Chapter 2

# An Egocentric Robot Model

In section 1.6.1 I claimed that in order to develop a complete picture of the state of a humanoid robot in its workspace, some kind of model is required to facilitate sensory fusion between data perceived in the workspace (vision) with that perceived in configuration space (proprioception).

Prevalent approaches to robot modeling tend to fall into two categories. There are flexible, robot independent, do-everything modeling environments, such as the Open Robotics Automation Virtual Environment (OpenRAVE) [Diankov and Kuffner, 2008], and there are lean, robot specific models [Dietrich et al., 2011]. The former tend to assume a 'plan first, act later' paradigm, while the latter tend to be difficult to separate from hardware for offline use. My novel kinematic robot model, on the other hand, is robot independent, easily reconfigurable, and suitable for both offline search *and* online reactive control applications.

## 2.1 The Lack of Skin

Due to challenges in tactile sensing, there does not yet exist a serviceable robotic skin, which can cover a manipulator's entire 'body,' deforming around it is it moves. Therefore, when it comes to re-planning motions around unforeseen obstacles, complex robots are incredibly information poor.

Consider that a humanoid has a very large number of controllable DOF (the iCub has 41), and operates in 3D space where an object has 6 DOF. Still, it only has an array of cameras or range finders, which capture narrow, 2 and 3D projections of the state of the high dimensional humanoid-world system. Without access to tactile information, or what to the robot 'feels' is feasible, moving around in the workspace in an 'intelligent' manner is a very hard job

indeed.

This motivates a parsimonious, egocentric, kinematic model of the robot/-world system to simulate tactile feedback for real time motion re-planning. In addition to simulated tactile feedback, the model should provide access to a cartesian operational space, in which task relevant states, state changes, cost/objective functions, and rewards can be defined. By computing forward kinematics, and maintaining a geometric representation of the 3D robot/world system, the model can not only facilitate fusion of sensory signals native to the configuration and operational spaces, but it can also provide a useful and general state machine, which does not arise naturally from the 'raw' sensory data.

## 2.2 The iCub as a Distributed System

A modern robot is an electromechanical system, composed of sensors and actuators. Each actuator is potentially a separate physical piece of hardware, such as a motor. Therefore, to assemble a complete state of the robot, comprising the states of each actuator, is a distributed systems problem.

Perhaps not surprisingly, throughout the history of robotics research, a great deal of effort has gone into hacking bits together to solve the distributed systems problem and get good communication between sensors and actuators.

Roboticists have often been compelled to 'reinvent the wheel', continually re-implementing necessary software components as new hardware becomes available or other software components change. In recent years, the topic of software engineering has received increased attention from the robotics community, and 'robotics platforms', such as Yet Another Robot Platform (YARP) [Metta et al., 2006; Fitzpatrick et al., 2008], Robot Operating System (ROS) [Quigley et al., 2009], and Microsoft Robotics Studio (MSRS) [Jackson, 2007], have gained widespread popularity. Not only do these middleware solutions abstract away the details of sensors and actuators, they offer simple network communication from virtually any language on MacOS, Windows or Linux. Robots can be controlled with relative ease by one or more distributed applications running on a cluster. By providing hardware abstraction, YARP, ROS, and MSRS have drastically improved the efficiency with which experimental robots can be programmed. In the process of developing behaviors, one would do well to follow the example set by these projects, and develop modular behavioral components around abstract interfaces. In the spirit of these open-source projects, a goal of mine throughout my doctoral studies has been to develop solid, reusable software that can help facilitate future robotics research.

Much of the work leading up to this dissertation was done on the iCub humanoid robot, which is accessed via 'YARP ports.' The ones relevant to the work presented here are listed in table 2.1. As far as the software infrastructure discussed here is concerned, the ports *are* the iCub robot.

The ports are specific to 'body parts,' each of which comprises a number of motors, or provides read access to sensors, such as the cameras, left and right. Ports ending in '/state:o' stream motor encoder positions, those ending in '/cmd:i' accept streams of motor commands (position, velocity, and force control are supported), and those ending in '/rpc:i' can answer various queries over Remote Procedure Call (RPC).

| iCub YARP Ports |
|---|
| /icub/head/state:o |
| /icub/head/cmd:i |
| /icub/head/rpc:i |
| /icub/torso/state:o |
| /icub/torso/cmd:i |
| /icub/torso/rpc:i |
| /icub/right_arm/state:o |
| /icub/right_arm/cmd:i |
| /icub/right_arm/rpc:i |
| /icub/left_arm/state:o |
| /icub/left_arm/cmd:i |
| /icub/left_arm/rpc:i |
| /icub/cameraL |
| /icub/cameraR |

Table 2.1. iCub robot YARP Ports

## 2.3 Developing a Kinematic Model

The process of designing the robot model began with the following functional requirements:

1. Speed - The model must be fast enough represent the state of the robot in real time, such that it can facilitate reactive control. For the iCub, the highest frequency sensory signal is the stream of motor encoder positions,

which is about 100Hz, so the model must be able to compute robot states at least that fast.

2. Flexibility - It must provide a usable interface for robot modeling, such that models can be modified to keep up with changing robot hardware and task requirements, and entirely different robots can be modeled.

3. Versatility - The model should not only provide reactive control when used in conjunction with hardware, but also stand alone and function as an oracle for traditional offline planning.

In light of these requirements, I decided early on to pursue a kinematic/geometric model, which neglects dynamics. Such a model provides the speed necessary to exhaustively search the configuration space of high DOF robots such as the iCub. Moreover, since I am primarily interested in robust/adaptive reach planning, highly dynamic motions are not of primary importance. Since modeling the dynamics of complex, cable driven robots like the iCub analytically can be problematic, and since my model is designed to run alongside the hardware in real time anyway, I decided that if a dynamic model should become necessary during the course of my research, I would learn one empirically.

Robot poses, $q \in C$, can be provided either by hardware (for reactive control) or by a sampling algorithm (for planning), and the model's primary responsibility is to carry out forward kinematics and collision detection computations and to broadcast the resulting information, including collision pairs, interference volumes and Jacobian matrices.

To keep the computations as efficient as possible, the model supports kinematic chains and trees, but not loops, and hierarchical pruning is employed to reduce the number of collision pairs to be tested. Moreover, objects in the robot's workspace are not collision-tested against one another, so the approximate complexity of collision detection is $O(n^2 \cdot m)$ where $n$ is the number of geometries in the robot model and $m$ is the number of objects in the environment. The robot kinematics and geometry, as well as pre-defined workspace configurations, are be specified via XML similarly to how it is done by the Open Robotics Automation Virtual Environment (OpenRAVE) [1] [Diankov and Kuffner, 2008].

[1] I initially put some effort into adapting OpenRAVE to my purposes, rather than building a robot model from scratch, but was dissuaded first by the fact that OpenRAVE works with ROS as opposed to YARP, making it unnecessarily difficult to interface with the iCub, and second by OpenRAVE's assumption of a 'plan first execute later' control paradigm, which I have tried to avoid.

The kinematic model is implemented in C++ as a static library, which depends only on two widely available, platform independent, free and open source libraries, Qt (for threading, OpenGL and an XML parser) and the Software Library for Interference Detection (SOLID), which is highly optimized and supports primitives, Minkowski sums [Schneider, 2013], and polyhedra. The kinematic model is designed to be compiled and used as such, entirely without YARP, to facilitate fast, offline algorithmic planning. To provide the required online functionality, I have wrapped the kinematic model with a thin YARP layer to provide communication with any YARP compatible robotic hardware.

### 2.3.1 Zero Reference Position Kinematics

The de facto standard notation for manipulator kinematics is known by the names of its inventors Jaques Denavit and Richard Hartenberg. The Denavit-Hartenberg (DH) convention [Denavit and Hartenberg, 1955] facilitate the systematic assignment of reference frames to the links of a kinematic chain. Essentially, the DH convention requires that a reference frame is chosen for each joint and represented by a right-handed coordinate system with principal directions $x_i$, $y_i$ and $z_i$, where the directions $z_i$ are parallel to the joint axes and the directions $x_i$ are parallel to the common normals, $z_{i-1} \times z_i$.

The DH convention sounds simple enough, but it can be quite confusing where geometric modeling is concerned. Each geometry must be defined within the local frame of reference, and each frame is rotated with respect to the previous one. Moreover the origins of the coordinate systems that define the frames may lie well outside of the robot's 'body,' which is counterintuitive for many people, including myself.

To facilitate easy and rapid prototyping of robot models, I have shunned the popular DH convention, instead basing my XML specification on a far more intuitive standard, Zero Reference Position (ZP) notation [Gupta, 1986], which has been used recently, not only to model robots but also other complex kinematic linkages, such as proteins [Kazerounian et al., 2005b,a].

As the name implies, ZP notation defines the transformations between reference frames according to their relative states with the linkage in some convenient position, in which all joint variable values are defined to be zero. Essentially the notation requires a set of joint axis direction vectors $u_i$, and a set of body vectors $b_i$, to represent the displacements associated with the links. These are all defined in the global coordinate system, and $b_i$ are constrained such that $(0,0,0)+b_0$ equals a point on the first joint axis, $(0,0,0)+b_0+b_1$ equals a point on the second joint axis, and so on.

If the body vectors are required to be the set of mutual perpendiculars between the joint axes, then ZP notation can yield a unique description of the linkage kinematics, as does the DH convention, which is elegant mathematically speaking but not necessarily desirable. By giving the user control over $b_i$, they can be chosen such that they approximate the robot's body skeleton. If with each $b_i$ is associated a radius, then the skeleton gains volume, and a useful geometric model of the robot is obtained without the need to painstakingly locate and orient each geometry in its local frame of reference.

In accord with ZP notation, I developed an XML specification for modeling kinematic trees with attached geometries. The key tags, with their key parameters are as follows:

1. <link x="float" y="float" z="float" radius="float" field="float"> represents a displacement along a link defined by the vector $(x,y,z)$ and can be given a solid volume, *radius*, and/or a force field[2] volume, *field*.

2. <joint x="float" y="float" z="float" minPos="float" maxPos="float" radius="float" field="float"> represents a joint with axis parallel to the vector $(x,y,z)$, minimum and maximum angular positions measured in degrees, *minPos* and *maxPos*, and optional solid and force field volumes, *radius* and *field*.

3. <bodypart name="string"> defines a group of motors to be controlled by a particular YARP (/state:o) port.

4. <motor minPos="float" maxPos="float"> maps a motor encoder interval onto one or more joint angle intervals.

5. <constraintList> defines a group of linear constraints in conjunctive normal form.

6. <constraint a="float vector" q="int vector" b="float"> represents a linear constraint, $a\dot{q} < b$, where $a$ is a vector of free parameters, $b$ is a free parameter, and $q$ is a vector of motor indices.

7. <marker name="string"> names a point of interest on the robot's body, such as an end effector, at which queries can be made for workspace coordinates, Jacobian matrices and the like.

8. <sphere radius="float" px="float" py="float" pz="float" field="bool"> defines a sphere of radius, *radius*, which can be appended to the robot's body,

[2]The function of force fields is described in chapter 5.

with its center at the position $(px, py, pz)$ relative to the parent link or joint, and which may be solid or a force field, depending on *field*.

9. <cylinder radius="float" height="float" px="float" py="float" pz="float" hx="float" hy="float" hz="float" field="bool"> defines a cylinder of radius, *radius*, and height, *height*, which can be appended to the robot's body, with its center at the position $(px, py, pz)$ relative to the parent link or joint, oriented with its axis of symmetry parallel to $(hx, hy, hz)$, and which may be solid or a force field, depending on *field*.

10. <box height="float" width="float" depth="float" px="float" py="float" pz="float" hx="float" hy="float" hz="float" angle="float" field="bool"> defines a box of dimensions, *height*, *width*, *depth*, which can be appended to the robot's body, with its center at the position $(px, py, pz)$ relative to the parent link or joint, oriented with its height axis parallel to $(hx, hy, hz)$, and which may be solid or a force field, depending on *field*.

This XML specification has facilitated the prototyping of various experiments, by allowing us to quickly model different robots in different ways. A complete model of the Katana manipulator, chosen for its simplicity and readability, is shown in appendix B. A visualization of the Katana and iCub robot models is shown in figure 2.1, and the iCub model is shown next to the hardware in figure 2.2. I have spared the reader the 726 lines of XML, which comprise the iCub model, however several versions of it, which differ primarily in the complexity of the hands and the placement of markers, are available for download through the iCub software repository (http://wiki.icub.org/iCub_documentation/).

### 2.3.2 Threading and Robot State

For the online use case, in which the robot model facilitates reactive control, its first job is to provide a unified notion of robot state, which must be assembled from the asynchronous messages arriving from several YARP ports as described in section 2.2. This creates the necessity that model updates are thread safe, and that design feature carries through not only position/orientation updates but also the creation and destruction of geometries. Therefore, vision or other sensory data can dynamically drive the state representation of the robot's workspace in the model.

Thread safety is a key design feature that facilitates the reusability and versatility of the robot model in different applications, and the functionality is shown graphically in figure 2.3. In order to define a whole body state of the physical

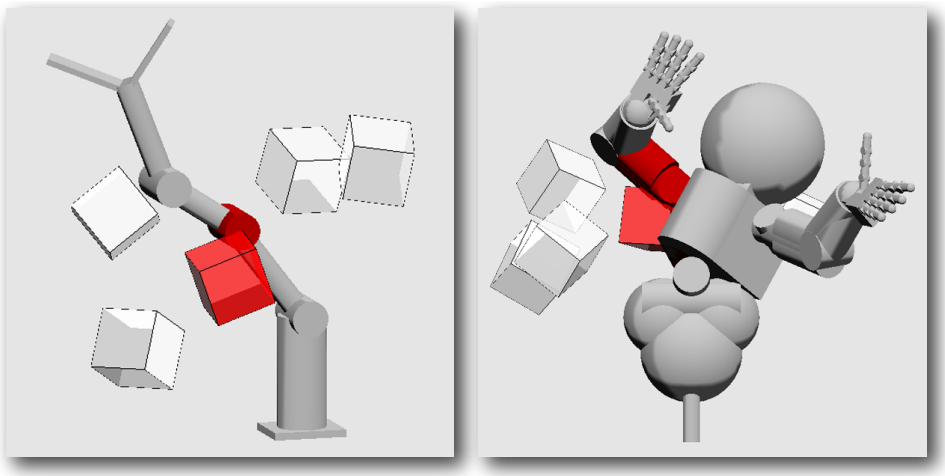

Figure 2.1. Kinematic Robot Models - The Katana arm (left) and the iCub humanoid (right) collide with random obstacles. Darkened (red) geometries are colliding.

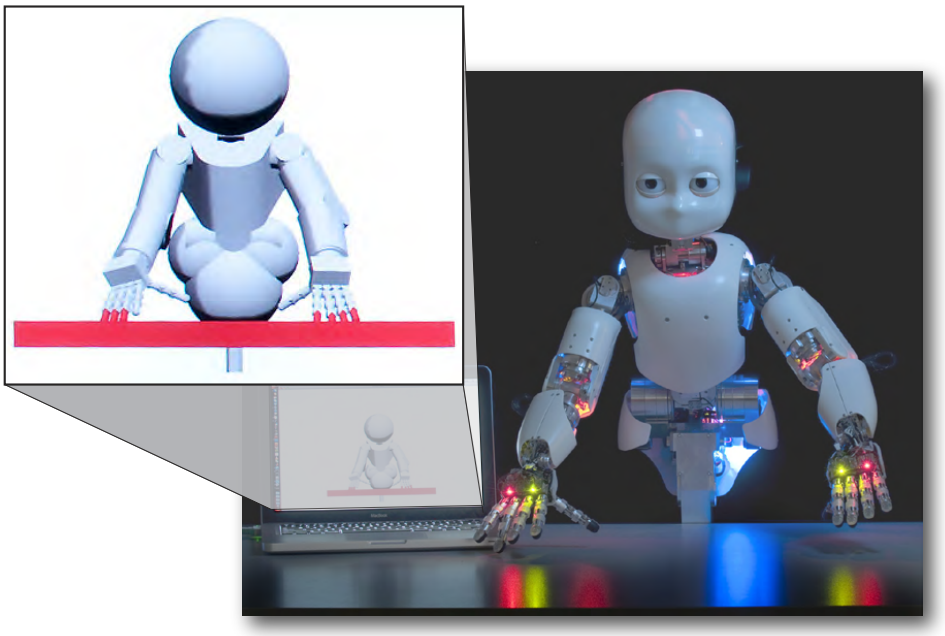

Figure 2.2. Impending Collision - The kinematic model detects impending collision between the iCub humanoid robot and a table. Darkened (red) geometries in the model (left) are colliding.

**Algorithm 1:** Floods the kinematic model thread with insertions and removals of workspace objects without any waiting.

```
FLOOD_MODEL(period,maxNumObjects) begin
    n ← 0;
    while n < maxNumObjects do
        create_object();
        m ← 0;
        for m < 1000 do
            remove_object();
            create_object();
            m ← m + 1;
        end
        n ← n + 1;
    end
end
```

robot as it is running, each body part requires a control thread, the minimal functionality of which is to read the appropriate 'state:o' port, and set the motor encoder positions of the relevant motors in the model, such that they reflect the most recent state of the hardware. Periodically, the model computes forward kinematics, updates the positions/orientations of the geometries comprising the robot, and does collision detection, finally publishing the state of the robot/-workspace system (what is colliding with what) to all listening threads, which are free to do whatever they like with this information in terms of robot control. Working under the assumption that these messages are quite frequent (100Hz on the iCub), time is not explicitly considered, and the control threads are allowed to write to the model whenever they like, provided the model is not actually in the middle of doing critical computations.

## 2.4 Model Validation: Thread Safety, Performance and Scalability

After implementing the kinematic model as described in section 2.3, and before moving on to the control side of the proposed system, I worked to validate my kinematic model experimentally, and get a feel for what kind of performance I could expect. Here I present the results of one experiment in particular, which I

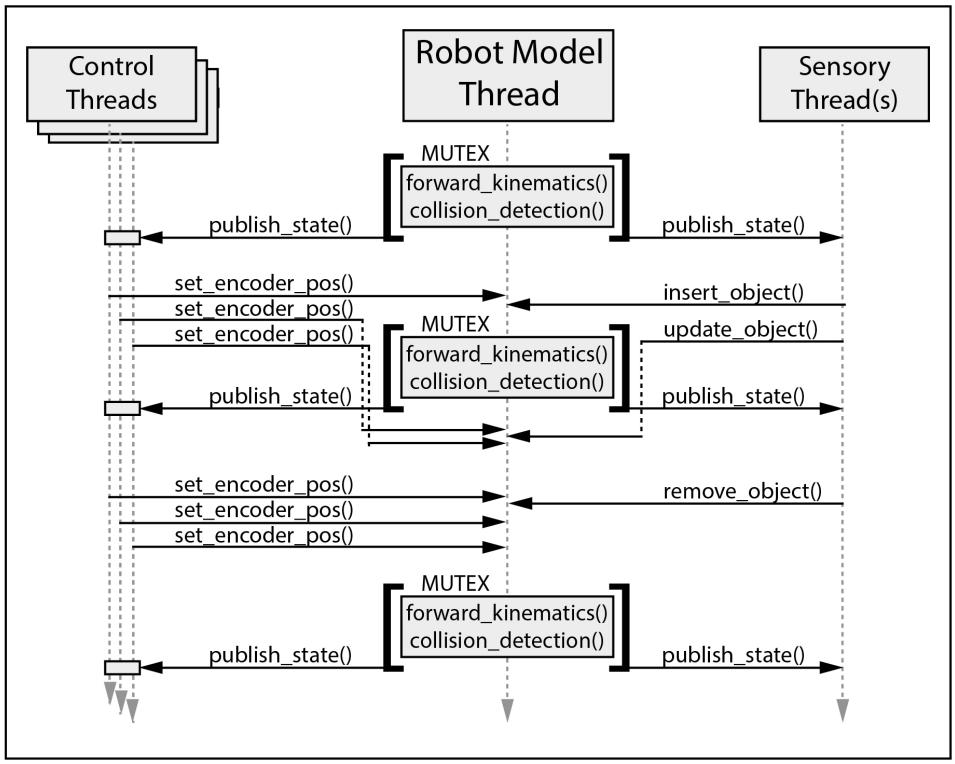


Figure 2.3. Kinematic Model Thread Design - The kinematic model is implemented around the 'critical section' design pattern. As time goes on (top to bottom), forward kinematics and collision detection are computed periodically, and the state of the robot/workspace system is published by the model thread. Sensory threads and motor control threads may insert/update/remove geometries in the robot model by inter-thread function calls, but a mutex provided by the Qt library prevents the changes from being affected during critical computations.

think is particularly instructive. It analyzes the scalability of the kinematic model with respect to the complexity of robot models and workspace configurations. Two different robots are considered, as the number of modeled objects in the robots' workspace grows. The experimental setup captures important aspects of both the offline and the online use cases.

In order to simulate the control thread(s) (figure 2.3) each joint is driven through its entire range of motion at by a simple $d$ dimensional ($d$D) oscillator.

The simulated sensory thread simply adds and removes geometries in the robots' workspace as fast as it can without any waiting, gradually increasing

the number thereof, as shown in algorithm 1. The geometries chosen for this experiment are boxes, as they are the most computationally expensive of the currently supported primitives.

The model thread is flooded by messages from the sensory thread, and periodically (every 10ms), the control thread updates the robot pose. While the the frequency of model computations (kinematics and collision detection) would normally be tuned to match the frequency of the control thread(s) (after all it is not very useful to repeat collision detection computations if the robot model is still in the same pose), for this experiment it is allowed to run as fast as it can, with no waiting.

The experimental setup allowed me to validate the mechanisms which provide thread safety, while getting an idea of the maximum runtime performance achievable for a subsequent offline application, which just tests robot poses as fast as possible.

The experiment was run on Mac OSX 10.6 on a dual-core 2.4GHz laptop with 4GB of memory, and two different robot models were used; a 6 DOF Katana arm (9 primitives), and the 41 DOF upper-body of an iCub humanoid robot (129 primitives). Figure 2.1 shows snapshots from the early stages of the running experiments, and the results are plotted in figure 2.4. Based on this experiment, I make the following claims:

1. For simple arms in simple environments, my kinematic model can keep pace with even the fastest control frequencies encountered in industrial practice.

2. It can compute hundreds of poses per second for a humanoid with hundreds of obstacles in the environment, which is adequate for most applications in developmental robotics.

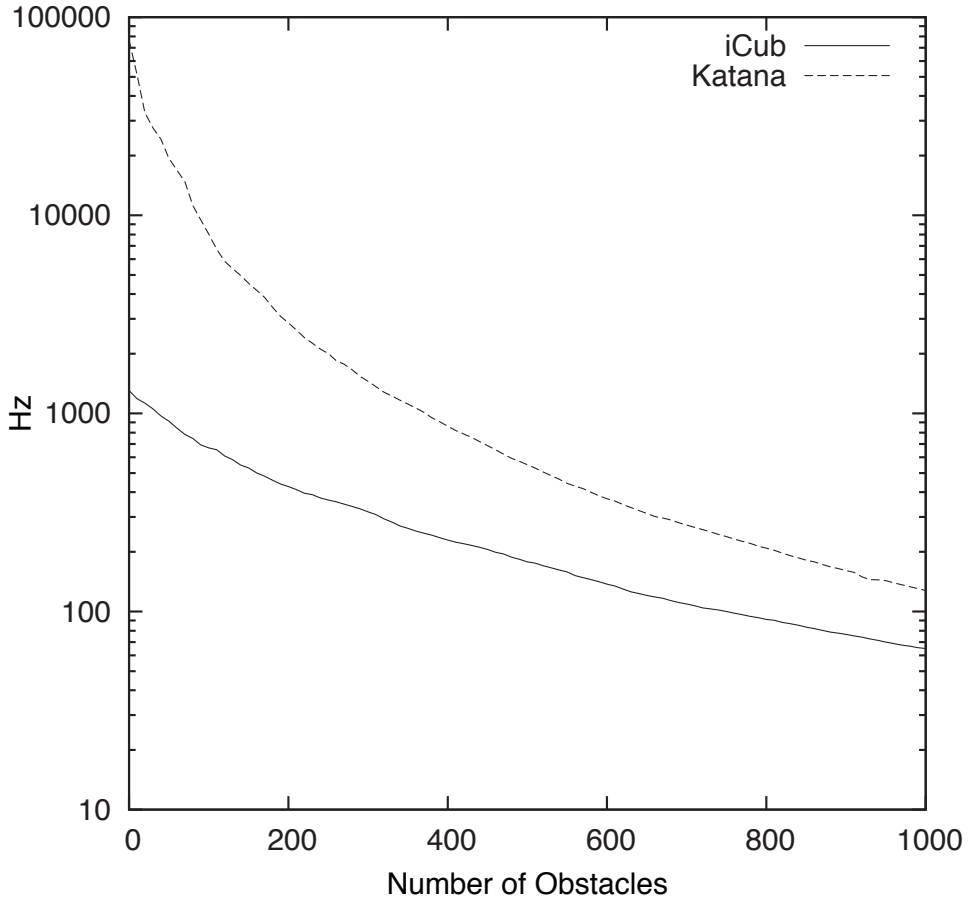


Figure 2.4. Kinematic Model Scalability - A simple robot model (Katana - 9 primitives) is compared with a complex one (iCub - 129 primitives). The curves show the number of times collision detection can be computed per second (y-axis) given a particular number of obstacles modeled in the robot's workspace (x-axis).

# Chapter 3

# Task Relevant Roadmaps

One of the first practical applications of my kinematic model was developed jointly by Marijn Stollenga and myself [Stollenga et al., 2013]. The kinematic model serves as an interference detection oracle in a traditional path planning framework, which is similar to PRM planning [Latombe et al., 1996]. However, in contrast to PRMs, our approach finds sets of robot poses by employing state-of-the-art black-box optimization on complex objective functions, defined through my kinematic model. We are able to produce Road Map data structures, which comprise complex motions embedded in high-dimensional spaces that are well suited to different tasks, Task Relevant Road Maps (TRMs). Some of the motions we have synthesized can be viewed in our short film, which won *Best Student Video* at the Association for the Advancement of Artificial Intelligence (AAAI) video competition in 2013 (`http://www.youtube.com/watch?v=N6x2e1Zf_yg`).

## 3.1 Background

A traditional approach to making a manipulator do things can be described in terms of the following three sub-problems:

1. *Inverse Kinematics:* Find a robot pose, $q_{goal} \in C$, that satisfies some operational space constraints, such as touching an object with the end effector.

2. *Motion Planning:* Find a feasible configuration-space trajectory, $Q \subset C$, which is a curve that interpolates the current pose, $q_{initial}$ and the goal-pose, $q_{goal}$.

3. *Trajectory Tracking:* Use a feedback control system to track the trajectory, $Q$, moving the robot from $q_{initial}$ to $q_{goal}$.

Our framework focuses on *inverse kinematics* and *motion planning*, and delegates *trajectory tracking* to a simple feedback controller.

We begin by solving the inverse kinematics problem using the forward kinematic model and an optimization algorithm similarly to other recent approaches [Dutra et al., 2008; Courty and Arnaud, 2008; Hecker et al., 2008]. Our solver however, Natural Gradient Inverse Kinematics (NGIK), benefits from a recent and powerful black-box optimization algorithm, called Natural Evolution Strategies (NES) [Glasmachers et al., 2010], which was developed by other members of our group and is based on the Natural Gradient [Amari, 1998]. NES is in many ways comparable to Covariance Matrix Adaptation (CMA) [Hansen et al., 2003], but is more principled and outperforms CMA on some tasks.

In light of recent ideas from planning literature, which focus the search of the configuration space to subspaces, relevant to a task [Kalakrishnan et al., 2011; Berenson et al., 2009, 2011], we too define such *task spaces*, which we try to cover by iteratively applying NGIK. The resulting sample set, comprising a family of task-related poses, is interpolated to yield a TRM.

The TRM framework allows the task-space to be defined freely by the user in terms of hard and soft constraints alike. The excellent performance of NES on the inverse kinematics problem, as well as the speed and flexibility of the purpose-built kinematic model allow us to generate TRMs to plan complex, state-of-the-art motions. We demonstrate the effectiveness of our approach in the contest of object manipulation, using the 41DOF upper body of the iCub humanoid robot.

### 3.1.1 Inverse Kinematics

One approach to inverse kinematics (IK) for a manipulator, perhaps the oldest one, is to find a closed form transformation from a frame of reference attached to the the end effector to that of the base of the robot [Lee and Ziegler, 1984]. For kinematically redundant robots, additional constraints must be applied in order that a closed form solution exists, and different ways of doing this have been investigated over the years [Hemami, 1987; Kauschke, 1996]. This approach yields a fast solver, but requires careful engineering and restricts the kinds of constraints that can be used.

Numerical optimization approaches can find suitable poses by exploiting the gradient of the forward kinematics function. This is typically done by calculating

the pseudo-inverse or transpose of the well known Jacobian matrix [Wolovich and Elliott, 1984; Goldenberg et al., 1985; Zohdy et al., 1989], which is linear, when evaluated at a particular pose. Recent work applies such methods to anthropomorphic limbs [Tolani et al., 2000] and humanoid robots [Baerlocher and Boulic, 2004]. The latter adds an efficient way to handle prioritized hard constraints.

Although these approaches are much more flexible than those yielding closed form solutions, they are sensitive to singularities and require the gradient/Jacobian to be known, which restricts the set of constraints and kinematic chains that can be represented.

Recently sampling-based methods, have tried to circumvent there problems. Such methods never explicitly calculate a gradient/Jacobian, but estimate it by sampling $q \in C$ and computing the poses of the relevant body parts, $A_i(q)$. Sampling-based methods can deal with arbitrary cost functions, making them much more flexible and robust than traditional IK algorithms.

Several sampling optimizers have been proposed recently. Simulated Annealing [Dutra et al., 2008], is very flexible but has only been used for small kinematic chains. Sequential Monte Carlo (SMC) [Courty and Arnaud, 2008] uses a non-parametric distribution of particles, but relies on good proposal distributions, which puts constraints on the kinematic chain that can be used. A particle filter method [Hecker et al., 2008], which can robustly handle arbitrary kinematic chains, is used in the computer game "Spore," allowing the player to create their own creatures.

In light of these insights, we propose a sampling-based method, NGIK, which is based on NES. NGIK is both robust and easy to use, and we show in experiments that NES outperforms the other optimization approaches at solving IK for our 41 DOF iCub humanoid robot.

### 3.1.2 Motion Planning

Searching the configuration space of a complex, high DOF robot, such as a humanoid is a computationally expensive procedure. Therefore, a multi-query motion planner, one that stores knowledge about the configuration space, such as PRM [Latombe et al., 1996; Li and Shie, 2007], is far preferable to a single query planner, such as RRT [LaValle, 1998; Perez et al., 2011],which starts a new search each query from scratch.

It has however proven difficult to control how the configuration-space is searched in light of complex constraints. Recent work has acknowledged the lack of control over the search space: Stochastic Trajectory Optimization for Mo-

tion Planning (STOMP) [Kalakrishnan et al., 2011] allows for flexible arbitrary cost-functions and plans a path that minimizes these costs. However, it is only a local trajectory optimizer, which satisfies a single query and does so exclusively in the configuration-space.

Constrained Bi-directional Rapidly Exploring Random Trees (CBiRRT) [Berenson et al., 2009] uses RRT on a constrained manifold; a subset of the configuration-space defined by constraints. It has also been augmented with the concept of task-space regions [Berenson et al., 2011]. However, it is still a single query algorithm and can only use a restricted set of constraints, which can be projected into the configuration space. The latter work does claim that a direct sampling algorithm allows for "arbitrarily complex" constraint parameterization, but only uses it to sample goals and *not* to plan paths as it "can be difficult to generate samples in a desired region".

Clearly it is desirable to have a maximum flexibility in the defining constraints and task-spaces, but current approaches either cannot handle such flexibility, use it only in a part of their algorithm, or find only *one* posture and not a full movement. Recently several frameworks have approached both IK and planning, aiming to be generic and flexible to use [Sentis and Khatib, 2006; Badger et al., 2011; Hauser et al., 2011; Kallmann et al., 2010]. These frameworks can produce intricate motions, but still restrict the kinds of constraints that can be used and many have difficulty with high DOF.

Our framework tackles complex IK and planning at the same time by combining our novel sampling-based inverse kinematics solver NGIK with an iterative roadmap construction strategy. It finds a family of postures that are optimized under constraints defined by arbitrary cost-functions, and at the same time maximally covers a user-defined task-space. Connecting these postures creates a rather dense, traversable graph, called a task-relevant roadmap (TRM). In other words, the task-relevant constraints are built directly into the TRM, and as with the PRM approach, motion planning is reduced to graph search. As shown in Section 4.3, it allows us to build TRMs that can perform useful tasks in the 41-dimensional configuration space of the upper body of the iCub humanoid.

## 3.2 Natural Gradient Inverse Kinematics

In order to find a robot pose with some desired properties, our numerical optimization-based approach requires a cost-function, which we define for convenience as the sum:

$$h \equiv \sum_i h_i(q, B) \tag{3.1}$$

where $q$ and $B$ are the robot configuration and set of workspace obstacles, respectively. Thanks to the robustness of NES, our chosen optimization algorithm, the functions, $h_i$, need not be differentiable or have any other special properties other than being non-negative.

$$h_i(q, B) > 0 \tag{3.2}$$

Given a some arbitrary set of cost functions, $\{h_1, h_2, h_3, \dots\}$, NES generates a sequence of samples, $\{q_1, q_2, \dots, q_\infty\}$, which (hopefully) minimizes $h$. Most of the engineering and computational burden in doing this kind of optimization for IK comes from defining and evaluating cost functions, which encapsulate the results of forward kinematics and collision detection computations. For this reason, NGIK benefits greatly from its use of my kinematic model (section 2), which can be quickly and easily reconfigured, and computes the following:

1. $X(q, B) \equiv \big((x_1, \hat{u}_1), (x_2, \hat{u}_2), (x_3, \hat{u}_3), \dots\big)$ is the set of *marker states*. Each marker is a user-defined, workspace point and unit vector pair, $(x_i, \hat{u}_i)$, which is defined relative to one of the robot's links, $A_i$, in its local reference frame, and therefore moves with that link.

2. $K(q, B) \equiv \big((j_1, k_1), (j_2, k_2), (j_3, k_3), \dots\big)$ is the set of *collision pairs*, where $j_i$ and $k_i$ represent the indices of the $i^{th}$ pair of colliding semi-algebraic models.

Essentially, the kinematic model tells NGIK for a given robot/world configuration, which parts of the robot if any, collide with what, and what are the key geometric features of the pose in the workspace. Thus, some elements of the scalar functions, $h_i$, can be conveniently expressed in terms of $X(q, B)$, and $K(q, B)$:

$$h_i \equiv f_i(q, B) + g_i(X(q, B), K(q, B)) > 0 \tag{3.3}$$

where $f_i$ and $g_i$ are arbitrary functions defined on their respective parameters. Some constraints are much easier to define over $X(q, B)$ and $K(q, B)$ than they are to define over $q$ and $B$, directly, and so the kinematic model increases our creative power significantly. Some interesting cost functions, defined both as $f(q, B)$ and $g(X(q, B), K(q, B))$ are listed in table 3.1.

| Formula | Purpose |
|---|---|
| $h_{home} = \|q - q^*\|$ | Bias search for $q$ toward some particular posture, $q^*$, to focus on a region of configuration space and/or prevent multiple solutions. |
| $h_{collision} = \|K\|$ | Estimate a gradient to avoid collisions ($\|K\|$, the cardinality of the set of collision pairs, usually increases with deeper penetration). |
| $h_{position} = \|x - x^*\|$ | Put a marker, $x$, on some workspace point, $x^*$. |
| $h_{orientation} = \hat{u} \cdot \hat{u}^* + 1$ | Align a marker direction, $\hat{u}$, with some workspace direction, $\hat{u}^*$. |
| $h_{hold} = \|\|x_{left} - x_{right}\| - d\|$ | Hold the left and right hand markers, $x_{left}$ and $x_{right}$, at a fixed distance, $d$. |

Table 3.1. A few simple cost-functions, which NGIK can use to find interesting robot poses.

## 3.3 Task-Relevant Roadmap Construction

To build a TRM, NGIK must be applied iteratively to develop a set of robot poses. NGIK relies on NES, which samples from a gaussian distribution, and so each run must be initialized with a mean and a standard deviation. The optimization runs until the distribution converges on a pose, or a certain time budget is exceeded.

Each iteration of NGIK must return a new solution to the optimization. Furthermore, it is highly desirable that the sample set, once complete, should have some 'nice' statistical properties in terms of regularity (dispersion and discrepancy), such that the poses in the resulting TRM can be smoothly interpolated to create motions. The need to measure a meaningful distance between the samples (given the task at hand) gives rise to the final key idea behind our TRM generation algorithm, the *task space*.

The task space, $Y$, is chosen to parameterize the task at hand, providing the dimensions along which the set of poses should be expanded and interpolated. The robot poses, $q$, are mapped into the task space, yielding points, $y$, which are defined:

$$y \equiv \left\{ \begin{array}{c} y_1 \\ y_2 \\ \vdots \\ y_n \end{array} \right\} \in Y \tag{3.4}$$

Each task function, $y_i$, is some scalar function of $q$ and $B$ (examples in table 3.2), and like the constraints, $h_i$, task functions can be freely defined in terms of other functions of $q$ and $B$, provided by the robot model:

$$y_i \equiv f_i(q,B) + g_i(X(q,B),K(q,B)) > 0 \tag{3.5}$$

where again, $f_i$ and $g_i$ are arbitrary functions defined over their respective parameters.

The goal of the TRM building process is to cover as much of the task space as possible. Accordingly, each NGIK iteration should yield a solution that is displaced from its nearest neighbors by some distance, measured in task space. To repel a robot pose with task space coordinates $y$ from its nearest neighbor $y'$, an additional constraint is required:

$$|(y - y') - d| \tag{3.6}$$

where $d$ controls the density of the set of roadmap vertices in task space.

| Definition | Task Description |
|---|---|
| $Y = \left\{ x_{hand,1} + x_{hand,2} + x_{hand,3} \right\}$ | Move the hand back and forth along some (1D) workspace direction. |
| $Y = \begin{Bmatrix} x_{hand,1} \\ x_{hand,2} \\ x_{hand,3} \end{Bmatrix}$ | Move the hand in all three workspace dimensions. |
| $Y = \begin{Bmatrix} q_a \\ q_b \\ q_c \end{Bmatrix}$ | Move in the null space of the cost function, $w_h \cdot h$, in joints $a$, $b$, and $c$. |
| $Y = \begin{Bmatrix} q_a \\ q_b \\ q_c \\ x_{hand,1} \\ x_{hand,2} \\ x_{hand,3} \end{Bmatrix}$ | Move the hand in workspace and configuration/null space. |

Table 3.2. A few simple task spaces, which facilitate the building of TRMs.

The TRM should be connected in a way that 'makes sense' with respect to the dimensionality, $n$, of the task space. In the work presented here, we connect each new pose to its $n$ nearest neighbors, to form an $n-simplex$. Therefore, to each NGIK iteration are added $n$ constraints, which take the functional form of equation 3.6 and bias the optimization toward a pose that is displaced by a task-space distance, $d$, with respect to its eventual nearest neighbors in the TRM. This is done over and over again, with each new search being initialized at a recently found pose, such that the set of poses grows according to an advancing front. The process is expressed formally in algorithms, 2, 3, and 4.

The map building process produces a graph, $G(V, E)$, consisting of a set of vertices, $V \equiv \{v_1, v_2, v_3, \dots\}$, and a set of edges, $E \equiv \{e_1, e_2, e_3, \dots\}$. With each vertex $v_j$ is associated a robot configuration, $q_j$, a point in task space, $y_j$, and an *expansion weight*, $w_j$, such that each vertex is actually an ordered pair: $v_j \equiv (q_j, y_j, w_j)$. The expansion weight, $w_j$, represents the likelihood that the map can be grown in the neighborhood of $v_i$. It is initialized to 1, and its value decays each time NES fails to return a valid new sample around $q_j$. Each edge represents a robot motion that interpolates two vertices: $e_j \equiv \{v_a, v_b\}$.

**Algorithm 2:** NEAREST_NEIGHBORS - Get the TRM vertices nearest to some task space point.

**Input**: $n$ - integer
$y$ - task space point
$V$ - finite set of TRM nodes, where $v_i = \{q_i, y_i, e_i\} \in V$
**Output**: $N$ - finite set of TRM nodes, $N \subset V$, where $|N| = n$
**begin**
$N \leftarrow n$ elements of $V$, nearest to $y$, according to:
$\|y - (y_i \in v_i)\| \forall v_i \in V$;
**return** $N$;
**end**

## 3.4 TRM Examples

In this section, I present some of the TRMs resultant of my collaboration with Marijn Stollenga. For a more detailed discussion of the experiments, I refer the interested reader to our paper [Stollenga et al., 2013]. The following figures represent Marijn's and my best effort to communicate what kinds of intricate motions we can plan for a complex humanoid, however our award winning video (`http://youtu.be/N6x2e1Zf_yg`)[1], which showcases the roadmaps being applied on real hardware, is infinitely more instructive.

Figures 3.1 and 3.2 show marker poses, (points, $x$, plotted in workspace) for all all robot poses $q_j$ in the TRM. These highlight the shape and character of each TRM, viewed as a whole.

Figures 3.3 through 3.11 show time-lapse snapshots of motions, planned within TRMs. These are intended to give the reader an idea of what the motions look like as well as the potential of our approach for whole-body motion planning.

[1] 'Best Student Video' - Association for the Advancement of Artificial Intelligence (AAAI) Video Competition 2013

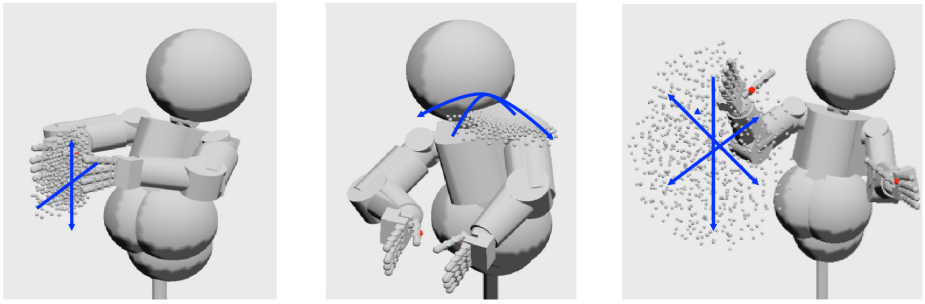

Figure 3.1. Marker positions (dots) for three different task spaces - Relative hand position/orientation is constrained, while the 2D task space is up/down, forward/back for the pair of hands (left). Absolute hand position/orientation is constrained, while the 2D task space is left/right, forward/back for the top of the torso (center). The elbow is constrained to stay down while the 3D task space is up/down, left/right, forward/back for the hand (right).

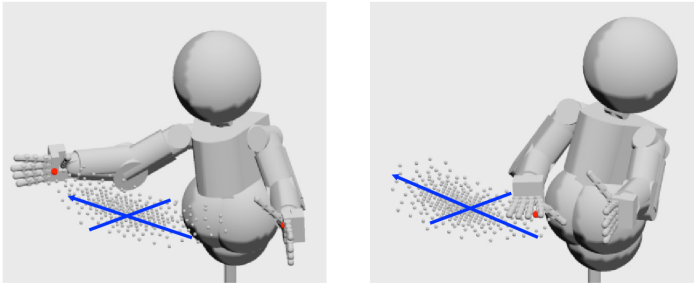

Figure 3.2. Hand marker positions (dots) for the task space: left/right, forward/back for the right hand, while the hand orientation is constrained in two different ways - Palm to the side (left). Palm down (right).

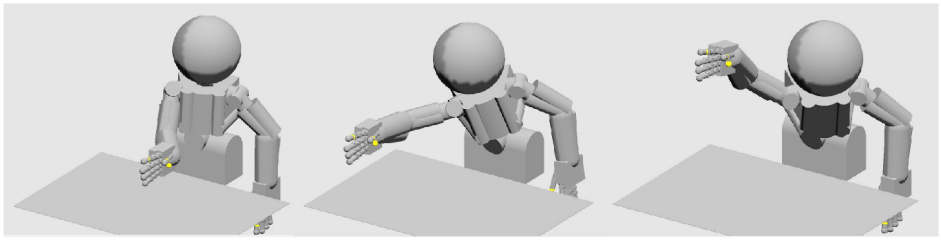

Figure 3.3. Raise a glass - The hand orientation is constrained, while the task space is the 3D workspace. The result is a map for pick-and-place with objects that must not be tipped over.

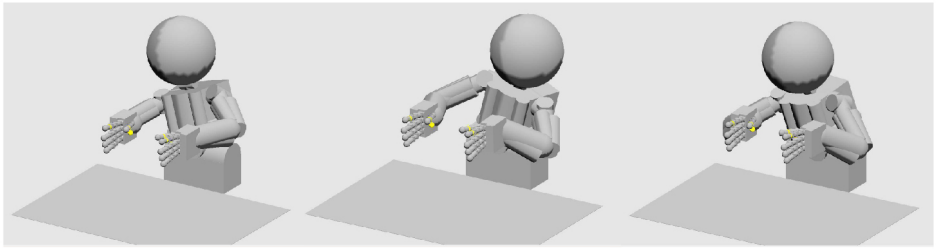

Figure 3.4. Bimanual Inspect - The position/orientation of both hands is constrained, while the task space moves the head around. This map is similar to figure 3.1 (center).

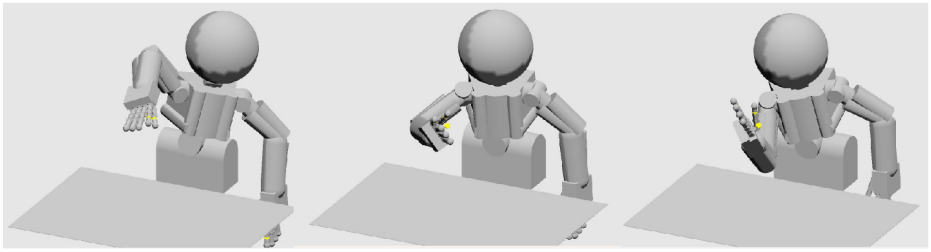

Figure 3.5. Inspect - the 3D position of the hand is constrained, and the task space is its angle with respect to the gaze direction. The resulting map rotates the hand (and any grasped object) in front of the eyes.

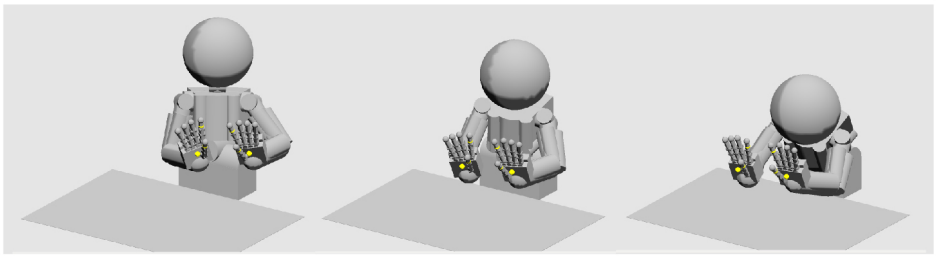

Figure 3.6. Push Forward - The height and relative position/orientation of the hands is constrained, and the 1D task space is to move them forward/back. The resulting map allows the robot to push a large object forward.

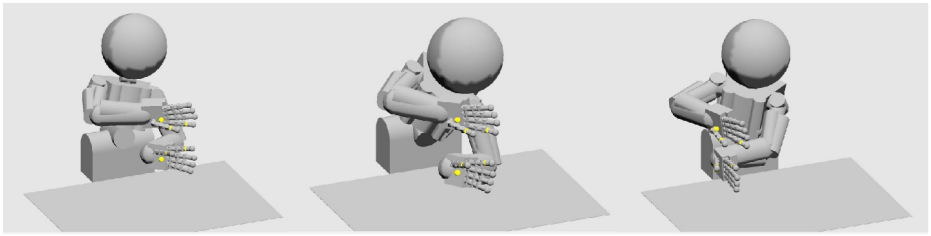

Figure 3.7. Push Over - The height and relative position/orientation of the hands is constrained, and the 1D task space is to move them left/right. The resulting map allows the robot to push a large object to the side.

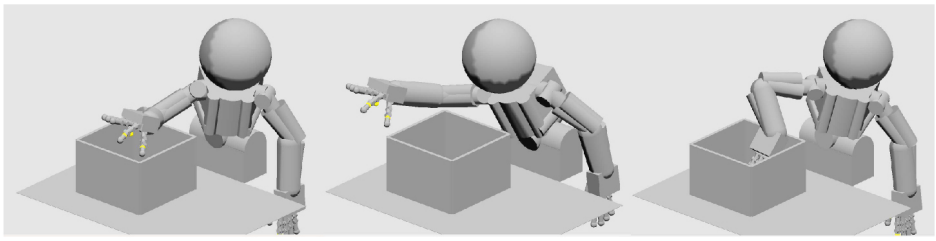

Figure 3.8. Reach Into - The robot is constrained not to collide with the box, and the task space is the 3D workspace for the hand. The resulting map allows the robot to pick/place from/into the box.

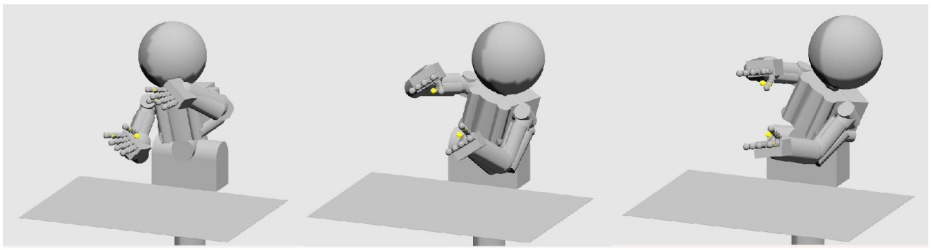

Figure 3.9. Bimanual Rotate - The relative positions and orientations of the hands are constrained, and the 1D task space is the angle of the palm normals (they are parallel with opposite orientation) in the up/down, left/right plane. The resulting map allows the robot to rotate a large object about the forward/back direction.

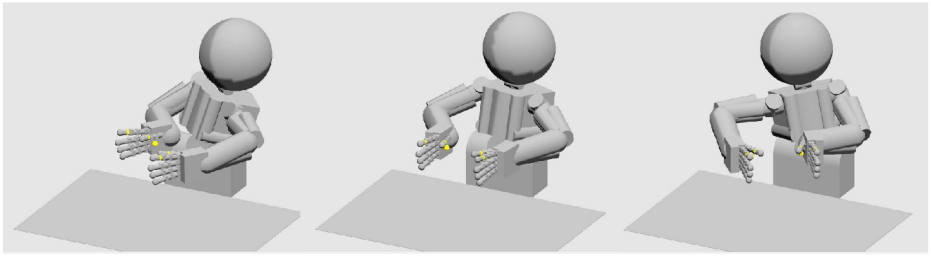

Figure 3.10. Bimanual Twist - The relative positions and orientations of the hands are constrained, and the 1D task space is the angle of the palm normals (they are parallel with opposite orientation) in the table plane. The resulting map allows the robot to rotate a large object about the up/down direction.

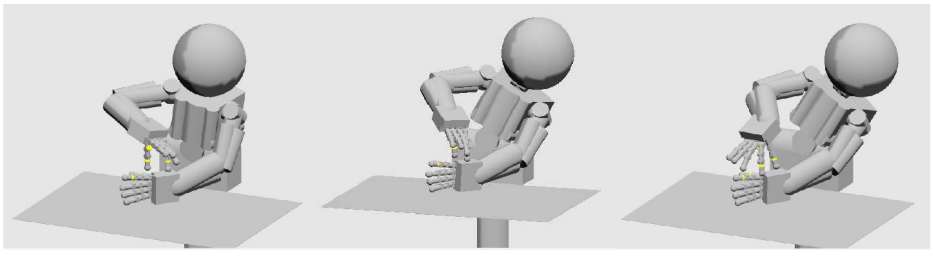

Figure 3.11. Unscrew - The position of both hands is constrained, as well as the orientation of the left hand and the palm normal direction of the right hand. The task space is the 1D orientation of the right hand about its palm normal. The resulting map would be suitable for unscrewing a bottle cap.

**Algorithm 3:** COST - The cost function used by NES for TRM construction with NGIK. NEAREST_NEIGHBORS(*n,y,V*) refers to algorithm 2.

**Input**: $q$ - candidate robot pose
$B$ - set of workspace geometries
$h$ - robot constraints (eq. 8.2)
$Y$ - task space definition (eq. 3.4)
$d$ - scalar task space distance between TRM vertices (eq. 3.6)
$V$ - finite set of TRM vertices, where $v_i = \{q_i, y_i, e_i\} \in V$

**Output**: $cost$, the scalar cost of pose $q$
$y$ - task space projection of $q$
$N$ - nearest TRM vertices to $y$ (algorithm 2)

**begin**

$\{X, K\} \leftarrow$ KINEMATIC_MODEL(*q,B*);

$$y \leftarrow \left\{ \begin{array}{c} f_1(q,B) + g_1(X,K) \\ f_2(q,B) + g_2(X,K) \\ \vdots \\ f_n(q,B) + g_n(X,K) \end{array} \right\} \in Y;$$

$N \leftarrow$ NEAREST_NEIGHBORS(*n,y,V*);

$cost \leftarrow \sum_i h_i$;

**for *each*** $v = \{q', y', e'\} \in N$ **do**

  $cost \leftarrow cost + |(y - y') - d|$;

**end**

**return** $\{y, N, cost\}$;

**end**

**Algorithm 4:** TRM - Build the Task Relevant Roadmap. `COST(`$q^*$`,...)` refers to algorithm 3.

**Input**: $q_{init}$ - pose around which to begin searching
$\delta$ - decrement for expansion success likelihood ($0 < \delta <= 1$)
**Output**: $V$ - vertices of the TRM graph, G(V,E)
$E$ - edges of the TRM graph
$R$ - region $R \subset Y$ covered by G(V,E)

```
begin
    V ← ∅;
    E ← ∅;
    R ← ∅;
    q* ← q_init;
    e* ← 1.0;
    while e* > 0.0 do
        {q, y, N} ← NES(COST(q*,...));
        if IS_VALID(q) and NOT_COLLIDING(q) and y ∉ R then
            v ← {q, y, 1};
            V ← V ∪ {v};
            S_y ← {y};
            for each v_i ∈ N do
                e ← {v, v_i};
                E ← E ∪ {e};
                S_y ← S_y ∪ {y_i};
            end
            R ← R∪ CONVEX_HULL(S_y);
        else
            e* ← e* − δ;
        end
        j ← arg max_i e ∈ v ∀v ∈ V;
        q* ← q_j ∈ v_j;
        e* ← e_j ∈ v_j;
    end
end
```

# Chapter 4

# Real Time Collision Response

TRMs offer a powerful, preemptive motion planning solution that can cope with objects in the robot's workspace, allowing the robot to reach into a stationary box for example (figure 3.8). However like other roadmap approaches, they take time to build, and cannot be updated rapidly enough to cope with a changing environment. Still, the PRM approach, which underlies TRMs, is the most likely antecedent to a developmental learning system for path planning due to its incrementally expandable representation of known motions. The problems it has are all related to the current *separation* between planning and control.

To build up a PRM planner, one must first sample the configuration space to obtain a set of vertices for the graph. The samples are then interpolated by trajectories, which form the set of edges that connect the vertices. The feasibility of each sample (vertex) and trajectory (edge) must be preemptively verified, typically by forward kinematics and collision detection computations, which collectively amount to a computationally expensive pre-processing step. The configuration of the robot *must* remain on the verified network of samples and trajectories at all times, or there may be unwanted collisions. This implies that all the trajectories in the graph must also be controllable, which is in general difficult to verify in simulation for complex robots, such as the iCub, which exhibit nonlinear dynamics (due to do friction and deformation) and are thus very difficult to model faithfully. If these problems can be surmounted, then a PRM planner can be constructed, however the configuration of the robot's workspace must be static, because moving anything therein may affect the feasibility of the graph edges.

All of these problems can be avoided by *embodying the planner* and giving the system the capacity to *react*. If there were a low-level control system, which could enforce all necessary constraints (to keep the robot safe and operational)

in real time, then the planner could simply try things out, without the need to exhaustively and preemptively verify the feasibility of each potential movement.

In this case, reference trajectories would become unnecessary, and the planner could simply store, recall, and issue control commands directly. Lastly, and perhaps most importantly, with the capacity to react in real time, there would be no need to require a static workspace. I have been calling this idea, *adaptive roadmap planning*, and in this chapter, I will describe my initial implementation.

To provide real-time collision response, the kinematic model was embedded within what I will call a 'behavioral framework,' which has evolved as my research developed. Initially, I had envisioned a kind of switching control (figure 4.1), whereby the kinematic model could interrupt a control program, if it threatens to cause an unwanted collision. This would maximize the versatility of the collision response layer, allowing it to be used with any kind of controller, including those already in the iCub repository.

Robot behaviors would thus be decomposed into three abstract tasks that correspond to key objectives in Computer Vision, Motion Planning, and Feedback Control. *Sensory modules* process sensory data and report the state of the workspace, *deliberate planning/control modules* plan (sequences of) actions that are temporally extended and may or may not be feasible, and *reactive control modules* take over if/when the deliberate planner/controller gets the robot into trouble.

In order to realize the interrupt, there must be a 'man in the middle,' which acts as a proxy between the client (control) program and the robot. This proxy should provide its own interface to the robot, which is indistinguishable from the real one, and it should be able to suppress the control commands from the client and inject other (reactive) control commands instead. This was implemented at my behest by Gregor Kaufmann for his master's thesis project at USI [Kaufmann, 2010], resulting in the YARP Port Filter module (figure 4.2), which is written entirely in YARP's API. It allows some arbitrary application (in this case the kinematic model) to proxy YARP's 'Control Board Interface,' an abstraction that represents a group of actuators, such that it can cut the communication between the client program and the robot and/or inject arbitrary data and/or process data as it moves through the filtered ports.

I integrated my kinematic model and Gregor's port filter, such that the state of the model regulates the state of the port filter. When there is no geometric interference (collision), data is allowed to flow freely from the client program (the planner/controller) to the robot, however when a collision occurs, the connection is cut, and control is given to a separate controller (also supplied by the user), which is responsible for reacting to the collision in some way and then

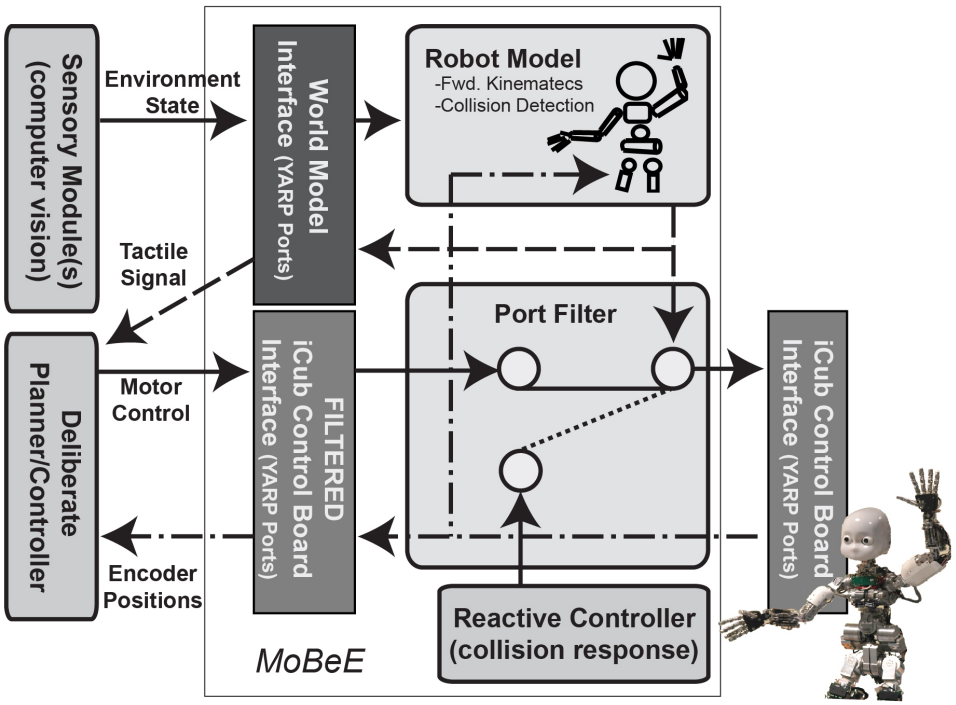


Figure 4.1. Simplified Architecture - Reactive collision response by switching control. An arbitrary control program is connected to the filtered robot interface (left). The port filter connects the real interface to its filtered twin (middle). The kinematic robot model (top), driven by streams of motor encoder positions from the robot, does collision detection and regulates the state of the port filter. When the model detects collision, the port filter cuts off the stochastic controller and invokes an alternative user-defined controller (bottom) to recover from the dangerous configuration.

eventually returning control to the deliberate planner/controller. This mechanism, wrapped in some YARP code to provide communication with client applications (sensory modules and planner/controller modules), became the first implementation of my Modular Behavioral Environment (MoBeE).

MoBeE was never intended to be a foolproof safety mechanism, but rather to facilitate adaptive roadmap planning, as described in the remainder of this chapter. The switching control described here does not provide guarantees that collisions will be prevented regardless of the robot's inertial state. Therefore, obstacles must be modeled with generous bounding volumes, and motor velocities must be selected in accordance with the safety margin afforded by the bounding volumes.

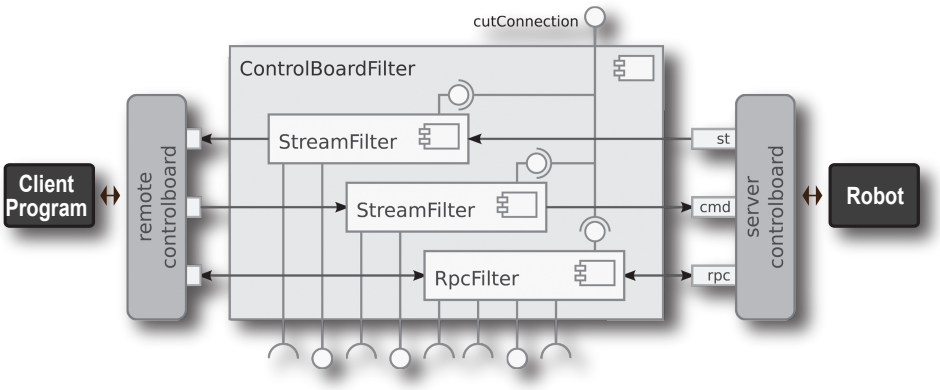


Figure 4.2. The port filter proxies YARP's ControlBoardInterface.

## 4.1 A Simple, Reflexive Collision Response

Initially, my approach was to implement a reactive controller (figure 4.1) as transparently as possible, so that in addition to facilitating adaptive roadmap planning, the collision response could protect any arbitrary control program.

I pursued what was in my view at the time the most elegant solution from a software engineering standpoint. Go back the way you came. This way, the reactive controller would need no information other than the history of robot poses, and would be completely independent of the deliberate planner/controller.

Unlike the sensory module(s) and the deliberate planner/controller, the reactive controller is part of the MoBeE process. Thus, it has direct access to the robot model, the port filter, and the robotic hardware. The reactive controller's privileged position within the MoBeE process allows it to:

1. Respond to state changes in the kinematic robot model.
2. Process the data streaming in and out of the robot in realtime.
3. Suppress input from the deliberate planner/controller.
4. Directly control the robot.

My reflex response is a reactive controller that logs the history of robot poses over time. When it is triggered, poses from the recent history $q_i \in \{q_t, q_{t-1}, q_{t-2}, \dots\}$ are sent to the robot as sequential position move commands, causing the robot to retrace its steps.

This rewinding of the robot pose according to the recent history either continues until the end of the history buffer is reached, or it terminates early when

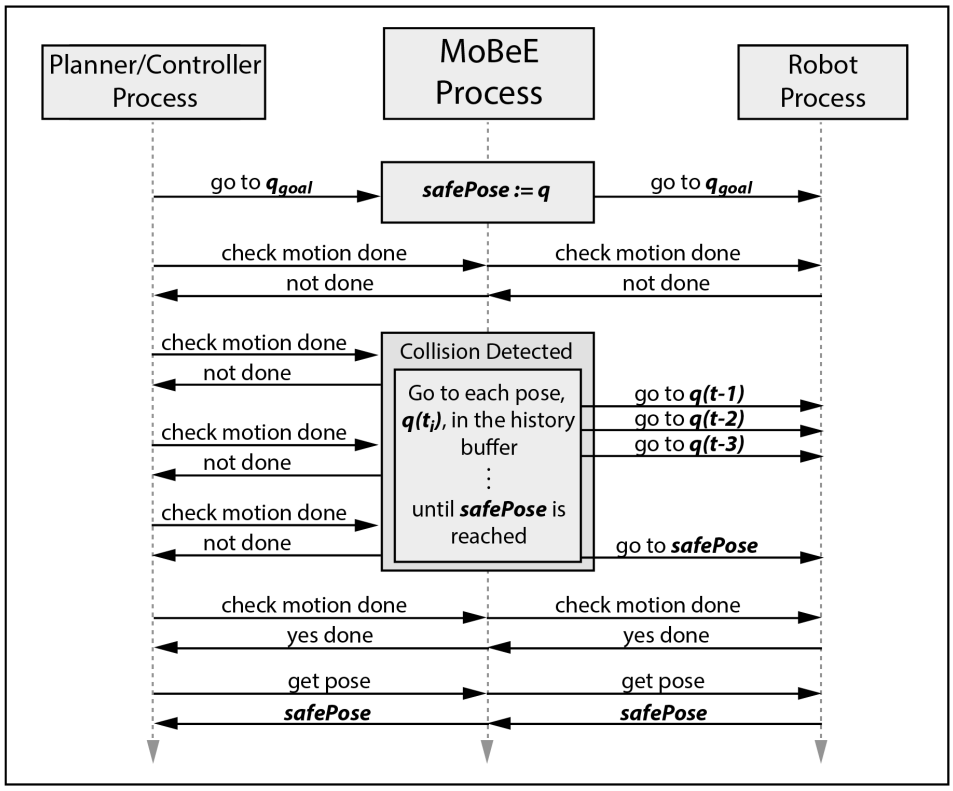


Figure 4.3. Reactive Reflex Controller - In this example, as time goes on (top to bottom), the deliberate planner/controller process (left) communicates (via RPC over YARP) with the hardware (right), through MoBeE (center). MoBeE contains a thread (not pictured), which constantly queries the robot and maintains a circular buffer of the recent history of robot poses. When a new RPC position move command arrives, the current pose in the buffer is marked safe, and the command is forwarded to the robot. As the robot moves, the buffering of poses continues until a collision is detected (or some other constraint is violated) and the reactive controller essentially undoes the half-executed position move command.

a *waypoint* is reached. A waypoint is marked in the history any time the reflex controller sees a position move command go through the port filter over RPC. Thus, from the perspective of the planner/controller, RPC position move commands can succeed or fail, and if they fail, the robot is returned to the position it was in when the RPC position move command was issued, as shown in figure 4.3.

Alternatively, waypoints can be marked in the history from outside MoBeE

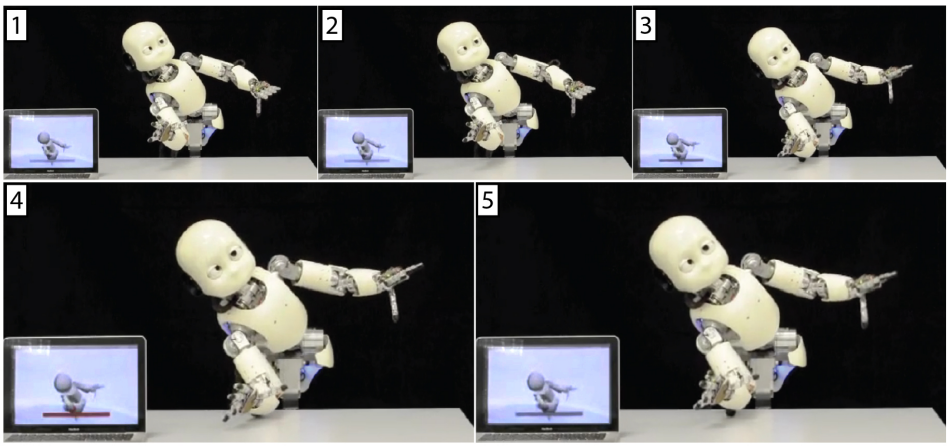


Figure 4.4. Time lapse images of MoBeE detecting impending collision with the table and invoking reflexive response. Collision is detected in frame 4, and response begins in frame 5.

via RPC calls. This way, deliberate planner/controller modules can use arbitrary control modalities and still communicate to MoBeE where the last known, safe pose is located.

### 4.1.1 Experiment: Motor Babbling

The software architecture employed in this experiment is shown in figure 4.1. The deliberate planner/controller is stochastic and sends randomly generated position move commands over RPC. All joints on the iCub upper-body except those in the hands are controlled, for a total of 23 DOF. When invoked, the reflexive collision response, tracks the inverted, recent history of robot poses, returning it to the configuration it was in prior to the issue of the currently active RPC command. The modeled environment consists of a table, as pictured in figures 2.2 and 4.4. The experiment ran for approximately two hours over several trials of 5 to 20 minutes each with joint velocity limited to 20% of maximum. Video excerpts of some of these trials are available on the IDSIA robotics web page (http://robotics.idsia.ch/).

As expected, the stochastic controller quickly produced many motions, which uninterrupted would have resulted in destructive collisions. However MoBeE effectively prevented all of them. Included were several commonly occurring self collisions, such as elbow vs. hip, and upper-arm vs. chest, as well as many collisions between the hands/forearms and the table.

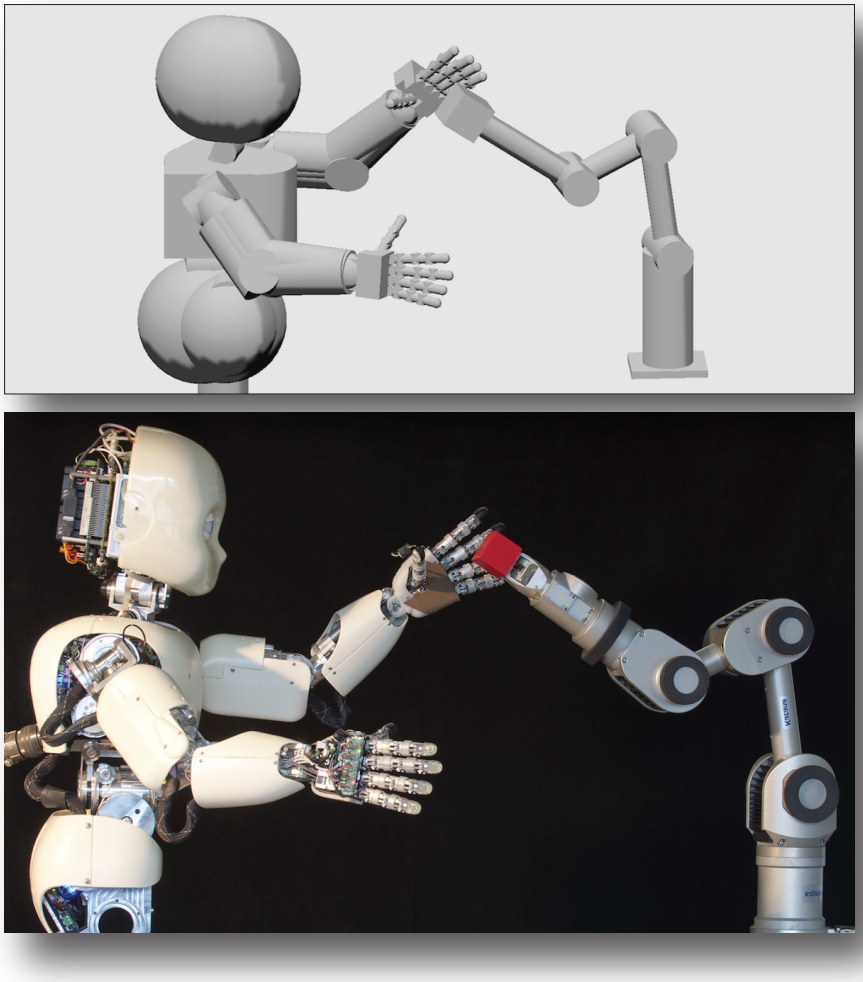

Figure 4.5. The iCub and Katana robots (bottom) cooperate in a shared workspace. Each robot is controlled via its own deliberate/reactive controller pair and the shared MoBeE framework (top).

The experiment demonstrates that the MoBeE framework can effectively simulate tactile feedback for a complex humanoid robot in realtime.

### 4.1.2 Experiment: Robots Teaching Robots

In this experiment, the MoBeE framework is exploited to develop a sensory module for computer vision, using a machine learning approach.

The Katana arm is used to place an object of interest, in this case a chil-

dren's block, precisely at a number of known 3-Space locations *within the iCub's workspace* (figure 4.5). Meanwhile, the iCub moves about the object according to a stochastic control policy. Seeing the object from different angles and distances, the iCub constructs a data set, from which it learns to map camera images to 3-Space locations, given body states.

The modular architecture of the MoBeE framework drastically facilitates the implementation of the rather complex experimental setup required to do this kind of multi-robot interaction. The kinematics of the iCub and the Katana are loaded from XML into a common model. The reactive controller described above, which implements reflexive collision response, is used for both robots. In order to produce the desired training data however, the Katana and the iCub require different planner/controller modules.

The Katana's planner/controller is very simple. It just moves through a series of predetermined poses, waiting at each one, such that the iCub can observe the block. The iCub's, on the other hand, is stochastic. For each move of the Katana, the iCub assumes a number of randomly selected poses, from which it observes the block. Occasionally, the two robot models do collide, and the reflexive collision response prevents physical collision, safely returning the hardware to a previous configuration.

In order to accomplish this reliably, the two reflex behaviors must be synchronized[1], by adjusting the control frequency. That is, the speed with which the reflex controller indexes through the history buffer and issues commands to the robot. With this done correctly for each robot (and with respect to one another), the reflexive responses of both are synchronous, and the stochastic collection of training data runs robustly for hours.

This experiment supports the following key claims: MoBeE is robot independent, and can exploit any device that can be controlled via YARP. It also supports multiple interacting robots, and behavioral components are portable and reusable thanks to their weak coupling.

So far, the reflexive controller has been demonstrated operating with three different deliberate modules on two different robots, the stochastic motor babbling planner/controller on the iCub, the stochastic image gathering planner/-controller on the iCub, and the deterministic block-indexing planner/controller, which ran on the Katana.

Since this first generation of MoBeE was completely transparent, it imposed *no* constraints on the deliberate planner, and in fact the different planner/con-

[1]In fact this synchronization is already an issue when considering only the iCub, since its body parts are controlled individually. However the synchronization between the iCub and the Katana requires more tuning, since their hardware differs significantly.

troller modules mentioned were implemented by different developers, some of whom had little or no knowledge of the reactive controller.

## 4.2 Adaptive Roadmap Planning

The motion planning literature provides many algorithms that perform well in static environments. On the other hand, the control literature provides methods for quickly reacting to avoid collisions in dynamic environments. However, manipulating objects means changing the environment drastically yet sporadically. This is a problem, which has received surprisingly little attention, but the simple reflexive collision response described in the previous section can help to address it.

Consider a roadmap planner that computes trajectories as shortest paths through a graph, $G(V, E)$[2], which covers the configuration space of a humanoid upper-body coarsely and respects (avoids) self-collisions. Such a graph allows the robot to move around safely in an empty workspace, but if one puts a table in front of the robot, some motions (graph edges) are likely made infeasible, changing the topology of $G$. The planner is broken, because it is no longer clear which edges are feasible and which are not. In order to restore the planner, the feasibility of the entire graph must be recomputed, which is an extremely expensive operation.

A motion planner fails for want of *feedback*. If a roadmap planner could somehow sense the failure to traverse an edge in realtime, and return the robot to a valid vertex, then it could perhaps recover from planner failure. The edge could be removed from the graph, and a new trajectory could be planned in the updated graph without the need to recompute the feasibility of every other edge. Rather than throwing the roadmap graph away and starting from scratch, it could be adapted to new constraints as they are discovered.

Consider that before the discovery of an infeasible edge, the graph, $G$, had been a valid, albeit trivial, Markov Decision Process (MDP), with states $V$ and actions $E$, and all state transition probabilities equal to one. The discovery of the infeasible edge invalidated the MDP. The probability of transitioning from $v_a$ to $v_b$ along $e_{ab}$ had been equal to one, now it is equal to zero. The probability distribution governing the state transition associated with action $e_{ab}$ no longer sums to one. In order to repair the MDP, planner failure must be cast into some

[2] Our graph implementation relies on Boost [The Boost Graph Library] and the Computational Geometry Algorithms Library (CGAL) [The CGAL Project]

kind of valid state transition. In other words, if the planner tries to go from $v_a$ to $v_b$ along $e_{ab}$, and does not end up in $v_b$, then it must end up in some other state.

Fortunately, this is exactly what the reflexive collision response does (figure 4.3). It returns the robot to the state from which it came. With the MoBeE framework and reflexive collision response, an edge, $e_{ab}$, now has two possible state transitions associated with it: $v_a \rightarrow v_b$ and $v_a \rightarrow v_a$. Thus, the state transition probabilities can be maintained such that they always sum to one. With the validity of the MDP restored, the discovery and avoidance of new constraints/obstacles can be treated as a reinforcement learning (RL) problem!

This simple behavior, when coupled to the roadmap planner, has the following important consequences:

1. Roadmap planning is generalized to non-static environments by adopting probabilistic state transitions and casting the roadmap graph into an MDP.
2. Within the MDP, the discovery and avoidance of novel objects/obstacles/-constraints can be phrased as an RL problem.
3. In contrast to other approaches that generalize roadmap planning to non-static environments [Brock and Khatib, 2000], the topology of the roadmap can be changed.

## 4.3 Demonstrative Experiments

Following are the results of three demonstrative experiments, which were carried out to evaluate the feasibility and usefulness of the MoBeE behavioral framework. First, I present two simple demonstrations of Adaptive Roadmap Planning without vision. Then I evaluate a real world application of adaptive roadmap planning on a large TRM with computer vision.

### 4.3.1 Roadmap from Scratch

In the first experiment, a roadmap is constructed optimistically, from scratch, and explored by the actual robotic hardware. 20 random samples are chosen in the configuration space of the iCub humanoid robot, and they are connected to their 10 nearest neighbors (figure 4.6, left), without verifying the feasibility of the resulting graph edges.

Importantly, the edges do not even represent trajectories explicitly. Instead, they represent the whatever motion ensues when the robot is at one configuration and a position move command is sent to move it to another configuration.

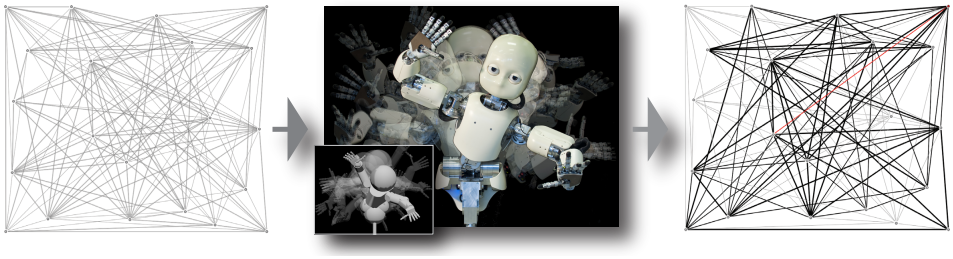

Figure 4.6. A Roadmap graph is built autonomously, online, by the iCub humanoid robot. Samples are connected optimistically to their $k$ nearest neighbors, yielding a Roadmap graph $G(V, E)$ (left). The iCub explores the graph (center), and collision detection is done by MoBeE (center inset). Infeasible edges are removed from the graph, which is thus adapted to the physical constraints of the iCub. The feasible portion of the graph is shown in bold (right). The remaining non-bold edges are unreachable, and the red edge represents the currently active motion.

Trajectory tracking is not necessary, because the graph edges represent control commands directly. This greatly simplifies and tightens the behavioral control loop as compared with the traditional plan first, act later paradigm.

The iCub explores the roadmap graph by randomly planning and executing motions (figure 4.6, center). The target pose selection is biased toward those roadmap vertices with unexplored, adjacent edges. Running the iCub at a conservative 10% of maximum velocity, the exploration process requires approximately 90 minutes to completely determine the feasible sub-graph. I have carried out similar experiments with a number of different graphs, and observe that the rollback of position move commands works robustly in practice, and roadmaps can be robustly constructed on-line, from scratch while avoiding self collisions.

Although the MoBeE infrastructure facilitates optimistic construction of the roadmap graph, I am compelled to point out the following: Small, randomly generated graphs often contain unreachable vertices and edges (figure 4.6, right). These can usually be connected to the graph by construction, if the map is grown incrementally, however a pruning step would improve the neatness of the graphs in general.

Secondly, it is possible that a vertex has feasible 'in' edges, but no feasible 'out' edges. Moving to these vertices causes the exploratory behavior to get stuck. To facilitate motions away from such partially-connected vertices, new edges (and possibly vertices) must be constructed. Ultimately, to maximize

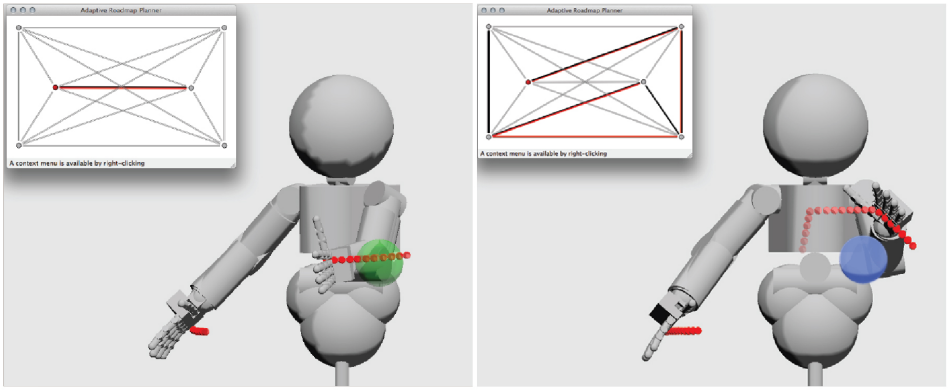

Figure 4.7. The iCub autonomously re-plans a motion to move from one side of the ball to the other. If the ball is not a solid object (top), the Agent moves the hand through it. When the ball is suddenly made an obstacle (bottom), the Agent quickly finds the path around it. The active plan is shown with red edges in the inset graphs.

the planner/controller's constructive power, it should be equipped with a single query planner that can robustly find paths back to the graph from partially connected vertices.

### 4.3.2 Adaptive Re-Planning

The second experiment is based on a very small graph, which I constructed deliberately, such that there exist two different paths that move the hand from one side of the ball to the other. The shorter path causes the hand to pass through the ball, whereas the longer path circumvents it. Initially, the model of the ball is left out of collision detection computations (figure 4.7, top, green ball), and the planner/controller prefers to move the hand to the other side of the ball via the shortest available path, through the ball. When the ball is made solid[3] (figure 4.7, bottom, blue ball), the planner/controller immediately finds the alternative path around it. This demonstrates that with the supervision of MoBeE, the planner/controller can alter the topology of the roadmap in realtime to adapt to a changing environment.

[3]MoBeE supports on-the-fly editing of objects, including collision checking behavior.

### 4.3.3 A Real-World Reaching Task

This final experiment integrates the adaptive roadmap planner/controller, the reactive reflex controller, and the sensory module learned in section 4.1.2, to produce reaches to real-world objects, using the iCub.

The sensory module identifies and locates the objects of interest at regular intervals and sends RPC commands to update the world model in MoBeE. Meanwhile, the planner/controller queries MoBeE (again via RPC) for the state of the salient object, plans a reach, and tries to execute it. Of course the reflexive controller may intervene.

A task of this scale, requires that we use a much larger roadmap than we have shown in the previous experiments. Consider for a moment what such a map should look like. Most of the robot configurations associated with the vertices of the roadmap graph should put the iCub's hand at feasible pre-grasp postures. If we intend to cover the approximately $\frac{1}{2}m^2$ of reachable table with pre-grap poses at, say, $1cm$ resolution, we require $5,000$ vertices in the map. It is impractical to construct such a map by hand. Random sampling is also infeasible, and we must therefore search for our graph vertices more intelligently.

To find the vertices of the large roadmap, we employ the TRM framework introduced in chapter 3, and the result is shown in figure 4.8. I would like to reiterate that collision detection computations are unnecessary to verify the feasibility of the edges. Instead the map is connected optimistically using $k$ nearest neighbor search. In this case $k = 8$.

This optimism makes a lot of sense in light of the application. Since the map consists of pre-grasp poses with the hand above the table, there are very few infeasible edges. Although it would clearly take a very long time to explore the entire map, controlling the hardware through every edge, there is actually no reason to do so. Instead, the map is simply exploited greedily, generating reaches as necessary, and whenever infeasible edges are found, for example when the back of the hand bumps into the object it is trying to pre-grasp, the planner/controller can adapt the map to the newly discovered constraint and re-plan.

The canonical roadmap planner would sample every edge in the graph and do extensive collision detection computations to verify the feasibility of each motion whenever the world state changes. Consider briefly how much time that would take.

There are $5,000$ vertices at roughly 1cm resolution in operational space, with 8 edges per vertex. If the edges were sampled at 1mm resolution, there would be $400,000$ poses for which collision detection must be computed. The kinematic model within MoBeE, when run offline, can compute collision detection for iCub

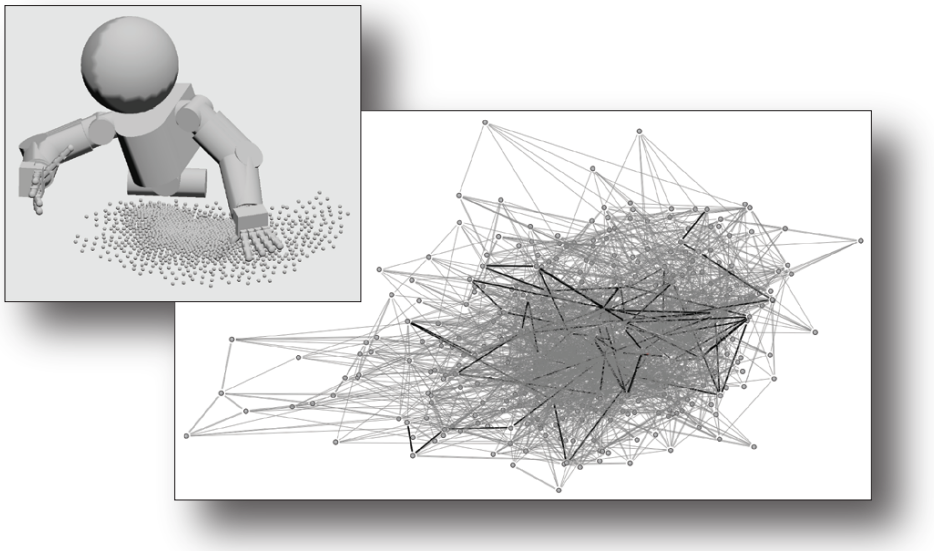

Figure 4.8. A large TRM is constructed by searching the configuration space for a set of approximately 5,000 'interesting' poses. The scattered dots in the robot model (top-left) represent the position of the left hand as the robot assumes the pose associated with each vertex in the map (bottom-right).

poses at about 1,000Hz, if the workspace is devoid of obstacles. Therefore we are talking about roughly 7 minutes of offline computation to validate the map every time the state of the workspace changes.

This experiment demonstrates that MoBeE and its implied behavioral decomposition, allow a roadmap data structure for motion planning to be built and maintained in a fundamentally different way than the state-of-the-art. In running this and other similar experiments, I observe that proposed adaptive roadmap planning works well in practice, generating reaches to objects as pictured in figure 4.9. Moreover, owing to the modularity of the MoBeE, behaviors can easily be modified with minimal development overhead.

## 4.4 Discussion

Manipulation behaviors require planning and re-planning in an environment in which things are moved around. In this chapter, I proposed a simple yet novel integration of roadmap planning with reflexive collision response, which transformed the roadmap graph representation into a simple MDP. Roadmap

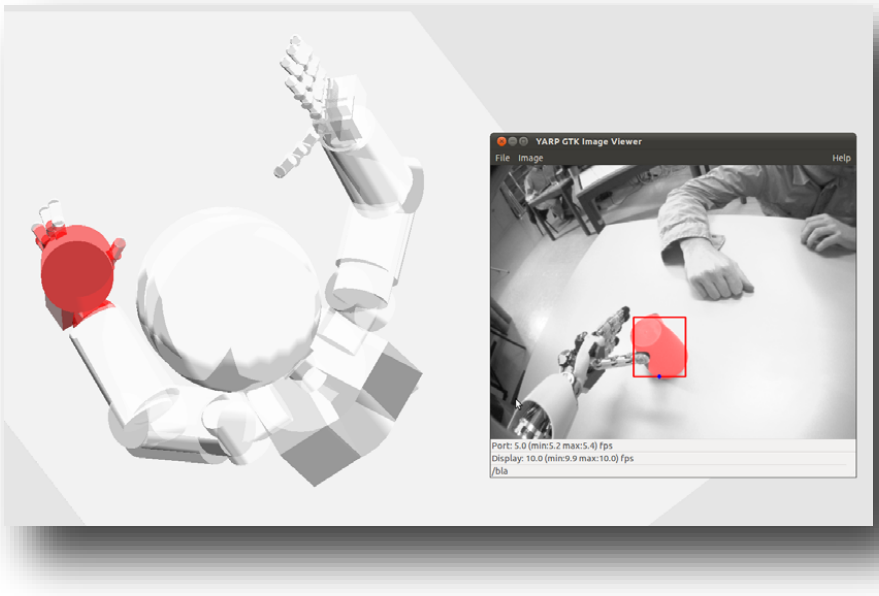


Figure 4.9. The resulting pose after reaching to the cup using an integrated Sensor, Agent, Controller system with the iCub robot. The inset (right) shows the iCub's vision, overlaid with the (red) output of an Detector module. The cup, modeled as a cylinder, has been placed into the MoBeE model (left) by a Locator module. The roadmap used to plan the reach is pictured in figure 4.8.

planning was thereby extended to changing environments, and the adaptation of the map could be phrased as an RL problem.

A consequence of the proposed adaptive roadmap planning is that it becomes uncertain whether or not particular motions are feasible. Therefore, the approach that the robot is able to respond to collisions in real time. This motivates the notion of a 'behavioral framework,' which tightly integrates key elements from computer vision, motion planning, and feedback control. I introduced MoBeE, my behavioral framework implementation, and I presented the results of some preliminary experiments on adaptive roadmap planning.

The next logical step would be to design an RL agent to do the roadmap exploration well, and/or to remember what the roadmap is like under different workspace constraints. It was in considering this problem that I began to rethink certain design decisions that had gone into MoBeE.

The first issue that became apparent is that any roadmap, which is dense enough to facilitate real world reaching quickly becomes far too large to be practical. The map in figure 4.8, for example, has many tens of thousands of edges. Finding a path around a novel obstacle takes a very long time, because

any obstacle invalidates many, many edges in such a dense graph.

Secondly, the assumption that any planner/controller must be supported is also limiting. With no knowledge of what the planner/controller does, MoBeE has little choice but to replace its input entirely when things go badly. Though elegant on paper, the switching control led to a number of issues all related to deciding when to switch and maintaining a history of collision free poses. Noise in the motor encoder position signal meant that rather than seeing a nice clean switch from collision free to colliding, one sees a high frequency oscillation over several ms, which finally ends in a stream of only colliding poses. One can work around this problem by introducing some filtering, and/or a wait period, however it is difficult to guarantee the robustness of such solutions.

Even more challenging were hysteretic problems. These were all related to the fact that the forward motions and their approximate inverses generated by the reflex response can have different shapes. This is particularly problematic when trying to rewind a motion, which approached a collision from a very low angle, such as moving the hand across the surface of a table. A small deviation between the forward motion and its inverse can drastically affect whether or not collisions are detected, and this can cause the collision recovery to fail.

# Chapter 5

# MoBeE 2.0

The first MoBeE implementation was intended to enforce constraints in real time while a YARP robot is under the control of *any arbitrary planner/controller*. This led to a design based on switching control, which facilitated experimentation with control modules pre-existing in the iCub repository, as well as collaboration with developers who had little or no knowledge of the inner workings of MoBeE.

Ultimately, the first MoBeE implementation does a pretty good job at protecting a robot from a stochastic/exploratory planner/controller, and it provides a sensible, albeit very simple, reflexive response to the violation of constraints, which is to interrupt the planner/controller and return the robot to a previous configuration, further away from the constraint boundary.

However, problems related to noise and hysteresis made it difficult to maintain a history buffer containing a collision free inverse of the recent robot motion. Therefore, the robustness of the switching control is questionable, and under certain challenging circumstances the robot can get stuck in a situation where it does not have any safe poses in the buffer.

Collaboration notwithstanding, my primary purpose for MoBeE has always been to facilitate adaptive roadmap planning. With this in mind, I began to think about how I could improve MoBeE's collision response if I were to relax the requirement for compatibility with arbitrary planner/controller modules. Upon reflection, I had the following insight.

The key feature of MoBeE 1.0, regarding adaptive roadmap planning, is that its realtime collision response allows the edges of the roadmap graph to represent control commands directly, rather than trajectories. This is what had, albeit

indirectly, allowed me to cast the roadmap planning framework[1] into an MDP[2].

In light of the MDP-Roadmap planning framework, there was no longer any reason to confine the robot to a finite set of trajectories. Moreover, *stop and go back* is an ugly collision response solution, which is wasteful of time and looks counterintuitive to observers of a demonstration.

I began to believe that I could do away with the switching control and its problems with noise and hysteresis as well as the reflexive controller described in section 4.1. They would be replaced by a more robust collision avoidance strategy, which would not only make the robot behavior more appealing to an observer, but also enrich the MDP far beyond the trivial version proposed in section 4.2.

## 5.1 MoBeE 2.0 Approach

Collision avoidance is essentially an inverse kinematics problem. The constraints to be avoided are defined in the workspace, but must be avoided by control commands, which are embedded in configuration space. Due to the complex, nonlinear, not-necessarily-invertible mapping from configuration space to workspace, it is not always straightforward to control the robot so as to affect the desired workspace changes in its pose.

One solution to this kind of inverse kinematics problem is to search configuration space for a solution, however where avoiding collisions and other constraints is concerned, time is often of the essence. MoBeE 1.0 avoided the inverse kinematics problem entirely by exploiting the recent history of robot motion. The second (and current) implementation, MoBeE 2.0, is based on the reactive collision avoidance approach of Ousamma Khatib, and its descendants, which were introduced at some length in section 1.6.2.

Crucial to the application of the approach to manipulators is the well known Jacobian matrix:

[1]Roadmap planning is based on a graph, $G(V,E)$, wherein the vertices $v \in V$ represent robot poses and the edges $e \in E$ represent verified collision free trajectories that interpolate the vertices.

[2]MDPs are based on a model-tuple, $\langle S,A,T \rangle$, which comprises abstract sets of *states*, $S$, and *actions*, $A$, and a matrix of state transition probabilities, $T$, the elements of which represent the probability of transitioning from one state to another, given the execution of some particular action.

$$J(\mathbf{q}) = \frac{\partial \mathbf{x}(\mathbf{q})}{\partial \mathbf{q}} = \begin{bmatrix} \frac{\partial x_1(\mathbf{q})}{\partial q_1} & \cdots & \frac{\partial x_1(\mathbf{q})}{\partial q_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial x_6(\mathbf{q})}{\partial q_1} & \cdots & \frac{\partial x_6(\mathbf{q})}{\partial q_n} \end{bmatrix} \tag{5.1}$$

where $\mathbf{x}$ represents the 6DOF position/orientation one of the robot's links[3], and $\mathbf{q}$ is an $n$ dimensional vector of joint positions.

The Jacobian can be evaluated at any robot configuration, $\mathbf{q}^*$, in order to map *small* configuration space displacements, $\Delta\mathbf{q}$, to the workspace, according to:

$$\Delta\mathbf{x} = J(\mathbf{q}^*)\Delta\mathbf{q} \tag{5.2}$$

Because the Jacobian is evaluated at $\mathbf{q}^*$, it is only valid in the neighborhood of that point, and so the magnitude of the configuration space displacement, $|\Delta\mathbf{q}|$, must be small in order that the magnitude of the workspace displacement, $|\Delta\mathbf{x}|$, is meaningful. It follows that the Jacobian inverse can be used to map desired workspace displacements to configuration space according to:

$$\Delta\mathbf{q} = J(\mathbf{q}^*)^{-1}\Delta\mathbf{x} \tag{5.3}$$

However here, $|\Delta\mathbf{x}|$ must be small in order that $|\Delta\mathbf{q}|$ is meaningful.

Applying equation 5.3 can be problematic due to the matrix inversion. $J(\mathbf{q}^*)$ may not be square, which is obviously the case for a 7DOF anthropomorphic arm, and even if it is square, it may be singular at certain $\mathbf{q}^*$. Non-square Jacobians can be pseudo-inverted [Whitney, 1969], however this does not help when the matrix is singular.

More robust approaches employ different kinds of nonlinear optimization including damped least squares [Wampler, 1986; Nakamura and Hanafusa, 1986; Buss and Kim, 2005] and quasi-Newton methods [Wang and Chen, 1991; Zhao and Badler, 1994; Deo and Walker, 1993]. However these are computationally expensive, and their application should therefore be well justified.

Contrastingly, a seemingly abusive use of the transpose instead of the inverse costs almost nothing to compute and works surprisingly well under many circumstances:

$$\Delta\mathbf{q} = J(\mathbf{q}^*)^T\Delta\mathbf{x}_\mathbf{d} \tag{5.4}$$

[3]The kinematic model provides easy access to Jacobian matrices for particular, named links of interest, which are defined using *<marker/>* tags (section 2.3.1).

Though not the same as the inverse, it can be shown that the Jacobian transpose always maps workspace displacements to configuration space reasonably well [Balestrino et al., 1984; Wolovich and Elliott, 1984].

Let $\Delta\mathbf{x_d}$ represent the desired workspace displacement and $\Delta\mathbf{x_a}$ represent the actual displacement, which results from computing $\Delta\mathbf{q}$ and moving the robot. Substituting equation 5.4 into equation 5.2, one obtains:

$$\Delta\mathbf{x_a} = J(\mathbf{q}^*)J(\mathbf{q}^*)^T\Delta\mathbf{x_d} \tag{5.5}$$

Now, dropping the $\mathbf{q}^*$ from the Jacobians for simplicity, consider the inner product:

$$\Delta\mathbf{x_d}\cdot\Delta\mathbf{x_a} = JJ^T\Delta\mathbf{x_d}\cdot\Delta\mathbf{x_d} = J^T\Delta\mathbf{x_d}\cdot J^T\Delta\mathbf{x_d} = \|J^T\Delta\mathbf{x_d}\|^2 \geq 0 \tag{5.6}$$

Therefore, in the worst case, the displacement, $\Delta\mathbf{x_a}$, will be orthogonal to the desired displacement, $\Delta\mathbf{x_d}$. In most cases however, a component of $\Delta\mathbf{x_a}$ will be in the direction of $\Delta\mathbf{x_d}$.

Moreover in practice, if equation 5.5 is applied cyclicly to reduce some error distance to a target point in workspace, (algorithm 5) it will, in fact, always do that, provided that the target is reachable and the steps $\Delta\mathbf{q}$ are small enough. However, due to the fact that $\Delta\mathbf{x_a}$ can be at worst perpendicular to $\Delta\mathbf{x_d}$, the path to the goal pose can be somewhat circuitous. Moreover, if the goal pose is not reachable, the robot will typically approach a singular configuration, and the Jacobian transpose control will cause the robot to oscillate.

If reaching a goal pose in workspace is the task at hand, then a proper pseudo-inverse can outperform the transpose, and least squares optimization and semi-Newton methods certainly do. However, consider that in the context of MoBeE, the task is to compute configuration space forces to avoid collisions. Therefore, the desired direction, $\Delta\mathbf{x_d}$, is *away from collision*. Any direction is acceptable, as long as it does not move the robot closer to collision. Moreover, there can potentially be many collisions computed by the robot model, and these may involve many different links, each of which requires its own projection, *and* since this is collision avoidance, responsiveness is paramount. Therefore, the simplicity and thus speed of the Jacobian transpose projection caused me to select it for the MoBeE 2.0 implementation.

**Algorithm 5:** $J^T$ Control - Operational space control with the Jacobian transpose.

**Input**: $\mathbf{x}^*$ - goal pose in workspace
$\mathbf{x}$ - current workspace pose
$\mathbf{q}$ - current robot configuration
$f(\mathbf{q})$ - forward kinematics function
$J(\mathbf{q})$ - the Jacobian matrix evaluated at the point $\mathbf{q}$
$\alpha$ - a scalar step size
**Output**: $Q$ - a piecewise linear trajectory comprising segments, $\Delta\mathbf{q}$
**begin**
  $Q \leftarrow \emptyset$;
  **while** $\mathbf{x} \nsim \mathbf{x}^*$ **do**
    $\mathbf{x} = f(\mathbf{q})$;
    $\mathbf{x_d} \leftarrow \mathbf{x}^* - \mathbf{x}$;
    $\Delta\mathbf{q} = \alpha J(\mathbf{q})^T \Delta\mathbf{x_d}$;
    $Q \leftarrow Q \cup \{\Delta\mathbf{q}\}$;
    $\mathbf{q} \leftarrow \mathbf{q} + \Delta\mathbf{q}$;
    **if** $\mathbf{x} = f(\mathbf{q})$ **then**
      break;
    **end**
  **end**
  **return** $Q$;
**end**

## 5.2 MoBeE 2.0 Implementation

MoBeE 2.0 controls the robot constantly, at a user-defined frequency according to a user-defined control modality. Position, velocity, and force control are supported. The control signal is computed according to the response of the following second order dynamical system:

$$\mathbf{M}\ddot{\mathbf{q}}(t) + \mathbf{C}\dot{\mathbf{q}}(t) + \mathbf{L}\left(\mathbf{K}\left(\mathbf{q}(t) - \mathbf{q_0}\right)\right) = \begin{cases} \mathbf{f_{config}} \\ +\mathbf{f_{work}} \\ +\mathbf{f_{lim}}(\mathbf{q}) \\ +\mathbf{f_{cst}}(\mathbf{q}) \\ +\mathbf{f_{coll}}(\mathbf{q}) \end{cases} \tag{5.7}$$

The vector function $\mathbf{q}(t) \in \mathbb{R}^n$ is the robot configuration over time, $\dot{\mathbf{q}}(t)$ and $\ddot{\mathbf{q}}(t)$ are its first and second time derivatives. The matrices $\mathbf{M}$, $\mathbf{C}$, and $\mathbf{K}$ contain

mass, damping, and spring constants respectively. The position vector $\mathbf{q_0}$ is an attractor configuration, which can be set by the client program via RPC.

The system is forced by the functions on the right hand side. The client program can contribute to the forcing (again over RPC) by providing a force in configuration space directly, which results in $\mathbf{f_{config}}$, or by applying workspace forces to markers defined in the robot's XML file (section 2.3.1), which MoBeE internally projects into configuration space as $\mathbf{f_{work}}$. The remaining forces constrain the system. Joint limit avoidance is implemented by $\mathbf{f_{lim}}(\mathbf{q})$, linear systems of constraints (also specified in the robot model's XML file) result in $\mathbf{f_{cst}}(\mathbf{q})$, and workspace collisions are projected into configuration space and summed to yield, $\mathbf{f_{coll}}(\mathbf{q})$. Finally, $\mathbf{L}$ is a sigmoidal Lyapunov function (equation 5.8), which squashes the spring force, such that no dimension exceeds a certain maximum. It is also used to control the maxima of some of the forcing terms, which are defined and and discussed in the following subsections. The architecture of the system is shown schematically in figure 5.1.

### 5.2.1 A Sigmoidal Lyapunov Function for Thresholding

To ensure that the MoBeE 2.0 system prevents collisions under ordinary operating conditions (with limited energy in the dynamical system[4]), it is essential that the different forces, $\mathbf{f_i}(t)$, mix well, and that their magnitudes also make sense with respect to the spring force, $\mathbf{L}\left(\mathbf{K}\left(\mathbf{q}(t)-\mathbf{q_0}\right)\right)$. In other words, the right forces must dominate the system at the right times, and their maxima must therefore be controlled.

To control the contributions of forces, which may otherwise go to infinity, I have designed a sigmoidal Lyapunov function, $L(\mathbf{v} \in \mathbb{R}^n)$ (equation 5.8), which takes the parameters, $\mathbf{m} \in \mathbb{R}^n$ and $\mathbf{k} \in \mathbb{R}^n$, the components of which represent the maximum value and maximum slope of each dimension of the function. A single dimension is plotted in figure 5.2.

[4]DLR's control system for the Justin humanoid is safer than MoBeE because it can sense when there is too much energy in the system, for example when a human comes along and shoves the arm toward the body, and apply brakes. I do not have access to a robot with brakes, so I have implemented no such emergency stop, and am content that MoBeE protects the robot from itself under normal operating conditions.

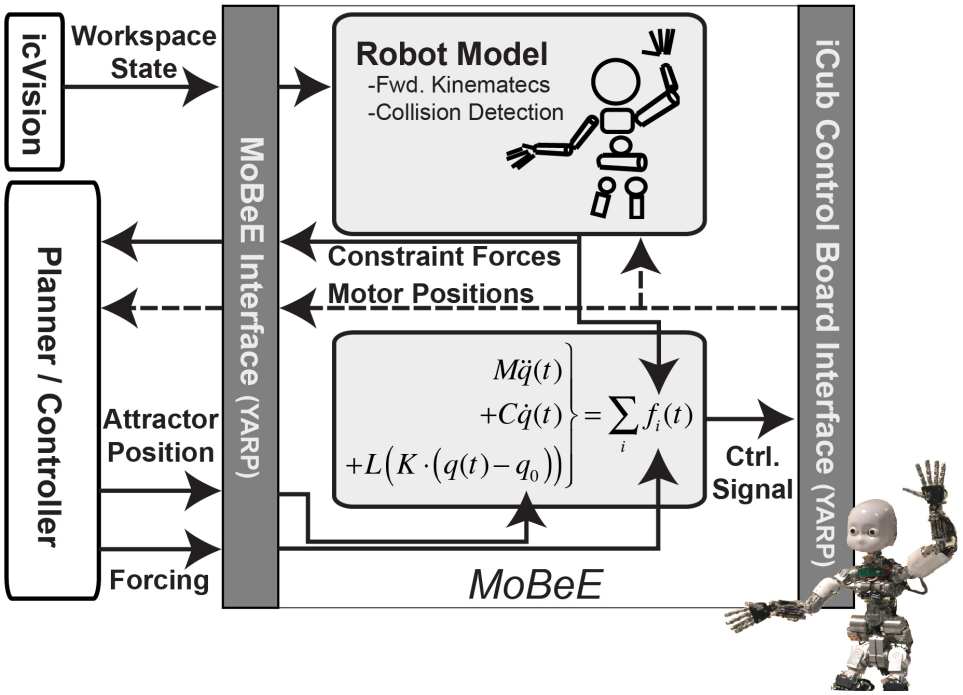


Figure 5.1. MoBeE 2.0 Simplified Architecture - The robot is controlled by a dynamical system (center-bottom, section 5.2), where $q(t)$ is its configuration over time. $M$, $C$, and $K$, are matrices containing mass, damping, and spring constants, respectively. $q_0$ is an attractor, which can be set by the client program. Additionally, the client can force the dynamical system directly. Meanwhile, the robot model also forces the system to affect geometric and kinematic constraints.

$$\mathbf{L}(\mathbf{v} \in \mathbb{R}^n) = \left\{ \begin{array}{c} \frac{m_1 \cdot v_1}{\frac{m_1}{k_1} + |v_1|} \\ \frac{m_2 \cdot v_2}{\frac{m_2}{k_2} + |v_2|} \\ \vdots \\ \frac{m_n \cdot v_n}{\frac{m_n}{k_n} + |v_n|} \end{array} \right\} \tag{5.8}$$

It should be noted that $\mathbf{L}(\mathbf{v})$ does not preserve the direction of $\mathbf{v}$. While this dimension-wise squashing works well for the applications presented in the remainder of this thesis, it may not work well for all applications, and future versions of MoBeE may offer different thresholding functions, which can be configured/selected at runtime.

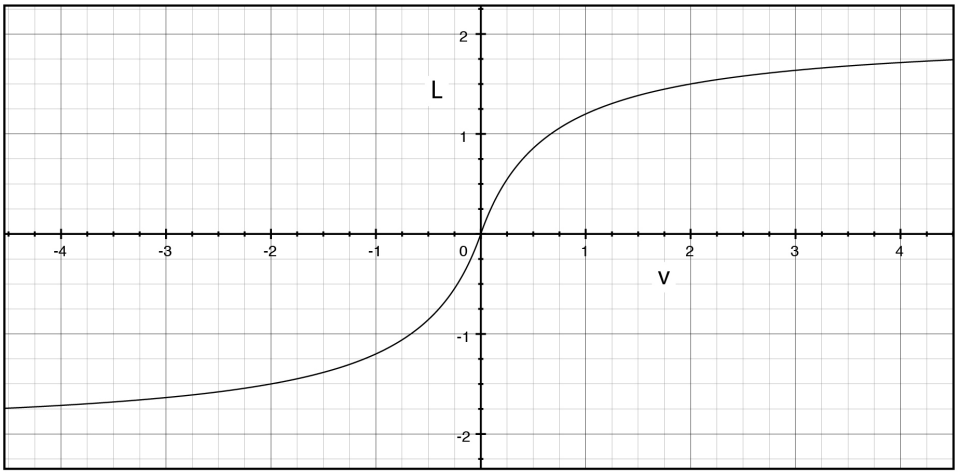


Figure 5.2. 1D Sigmoidal Lyapunov Function - $L(v) = \frac{m \cdot v}{\frac{m}{k} + |v|}$. Here, the maximum value is $m = 2$, and the maximum slope is $k = 3$.

### 5.2.2 Configuration Space Forcing

The client program can force MoBeE's dynamical system in configuration space directly by providing a force, $\mathbf{f}_{\mathbf{qRPC}} \in \mathbb{R}^n$, via RPC. For obvious reasons, it is of critical importance that $\mathbf{f}_{\mathbf{qRPC}}$ not dominate the constraint forces, however the burden on the client program to know about the internals of MoBeE should be minimized. Therefore MoBeE squashes $\mathbf{f}_{\mathbf{qRPC}}$ with $L$.

$$\mathbf{f}_{\mathbf{config}} = \mathbf{L}(\mathbf{f}_{\mathbf{qRPC}}) \tag{5.9}$$

### 5.2.3 Workspace Forcing

The client program can also force the robot in workspace over RPC by applying a force $\mathbf{f}_{\mathbf{m}} \in \mathbb{R}^3$ to one or more markers, $m$, specified in the robot model XML (section 2.3.1). In this case, MoBeE projects the forces into configuration space, and sums them. Similarly to the configuration space forces, these may be arbitrarily large, and must therefore be squashed by $L$. The workspace forcing term is therefore computed:

$$\mathbf{f}_{\mathbf{work}} = \mathbf{L}\left(\sum_m J(m)^T \mathbf{f}_{\mathbf{m}}\right) \tag{5.10}$$

where $J(m)$ represents the Jacobian matrix of some marker, $m$, computed for the current robot pose.

Note that the projection is done via the Jacobian transpose, which is well suited for collision avoidance but not necessarily so for pursuing target positions

in workspace. In the event that the Jacobian transpose projection is inadequate for the user's purposes, a client program can query the robot model for the robot state, including the Jacobian of any link, and compute any kind of optimization on the client side (thus not burdening MoBeE, which must run fast to prevent collisions), and finally forcing MoBeE in configuration space directly.

### 5.2.4 Collision Avoidance

The forces generated by collisions are very similar to the workspace forcing by the client, except that they are computed automatically by the kinematic model. Because the geometries in the model are user-defined, they can in principal be arbitrarily large and therefore, so can the interference volumes between them. Therefore, they too must be squashed by $L$.

To each colliding geometry, $g$, is applied a force $f_g \in \mathbb{R}^3$, which is parallel and proportional to a penetration vector approximated by the SOLID library. Like the workspace forcing commands, these collision forces are projected into configuration space and summed according to:

$$\mathbf{f_{coll}} = L\left(\sum_g J(g)^T \mathbf{f_g}\right) \tag{5.11}$$

where $J(g)$ represents the Jacobian matrix of some a colliding geometry, $g$, computed for the current robot pose.

### 5.2.5 Configuration Space Positional Constraints

In contrast to the above forces, positional constraints in configuration space are implemented by piecewise linear forcing functions. These may go to infinity as the independent variable goes to infinity, but this never happens in practice because the robot may never be in a configuration outside its joint limits. Therefore, there is no need to threshold configuration space positional constraints.

#### Joint Limit Avoidance

The forcing function that implements joint limit avoidance is formulated around the limits of each joint, $q_{min,i}$ and $q_{max,i}$, the distance over which the linear spring acts, $\delta_i$, and the maximum force, $f_{max,i}$, exerted when the joint is at its limit. Each dimension is defined:

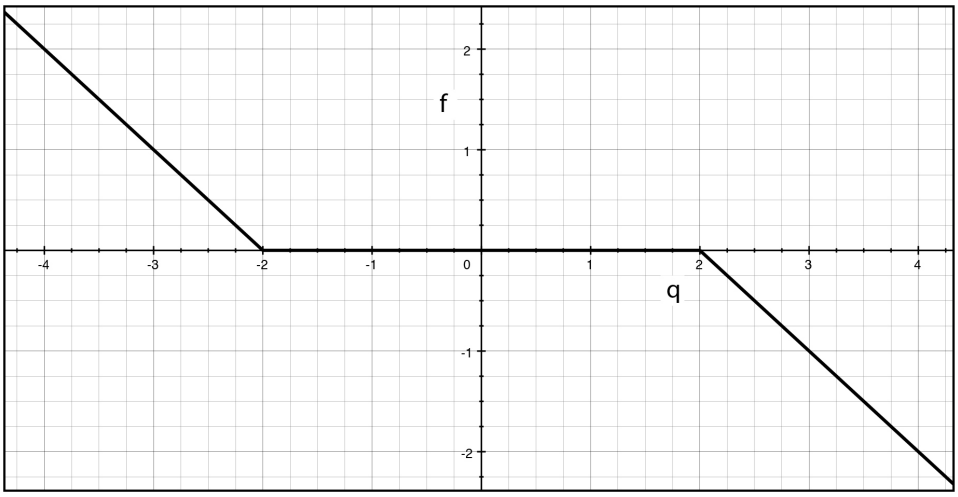


Figure 5.3. 1D Joint Limit Avoidance Function - Plot of equation 5.15 with joint limits, $q_{min,i} = -3$ and $q_{max,i} = 3$, spring distance, $\delta = 1$, and maximum force, $f_{max} = 1$ (at the joint limits).

$$f_{lim,i} = \begin{cases} \frac{f_{max,i}}{\delta_i}(q_i - q_{min,i}) + f_{max,i}, & \text{if } q_i \leq q_{min,i} + \delta_i \\ 0, & \text{if } q_{max,i} + \delta_i > q_i > q_{min,i} + \delta_i \\ \frac{f_{max,i}}{\delta_i}(q_i - q_{max,i}) - f_{max,i}, & \text{if } q_i \geq q_{max,i} - \delta_i \end{cases} \tag{5.12}$$

Here, $f_{max,i} \geq 0$ and $\delta_i \geq 0$. An instructive plot of $f_{lim,i}$ is provided in figure 5.3. The aggregate joint limit avoidance force is defined:

$$\mathbf{f_{lim}} = \begin{Bmatrix} f_{lim,1} \\ f_{lim,2} \\ \vdots \\ f_{lim,n} \end{Bmatrix} \tag{5.13}$$

Where $n$ is the dimensionality of $\mathbf{q}$.

### Linear System Constraints

The XML specification for robot modeling (section 2.3.1) supports the definition of linear constraints of the form:

$$\mathbf{n} \cdot \mathbf{q} < b \tag{5.14}$$

The vector, $\mathbf{n}$, which is normal to the hyperplane, and the scalar $b$, are chosen by the user. Together they define a half-space, into which points the other hyperplane normal, $-\mathbf{n}$. Associated with the constraint is a maximum constraint force, $f_{max}$, and a spring length, $\delta$, that defines the distance over which the repulsive force is active. A single constraint force, $\mathbf{f_{cst,i}}$, is defined as follows:

$$\mathbf{f_{cst,i}} = -\frac{\mathbf{n}}{|\mathbf{n}|} \cdot \begin{cases} \frac{-f_{max,i}}{\delta_i}(\mathbf{n}\cdot\mathbf{q} - b) + f_{max,i}, & \text{if } \mathbf{n}\cdot\mathbf{q} - b \leq \delta_i \\ 0, & \text{if } \mathbf{n}\cdot\mathbf{q} - b > \delta_i \end{cases} \tag{5.15}$$

From each set of linear constraints is constructed a conjunction of clauses, where each clause is a disjunction of literals, as shown in these three examples:

$$CNF = \begin{cases} \mathbf{f_{cst,1}} \leftrightarrow 0 \wedge \mathbf{f_{cst,2}} \leftrightarrow 0 \wedge \mathbf{f_{cst,3}} \leftrightarrow 0 \\ \mathbf{f_{cst,1}} \leftrightarrow 0 \wedge \left(\mathbf{f_{cst,2}} \leftrightarrow 0 \vee \mathbf{f_{cst,3}} \leftrightarrow 0\right) \\ \left(\mathbf{f_{cst,1}} \leftrightarrow 0 \vee \mathbf{f_{cst,2}} \leftrightarrow 0\right) \wedge \left(\mathbf{f_{cst,3}} \leftrightarrow 0 \vee \mathbf{f_{cst,4}} \leftrightarrow 0 \vee \mathbf{f_{cst,5}} \leftrightarrow 0\right) \end{cases} \tag{5.16}$$

This is known as conjunctive normal form (CNF), and whether it evaluates true or not controls whether or not the aggregate constraint force is applied to the robot.

$$\mathbf{f_{cst}} = \begin{cases} \sum_i \mathbf{f_{cst,i}}, & \text{if } CNF \\ 0, & \text{if } \neg CNF \end{cases} \tag{5.17}$$

In this way, either all of the constraint forces in the set are applied, or none of them are. Such a system of linear constraints is used to repel the iCub from an infeasible region of configuration space, which arises from the relative lengths of the cables that actuate its shoulder. The iCub shoulder constraint, which is also respected by much of the IIT control code, uses conjunction of literals only, however MoBeE exploits the CNF to do more complex things, such as constrain the forearm not to rotate too much when the elbow is bent too much[5].

## 5.3 MoBeE 2.0 Discussion

The central idea behind the MoBeE system from the beginning has been to facilitate exploratory behavior using a real robot. MoBeE has acted as a supervisor,

[5]On the iCub, turning the forearm with the elbow bent too much causes collisions between adjacent body panels which are hard to model geometrically.

intervening when the client program, which I have been calling a planner/controller, does something dangerous or undesirable. The first implementation of MoBeE did that in a discrete way, via switching control, but the result was a jerky, unnatural reflexive behavior, which was not very robust.

MoBeE 2.0 does away with switching control in favor of a dynamical system, which continuously mixes control and constraint forces to generate the robot motion in real time. The most obvious result of this is smoother, more intuitive motions in response to constraints/collisions. Many of the collisions encountered in practice no longer stop the robot's forward progress, but rather deflect the requested motion, bending it around an obstacle. Of course this is not always possible, and sometimes the constraint forces slow the robot to a stop near the encountered obstacle. However even this is quite smooth, and much more intuitive to an observer than the MoBeE 1.0 switching control.

In addition to smoothness and intuitiveness of the motions generated, MoBeE 2.0 offers drastically improved robustness. Without the need to invert the recent robot motion, there is no longer any problems related to hysteresis. Even the effects of sensory noise are mitigated. Because the constraint forces associated with collisions are proportional to their penetration depth, noise in the motor encoder signal has a minimal effect on collision response. The sporadic shallow collisions, which can be observed when the robot is operating very near to an obstacle, generate tiny forces that only serve to nudge the robot gently away from the obstacle.

The theoretical implications of MoBeE 2.0 on adaptive roadmap planning are very promising indeed. Essentially, the dynamical approach to enforcing constraints means that the planner/controller is free to explore continuous spaces, without the need to divide them into safe and unsafe regions.

Whereas the original roadmap planning required complete knowledge that all the trajectories in the roadmap graph are certainly safe, *adaptive* roadmap planning with MoBeE 1.0 required that only one trajectory is safe, namely the inverse of the recent robot motion. Now that MoBeE 2.0 eliminates the need to know even that, roadmap planning is no longer about safety at all. The burden of collision avoidance is completely lifted from the planner/controller, and the state of the robot must not necessarily be on the roadmap in any sense. In fact, the roadmap is no longer necessary at all; it is just about best practice. If the planner/controller plans a motion using the roadmap, and it does not work out, that is not a problem. It only means that the planner should be updated to reflect the new experience.

Unburdened of the need to know about every single pose in which the robot will ever be, the adaptive roadmap planner is now free to focus on that for which

it is best suited, coarse path planning. Many thousands of states/vertices are no longer required, as in section 4.3.3 to cover a table for example. Instead, a hybrid solution, wherein coarse planning is provided by a sparse roadmap, and the final reach to grasp is provided my a more reactive approach, is now possible under the abstract formalism of RL.

# Chapter 6

# An RL Agent for MDP Roadmap Planning

Allowing the planner to issue control commands directly offers considerable benefits, but it also requires a more complex representation of the configuration space than the *plan first, act later* paradigm did. Whereas the PRM planner made do with a simple graph, representing a network of trajectories, the embodied version seems to require a probabilistic model, which can cope with actions that may have a number of different outcomes. In light of this requirement, the embodied planner begins to look like a Markov Decision Process (MDP), and in order to exploit such a planner, the state transition probabilities, which govern the MDP, must first be learned.

The new *embodied* MDP planner differs from its antecedent PRM planner in several important ways. There is no need to require that the configuration of the robot be *on* any of the graph edges. In fact the graph no longer represents a network of trajectories, but rather the *topology* of the continuous configuration space. Instead of a trajectory, each edge represents a more general kind of *action* that implements something like *try to go to that region of the configuration space*. Such actions are available not when the true robot configuration is *on* a graph vertex, but rather when it is *near* that vertex. The actions may or may not succeed depending on the particular initial configuration of the robot when the action was initiated as well as the configuration of the workspace, which must not necessarily be static.

## 6.1 Action Implementation

For the purposes of MDP roadmap planning, an action will be defined as setting MoBeE's attractor $q^*$ (equation 5.7) to some desired configuration. When such an action is taken, $q(t)$ begins to move toward $q^*$. The action terminates either when the dynamical system settles or when a timeout occurs. The action may or may not settle on $q^*$, depending on what constraint forces (right hand side of equation 5.7) are encountered during the transient response. The role of the RL agent, is accordingly to model the topology of the configuration space using an MDP.

## 6.2 State-Action Space

The true configuration of the robot at any time $t$ can be any real valued $q \in \mathbb{R}^n$, however in order to define a tractable RL problem, the configuration space is discretized by selecting $m$ samples, $Q = \{q_j | j = 1 \dots m\} \subset \mathbb{R}^n$, which define a set of Voronoi regions $\{s_j | j = 1 \dots m\}$[1]. That is to say, with each sample, $q_j \in \mathbb{R}^n$, is associated an $s_j \subset \mathbb{R}^n$, where every point, $q \in s_j$, is closer to $q_j$ than to any other point $q \in Q$.

The state space of the Markov model comprises the sets $s_j$, not the points, $q_j \in Q$. More formally, the state space $S = \{s_j | j = 1 \dots m\}$. Throughout the remainder of this proposal, I will use the conventional RL notation, dropping the subscript, $j$, and referring to states $s \in S$. To say that the robot is in some state, $s$, at some time, $t$, means that the real valued configuration of the robot, $q(t) \in s$. The notation $s_j$ will be reserved for indexing sets of states as was done above for the set of all states, $S$.

An action is defined by setting MoBeE's attractor, $q^* = q_g$ (eq. 5.7), where $q_g \in Q$ is the sample in some goal state $s_g(a)$. When an action is tried, the robot moves according to the transient response, $q(t)$, of the dynamical system, which eventually settles at $q(t \to \infty) = q_\infty$. However, depending on the constraint forces encountered, it may be that $q_\infty \in s_g(a)$ or not. A diagram of the discrete state-action space is provided in figure 6.1.

[1] Generally, throughout this formalism I use uppercase letters to denote sets and lowercase letters to denote points. However, I have made an exception for the regions, $s_j \subset \mathbb{R}^n$, which themselves comprise sets of robot configurations. Although this is somewhat abusive from a set theoretic standpoint, it allows us to be consistent with the standard RL notation later in the paper.

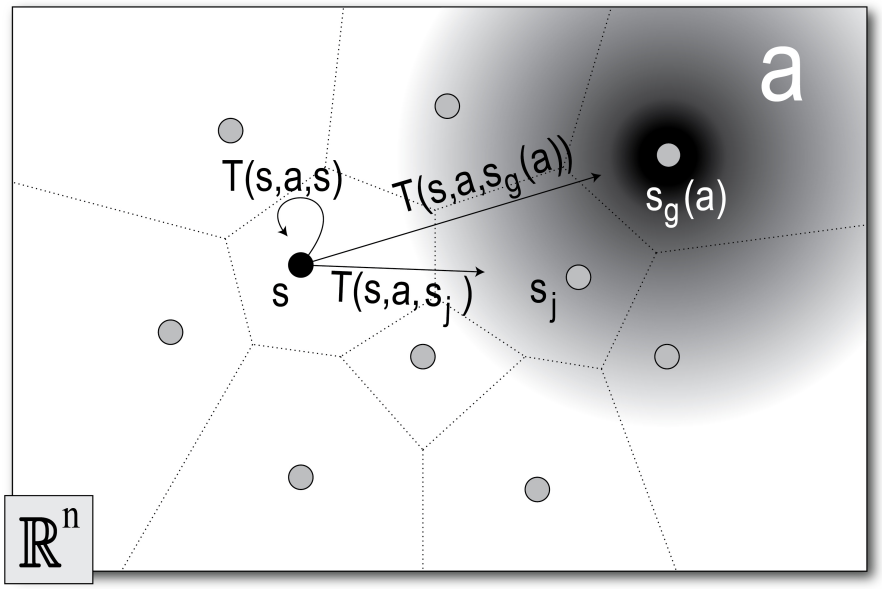


Figure 6.1. The discrete state-action space : The sample set $Q = \{q_j | j = 1 \dots m\}$ (dots) defines the Voronoi regions, or states $S = \{s_j | j = 1 \dots m\}$ (bounded by dotted lines). An action $a$ (gradient), exploits MoBeE's attractor dynamics to pull the robot toward some goal state, $s_g(a) \in S$. When the robot is in the initial state, $q(t_0) \in s$, and the agent selects $a$, MoBeE switches on the attractor (eq. 5.7) at the point $q_g \in s_g(a)$. The agent then waits for the dynamical system to settle or for a timeout to occur, and at some time, $t_1$, checks which of the states, $s_j$ contains the final real valued configuration of the robot, $q(t_1)$. Often the state-action, $(s, a)$, terminates in the goal state $s_g(a)$, but sometimes, due to constraint forces, it does not. This gives rise to a set of state transition probabilities $T(s, a) = \{T(s, a, s'_1), T(s, a, s'_2), \dots, T(s, a, s'_m)\}$, which correspond to the states, $\{s_j | j = 1 \dots m\}$.

## 6.3 Connecting States with Actions

With each state, $s$, is associated a set of actions, $A(s)$, which *intend* to move the robot from $s$ to each of $k$ nearby goal states, $A(s) = \{a_g | g = 1 \dots k\}$, and the set of all possible actions, $A$, can therefore be expressed as the union of the action sets belonging to each state, $A = \bigcup_{s=1}^{m} A(s)$.

This notion of connecting neighboring states makes intuitive sense given the problem domain at hand and the resulting Markov model resembles the Roadmap graph used by the PRM planner [Latombe et al., 1996]. Although the action set, $A$, is quite large ($|A| = |S|$), each state only has access to the actions,

$A(s)$, which lead to its $k$ nearest neighbors ($|A(s)| = k$). Therefore, the number of state-actions remains linear in the number of states. The reader should be advised that wherever the standard state-action notation, $(s, a)$, is used, it is implied that $a \in A(s)$.

## 6.4 Modeling Transition Probabilities

Although each action *intends* to move the robot to some particular goal state, in principal they can terminate in *any* state in the set $\{s_j | j = 1 \ldots m\}$. Therefore, state transition probabilities must be learned to represent the connectivity of the configuration space. A straightforward way of doing this would be to define a probability distribution over all possible outcomes $s_j$ for each state-action $(s, a)$:

$$T(q_\infty \in s_j | s, a) = \left\{ \begin{array}{c} p(q_\infty \in s_1 | s, a) \\ p(q_\infty \in s_2 | s, a) \\ \vdots \\ p(q_\infty \in s_m | s, a) \end{array} \right\} \tag{6.1}$$

To build up the distributions, $T(q_\infty \in s_j | s, a)$, one would simply initialize all probabilities to zero and then count the occurrences of observed transitions to the various states, $s_j$, resulting from the various state-actions $(s, a)$. However, this approach would be relatively wasteful, because much of the state-action space is deterministic. In practice, there are only three kinds of distributions that come out of applying the Markov model to motion planning. A state-action, $(s, a)$, can terminate deterministically in the goal state $s_g(a)$ (eq. 6.2), it can terminate deterministically in some other state $s_j \neq s_g(a)$ (eq. 6.3), or it can be truly nondeterministic (eq. 6.4), although the nonzero components of $T$ are always relatively few compared to the number of states in the model.

$$p(q_\infty \in s_j | s, a) = \left\{ \begin{array}{ll} 1 & \text{if } s_j = s_g(a) \\ 0 & \text{if } s_j \neq s_g(a) \end{array} \right. \tag{6.2}$$

$$p(q_\infty \in s_j | s, a) = \left\{ \begin{array}{ll} 1 & \text{if } s_j = s^* \neq s_g(a) \\ 0 & \text{if } s_j \neq s^* \end{array} \right. \tag{6.3}$$

$$p(q_\infty \in s_j | s, a) \left\{ \begin{array}{ll} = 0 & \text{if } s_j \in S_{\text{infeasible}} \\ > 0 & \text{if } s_j \in S_{\text{feasible}} \end{array} \right. \tag{6.4}$$

This is intuitive upon reflection. Much of the configuration space is not affected by constraints, and actions always complete as planned. Sometimes

constraints are encountered, such as joint limits and cable length infeasibilities, which deflect the trajectory in a predictable manner. Only when the agent encounters changing constraints, typically non-static objects in the robot's operational space, does one see a variety of outcomes for a particular state-action. However even in this case, the possible outcomes, $s'$, are a relatively small number of states, which are usually in the neighborhood of the initial state, $s$. I have never constructed an experiment, using this framework, in which a particular state-action, $(s, a)$, yields more than a handful of possible outcome states, $s'$.

### 6.4.1 Artificial Curiosity

What is interesting? For us humans, *interestingness* seems closely related to the rate of our learning progress [Schmidhuber, 2006]. If we try doing something, and we rapidly get better at doing it, we are often interested. Contrastingly, if we find a task trivially easy, or impossibly difficult, we do not enjoy a high rate of learning progress, and are often bored. This behavior makes humans efficient explorers of novel behavior, because we do not waste time on things already learned or things that are too difficult for us.

Experiments run on robotic hardware are slow. They are bound to real time, as opposed to simulations, which can be run faster than real time, or parallelized, or both. Therefore, as for us humans, efficient exploration is critical. This motivates the use of curiosity-driven reinforcement learning, which I have implemented using the information theoretic notion of *information gain*, or Kullback-Leibler (KL) divergence.

### 6.4.2 KL Divergence

KL Divergence, $D_{KL}$ is defined as follows, where $P_j$ and $T_j$ are the scalar components of the discrete probability distributions $P$ and $T$, respectively.

$$D_{KL}(P||T) = \sum_j \ln\left(\frac{P_j}{T_j}\right) P_j \tag{6.5}$$

For MDP roadmap planning, $T$ represents the estimated state transition probability distribution (eq. 6.1) for a particular state-action, $(s, a)$, after the agent has accumulated some amount of experience. Once the agent tries $(s, a)$ again, an $s'$ is observed, and the state transition probability distribution for $(s, a)$ is updated. This new distribution, $P$, is a better estimate of the state transition probabilities for $(s, a)$, as it is based on more data.

By computing $D_{KL}(P||T)$, one can measure how much the Markov model improved by trying the state-action, $(s, a)$, and this *information gain* can be used to reward the curious agent. Thus, the agent is motivated to improve its model of the state-action space, and it will gravitate toward regions thereof, where learning is progressing quickly.

There is however, a problem. The KL divergence is not defined if there exist components of $P$ or $T$, which are equal to zero. This is somewhat inconvenient in light of the fact that for the proposed application, most of the components of most of the distributions, $T$ (eq. 6.1), *are* actually zero. $P$ and $T$ must therefore be initialized cleverly.

Perhaps the most obvious solution would be to initialize $T$ with a uniform distribution, before trying some action for the first time. After observing the outcome of the selected action, $P$ would be defined and $D_{KL}(P||T)$ computed, yielding the *interestingness* of the action taken.

Some examples of this kind of initialization are given in equations 6.6-6.9.[2] Clearly the approach solves the numerical problem with the zeros, but it means that initially, *every* action the agent tries will be equally interesting. Moreover, *how* interesting those first actions are, $|D_{KL}(P||T)|$, depends on the size of the state space.

$$D_{KL}(\{1,2,1\}||\{1,1,1\}) = 0.0589 \tag{6.6}$$

$$D_{KL}(\{2,1,1\}||\{1,1,1\}) = 0.0589 \tag{6.7}$$

$$D_{KL}(\{1,1,2,1,1\}||\{1,1,1,1,1\}) = 0.0487 \tag{6.8}$$

$$D_{KL}(\{1,1,1,2,1,1,1\}||\{1,1,1,1,1,1,1\}) = 0.0398 \tag{6.9}$$

The first two examples, eq. 6.6 and eq. 6.7, show that regardless of the outcome, all actions generate the same numerical interestingness the first time they are tried. While not a problem in theory, in practice this means the robot will need many tries to gather enough information to differentiate the boring, deterministic states from the interesting, nondeterministic ones. Since the actions are designed to take the agent to a goal state, $s_g(a)$, it would be intuitive if observing a transition to $s_g(a)$ were less interesting than observing one to some other state. This would drastically speed up the learning process.

The second two examples, eq. 6.8 and eq. 6.9 show that the *interestingness* of that first try decreases in larger state spaces, or alternatively, small state spaces

[2] I have intentionally not normalized $P$ and $T$, to show how they are generated by counting observations of $q_\infty \in s_j$. In order to actually compute $D_{KL}(P||T)$, $P$ and $T$ must first be normalized.

are numerically more *interesting* than large ones. This is not a problem if there is only one learner operating in a single state-action space. However in the case of a multi-agent system, say one learner per body part, it would be convenient if the intrinsic rewards gotten by the different agents were numerically comparable to one another, regardless of the relative sizes of those learners' state-action spaces.

In summary, there are two potential problems with KL Divergence as a reward signal:

1. Slowness of initial learning
2. Sensitivity to the cardinality of the distributions

Nevertheless, in many ways, KL Divergence captures exactly what I would like our curious agent to focus on. It turns out that both of these problems can be addressed by representing $T$ with an array of variable size, and initializing the distribution optimistically with respect to the expected behavior of the action $(s, a)$.

### 6.4.3 'Kail' Divergence

By compressing the distributions $T$ and $P$, i.e. not explicitly representing any bins that contain a zero, the KL divergence can be computed between only their non-zero components. The process begins with $T$ and $P$ having no bins at all. However they grow in cardinality as follows: Every time a novel $s'$ is observed as the result of trying a state-action $(s, a)$, a new bin is appended to the distribution $T(s, a)$, and initialized with a 1. Then, $T(s, a)$ is copied to yield $P(s, a)$, and the $s'$ bin is incremented in $P(s, a)$. Finally, $KL(P||T)$ is computed. This process is formalized in algorithm 6.

The optimistic initialization is straightforward. Initially, the distribution $T(s, a)$ is empty. Then it is observed (algorithm 6) that $(s, a)$ fails, leaving the agent in the initial state, $s$. The KL divergence between the trivial distributions $\{1\}$ and $\{2\}$ is 0, and therefore, so is the reward, $R(s, a)$. Next, it is observed that $(s, a)$ succeeds, moving the agent to the intended goal state, $s_g(a)$. The distribution, $T(s, a)$, becomes nontrivial, a nonzero KL divergence is computed, and thus $R(s, a)$ gets an optimistically initialized reward, which does not depend on the size of the state-action space. Algorithm 8 describes the steps of this optimistic initialization, and table 6.1 shows how $T(s, a)$ and $R(s, a)$ develop throughout the initialization process.

The distributions $T$, as initialized above, are compact and parsimonious, and they faithfully represent the most likely outcomes of the actions. Moreover, the

second initialization step yields a non-zero KL Divergence, which is not sensitive to the size of the state space. Importantly, the fact that the initialization of the state transition probabilities provides an initial measure of *interestingness* for each state-action allows, *without choosing parameters*, the reward matrix to be initialized optimistically with well defined *intrinsic rewards*.

Consequently, a greedy policy can be employed to aggressively explore the state-action space while focusing extra attention on the most *interesting* regions. As the curious agent explores, the intrinsic rewards decay in a logical way. A state-action, which deterministically leads to its goal state (table 6.2) is less interesting over time than a state-action that leads to some other state (table 6.3), and of course most *interesting* are state-actions with more possible outcomes (table 6.4).

| Observation | $T$ | $P$ | $R = D_{KL}(P\|\|T)$ |
|---|---|---|---|
| - | {} | {} | - |
| $s_i$ | {1} | {2} | 0 |
| $s_g(a)$ | {2,1} | {2,2} | 0.0589 |

Table 6.1. Initialization of state transition probabilities

| Observation | $T$ | $P$ | $R = D_{KL}(P\|\|T)$ |
|---|---|---|---|
| init | {2,1} | {2,2} | 0.0589 |
| $s_g(a)$ | {2,2} | {2,3} | 0.0201 |
| $s_g(a)$ | {2,3} | {2,4} | 0.0095 |
| $s_g(a)$ | {2,4} | {2,5} | 0.0052 |

Table 6.2. A predictable action ends in the predicted state

| Observation | $T$ | $P$ | $R = D_{KL}(P\|\|T)$ |
|---|---|---|---|
| init | {2,1} | {2,2} | 0.0589 |
| $s_j$ | {2,2,1} | {2,2,2} | 0.0487 |
| $s_j$ | {2,2,2} | {2,2,3} | 0.0196 |
| $s_j$ | {2,2,3} | {2,2,4} | 0.0103 |

Table 6.3. A predictable action ends in a surprising state

| Observation | $T$ | $P$ | $R = D_{KL}(P\|\|T)$ |
|---|---|---|---|
| init | {2,1} | {2,2} | 0.0589 |
| $s_a$ | {2,2,1} | {2,2,2} | 0.0487 |
| $s_b$ | {2,2,2,1} | {2,2,2,2} | 0.0345 |
| $s_c$ | {2,2,2,2,1} | {2,2,2,2,2} | 0.0283 |
| $s_g(a)$ | {2,2,2,2,2} | {2,3,2,2,2} | 0.0142 |
| $s_a$ | {2,3,2,2,2} | {2,3,3,2,2} | 0.0133 |
| $s_b$ | {2,3,3,2,2} | {2,3,3,3,2} | 0.0125 |

Table 6.4. An unpredictable action

## 6.5 Reinforcement Learning

In this thesis, I claim that a PRM planner's compact, incrementally expandable representation of known motions makes it a likely antecedent to a developmental learning system. Furthermore, I observe that many of the weaknesses of PRMs can be avoided by *embodying the planner* and coupling it to a low-level reactive controller. Proxied by this low-level controller, the planner is empowered to try out arbitrary control signals, however it does not necessarily know what will happen. Therefore, the PRM's original model of the robot's state-action space, a simple graph, is insufficient, and a more powerful, probabilistic model, an MDP is required. Thus, modeling the robot-workspace system using an MDP arises naturally from the effort to improve the robustness of a PRM planner, and accordingly, Model-Based RL is the most appropriate class of learning algorithms to operate on the MDP.

Having specified what *action* means in terms of robot control (section 6.1), described the layout and meaning of the state-action space (section 6.2), and defined the way in which intrinsic reward is computed according to the artificial curiosity principal (section 6.4.1), I will now incorporate these pieces in a Model-Based RL system, which develops a path planner as follows.

Initially, sets of states and actions are chosen, according to some heuristic(s), such that the robot's configuration space is reasonably well covered and the RL computations are tractable. Then, the state transition probabilities are learned for each state-action pair, as the agent explores the MDP by moving the robot about. This exploration for the purposes of model learning is guided entirely by the intrinsic reward, and the curious agent continually improves its model of the iCub's configuration space. In order to exploit the planner, an external reward must be introduced, which can either be added to or replace the intrinsic reward

**Algorithm 6:** Observe($s$,$a$,$s'$,$T(s,a)$,$R(s,a)$) is responsible for processing occurrences of state transitions, $(s,a,s')$, and learning the MDP. The curious agent is rewarded according to learning rate, as defined by KL Divergence: $D_{KL}(P||T(s,a)) = \sum_{s'} \ln\left(\frac{P(s,a)_{s'}}{T(s,a)_{s'}}\right) P(s,a)_{s'}$ where $T(s,a)_{s'}$ and $P(s,a)_{s'}$ are the scalar components of the discrete probability distributions $T(s,a)$ and $P(s,a)$, respectively. By computing $D_{KL}(P(s,a)||T(s,a))$, we quantify how much the Markov model improved by trying the state-action, $(s,a)$, and this *information gain* is used to reward the curious agent.

**begin**
  **if** *there is no bin,* $T(s,a)_{s'}$*, in* $T(s,a)$ *to count occurrences of* $s'$ **then**
    append a bin, $T(s,a)_{s'}$ to $T(s,a)$
    $T(s,a)_{s'} \leftarrow 1$
  **end**
  $P(s,a) \leftarrow T(s,a)$
  $P(s,a)_{s'} \leftarrow P(s,a)_{s'} + 1$
  $R(s,a) \leftarrow D_{KL}(P(s,a)||T(s,a))$
  $T(s,a) \leftarrow P(s,a)$
**end**

function.

The MDP, which constitutes the path planner, is a tuple, $\langle S,A,T,R,\gamma\rangle$, where $S$ is a finite set of $m$ states, $A$ is a finite set of actions, $T$ is a set of state transition probability distributions, $R$ is a reward function, and $\gamma$ is a discount factor, which represents the importance of future rewards. This MDP is somewhat unusual in that not all of the actions $a \in A$ are available in every state $s \in S$. Therefore, I define sets, $A(s)$, which comprise the actions $a \in A$ that are available to the agent when it finds itself in state $s$, and $A = \bigcup_{s=1}^{m} A(s)$. The set of state transition probabilities becomes $T : \bigcup_{s=1}^{m} A(s) \times S \rightarrow [0,1]$, and in general, the reward function becomes $R : \bigcup_{s=1}^{m} A(s) \times S \rightarrow \mathbb{R}$, although the intrinsic reward, $R_{\text{intrinsic}} : \bigcup_{s=1}^{m} A(s) \rightarrow \mathbb{R}$, varies only with state-action pairs $(s,a)$, as opposed to state-action-state triples $(s,a,s')$. The state transition probabilities, $T$, are learned by curious exploration (algorithm 8, $\gamma = 0.9$, $\delta = 0.001$), the RL algorithms employed is value iteration (algorithm 7), and the intrinsic reward is computed as shown in algorithm 6.

**Algorithm 7:** Value_Iteration(S,A,T,R,$\gamma, \delta$) recomputes the value function $V$ each time an action completes.

**begin**
  **for** *each state-action* $(s \in S, a \in A(s))$ **do**
    $V(s,a) \leftarrow 0.0$
  **end**
  **for** *each state* $s \in S$ **do**
    $V(s) \leftarrow 0.0$
  **end**
  **while** *true* **do**
    $max\_delta \leftarrow 0.0$
    **for** *each state-action* $(s \in S, a \in A(s))$ **do**
      $V(s,a)_{new} \leftarrow R(s,a) + \gamma \sum_{s'} T(s,a)_{s'} V(s')$
      **if** $V(s,a)_{new} - V(s,a) > max\_delta$ **then**
        $max\_delta \leftarrow V(s,a)_{new} - V(s,a)$
      **end**
      $V(s,a) \leftarrow V(s,a)_{new}$
    **end**
    **for** *each state* $s \in S$ **do**
      $V(s) \leftarrow argmax(\{V(s,a)|i = s\})$
    **end**
    **if** $max\_delta < \delta$ **then**
      $break$
    **end**
  **end**
**end**

**Algorithm 8:** Curious_Explore(S,A,T,R,$\gamma, \delta$) is the top-level behavioral control loop that exploits the other algorithms. It first initializes the MDP optimistically, assuming that each action will either succeed as expected, bringing the agent to the goal state, or fail, leaving the agent at the initial state. After initialization, the agent explores the state-action space pursuing reward, which is defined in terms of *information gain*.

**begin**
  **for** *each state-action* $(s \in S, a \in A(s))$ **do**
    $Observe(s, a, s, T(s,a), R(s,a))$
    $Observe(s, a, s_g(a), T(s,a), R(s,a))$
  **end**
  **while** *true* **do**
    $Value\_Iteration(S, A, T, R, \gamma, \delta)$
    $s \leftarrow s_j | q(t_{before}) \in s_j$
    $a_{greedy} \leftarrow a | V(s,a) = argmax(\{V(s,a) | a \in A(s)\})$
    run $a_{greedy}$ on the robot
    $s' \leftarrow s_j | q(t_{after}) \in s_j$
    $Observe(s, a, s', T(s,a), R(s,a))$
  **end**
**end**

# Chapter 7

# Model Learning Experiments

Here I present the results of a series of experiments, in which MDP motion planners are learned for the iCub humanoid by the RL agent proposed in the previous chapter. The first experiments are simple as they are intended to validate the RL implementation. Subsequent experiments compare the exploratory behavior of the curious agent to more naive strategies and characterize the behavior of the agent under different initializations. Finally, larger, more complex state-action spaces are considered, which constitute a more realistic real-world application, and a multi-agent experiment demonstrates how MDP motion planning may eventually scale up to control the entire humanoid, which, due to the size of a humanoid's configuration space, is beyond any planning approach I am currently aware of.

For a humanoid in the context of manipulation tasks, many of the subproblems can be addressed by reactive, gradient-based methods. Given a region of interest in the visual field(s), the robot's gaze for example, can be controlled quite straightforwardly using some kind of error minimization. Also, if the hand is in some sort of reasonable pre-reach position, its orientation (with respect to a target object) can be fine tuned by a very simple reactive controller. From my point of view, the hard part is finding and getting to that sensible pre-reach position, and the joints most relevant to solving it comprise the shoulder and elbow. Therefore, the MDP planning experiments (aside from the multi-agent one), are designed to facilitate coarse motion planning in the 4D shoulder-elbow space.

## 7.1 Implementation Validation

In the first experiment I validated my RL implementation, using the simplest possible Markov model, which captures the 4D shoulder-elbow configuration

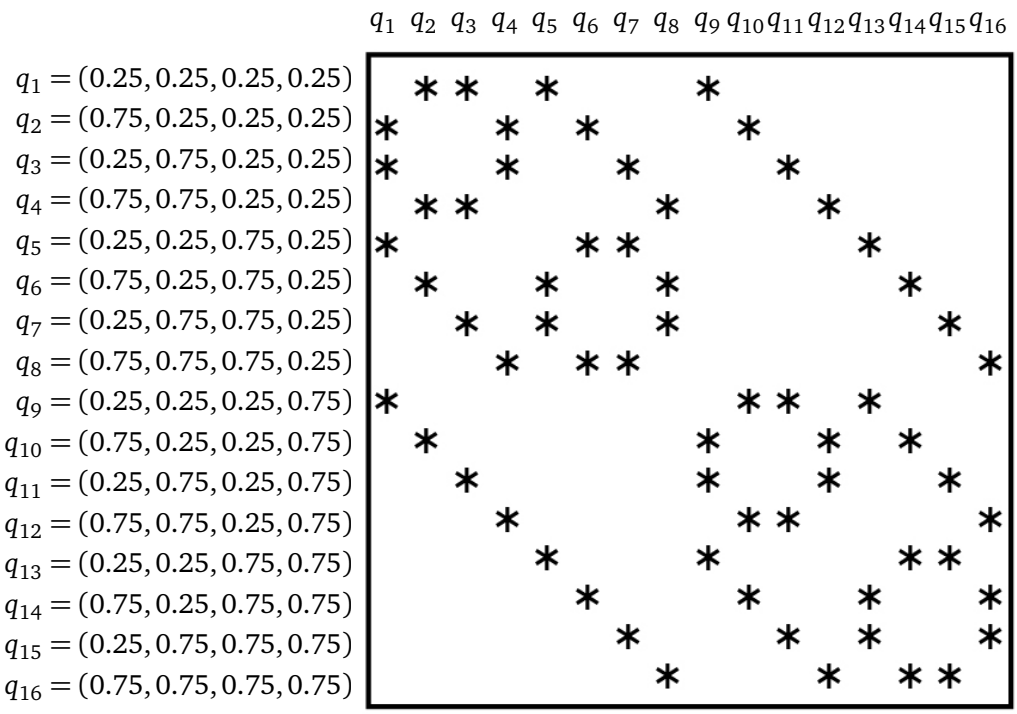


Figure 7.1. Hypercube state-action space for the proof of concept experiment. The samples, $\{q_1, q_2, q_3, \ldots\} = Q \subset \mathbb{R}^n$ (left) define the states in the MDP. The values are normalized to represent percent range of motion corresponding to shoulder flexion/extension, arm abduction/adduction, lateral/medial arm rotation, and elbow flexion/extension, respectively. Actions are defined to take the agent from each state to its 4 nearest neighbors. This connectivity is shown graphically (right) where $*$ indicates that there exists an action, which is available to the agent in state $row$ and intends to bring it to state $column$.

space. The state-action space is shown graphically in figure 7.1.

For this experiment, the workspace was devoid of obstacles, and the states and actions were deliberately chosen such that the agent does not encounter any constraints during the exploration of the model. Therefore, all actions succeeded, bringing the agent to the intended goal state. The state of the robot was initialized randomly, and the agent explored the model until every state-action had been tried at least 5 times. The experiment was repeated many times over, as the discount factor, $\gamma$, was varied. The measured result was the cumulative history of the agent's action selection and the value function, and these were observed over time.

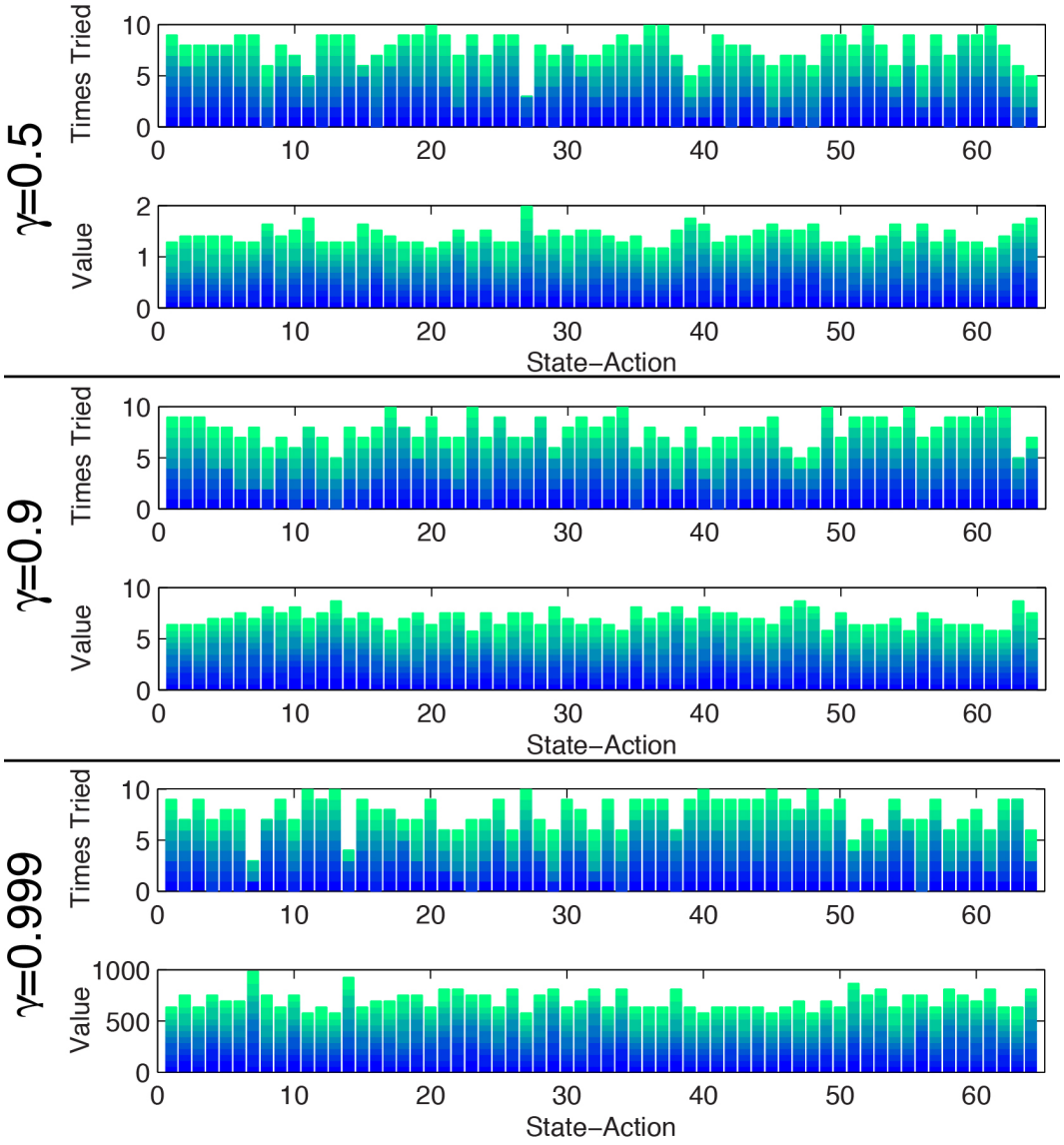


Figure 7.2. Stacked bar charts show the action selection history and value function during the early stages of hypercube exploration, after 50 actions have been executed. Each bar indicates a particular state-action, and each color represents data from an individual randomly initialized experiment. Data are presented for three different discount factors ($\gamma$). At this early stage of exploration, action selection is fairly uniform but very noisy.

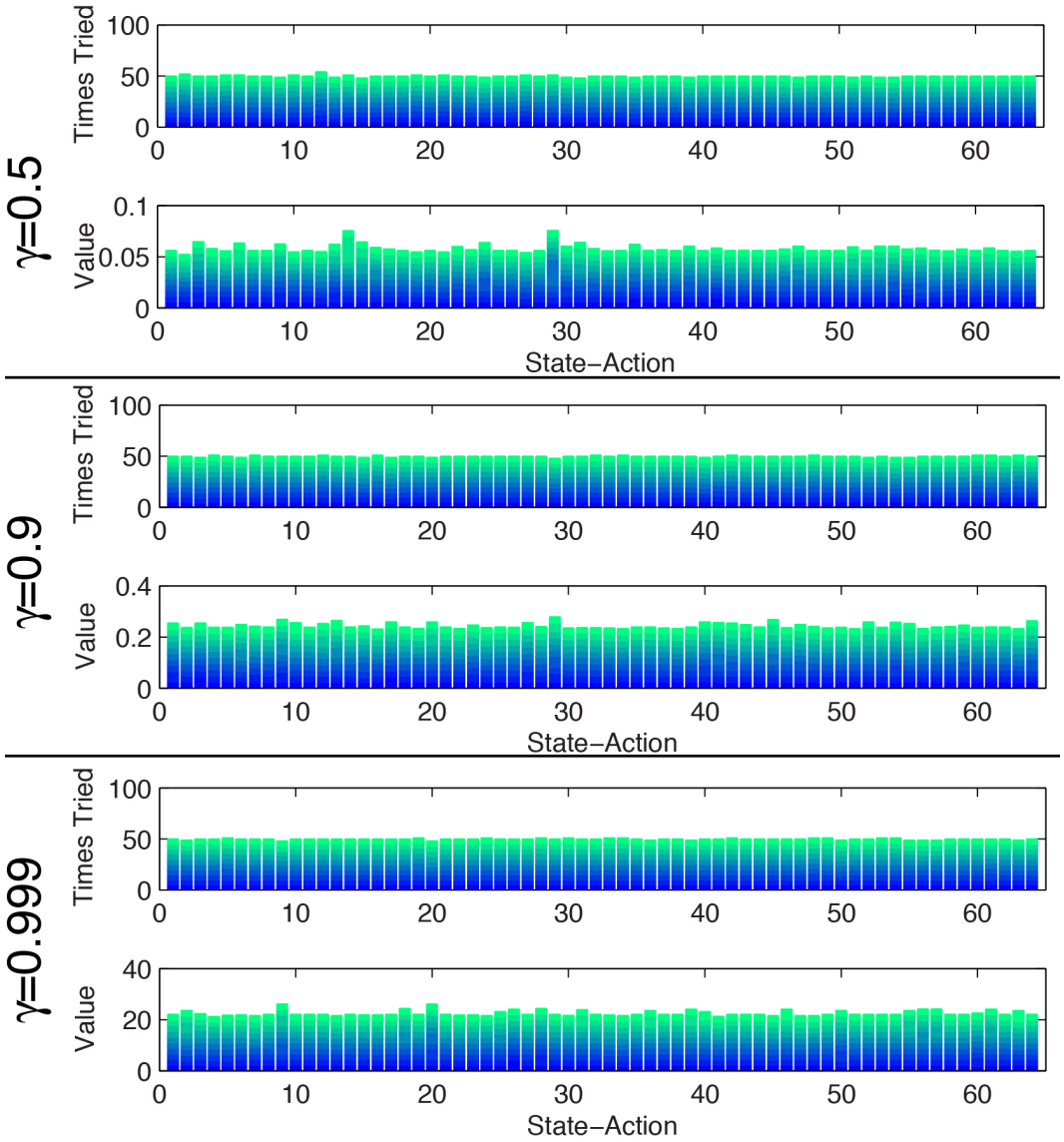


Figure 7.3. Stacked bar charts show the action selection history and value function at the final stage of hypercube exploration, after 320 actions have been executed. Each bar indicates a particular state-action, and each color represents data from an individual randomly initialized experiment. Data are presented for three different discount factors ($\gamma$). Now that exploration is almost complete, action selection is quite uniform, as is the value function, and value is roughly proportional to the discount factor.

Tabulated results of 10 runs are presented, showing the distribution of actions selected so far and the current value function after 50 actions have been tried (figure 7.2) and at the conclusion of the experiment after 320 actions have been tried (figure 7.3).

In the early stages of exploration, the curious RL agent's action selection behavior resembled sampling from a uniform distribution. This is intuitive, because the state-action space is highly symmetric (every state has 4 available state-actions and no state is further than 2 state-actions from any other state) and because all of the state-actions were equally interesting (they terminate deterministically in the intended goal state). As exploration progressed, the distribution of actions selected, which was initially quite noisy (figure 7.2), became increasingly uniform (figure 7.3). Moreover, the magnitude of the value function wass proportional to the discount factor, $\gamma$, as expected. The results of the hypercube exploration indicate that the curious RL agent operates as expected.

## 7.2 Planning Around Shoulder Constraints

The second experiment was in principal quite similar to the first, except the state-action space was made more complex. Instead of the hypercube of the previous experiment with its 16 vertices, the state space was a 4D hyper-lattice of rank 3, with 81 vertices at 25, 50, and 75 per cent of each joint's range of motion. Thus, the lattice had the same volume as the hybercube, and like the hypercube, it was centered within the robot's range of motion. Each vertex/state was connected to its nearest *equidistant* neighbors, such that at any given time, the agent could select from between 4 and 8 state-actions, depending on where it was in the hyper-lattice. The topology of the state-action space is shown in figure 7.4.

### 7.2.1 Efficiency of Exploration

The first experiment carried out in this larger state-action space[1] was intended to characterize the exploratory behavior of the agent employing artificial curiosity (AC), compared to two other agents using benchmark exploration strategies

[1]The rank 3 hyper lattice employed in this experiment has all the same states as the one described previously, however I naively/mistakenly connected each state to its 16 nearest neighbors rather than the set of equidistant nearest neighbors, so the total number state-actions was 1296 rather than 432 shown in figure 7.4.

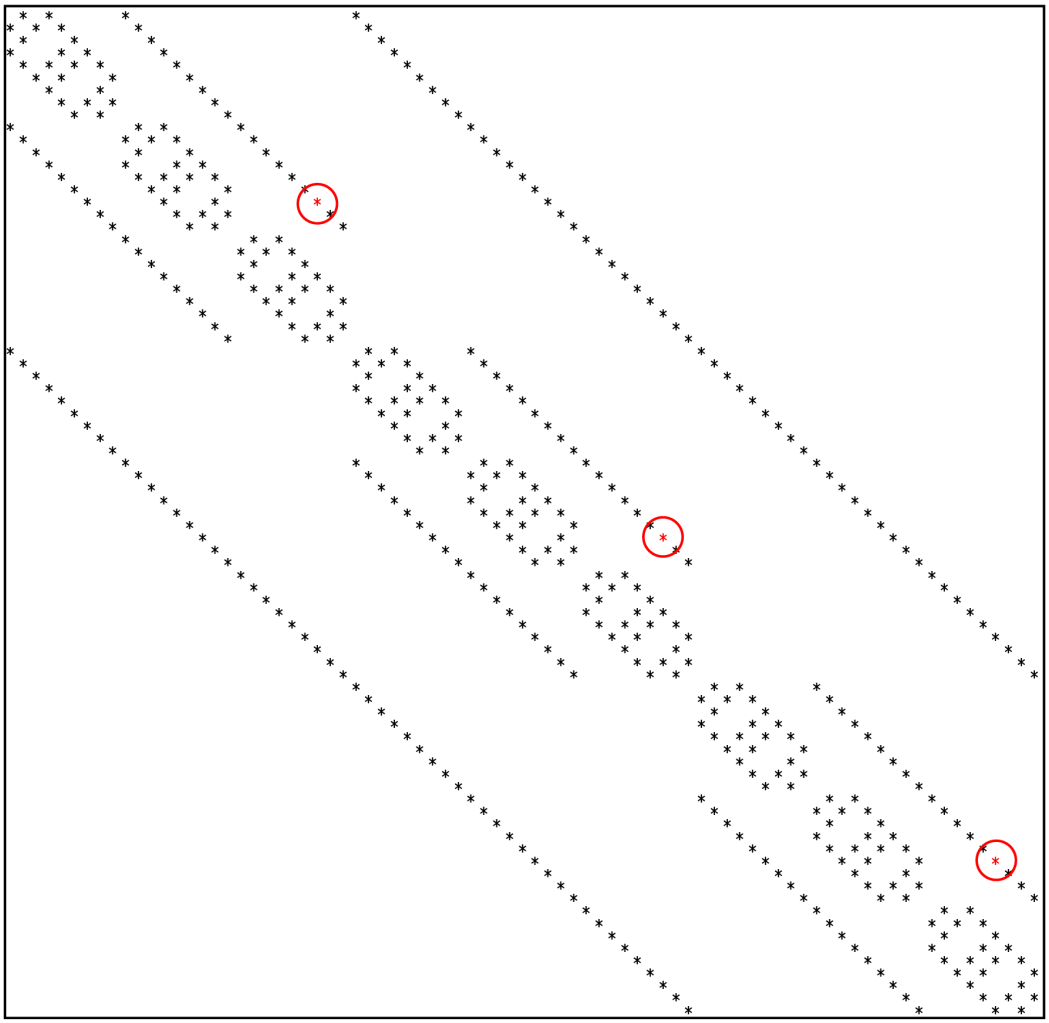

Figure 7.4. Rank 3 hyper-lattice for the shoulder constraints experiment. The connectivity is shown graphically where $*$ indicates that there exists an action, which is available to the agent in state *row* and intends to bring it to state *column*. Each state is connected to its nearest equidistant neighbors, and the number thereof varies between 4, at the corners of the lattice, and 8, at the center. The circled state-actions, shown in red, are interesting, as they do not necessarily terminate in the intended state, indicated by the column in which they appear.

from the RL literature. One explores randomly (RAND), and the other always selects the state-action least tried (LT)[2].

In comparing the AC agent with the RAND agent and the LT agent, it is claer that AC produces, by far, the best explorer (figure 7.5). In the early stages of learning, AC and LT try *only* novel actions, whereas RAND tries some actions repeatedly. Early on (before the agent has experienced about 220 state transitions), the only difference evident between AC and LT is that AC visits novel states more aggressively. This is intuitive upon reflection, as AC values states with *many* untried state-actions, and will traverse the state space to go find them, whereas LT has no global knowledge and just chooses the locally least tried state-action, regardless of where it leads. As learning continues, this key difference between AC and LT also begins to manifest in terms of the coverage of the action space. In fact, AC tries all possible state-actions in about $\frac{1}{2}$ the time it takes LT.

### 7.2.2 Action Distribution and Value Function

The next set of experiments investigated the distribution of actions selected by the curious agent as it explored the hyper lattice shown in figure 7.4. The measured result was the cumulative history of the agent's action selection and the value function. As in the first experiment, the workspace did not contain obstacles, the state of the robot was initialized randomly, and the agent explored the model until every state-action had been tried at least 5 times. The experiment was repeated many times over, varying not only the discount factor, $\gamma$, but also the initialization of the state transition probabilities and intrinsic reward. Tabulated results of 5 runs are presented, showing the distribution of actions selected throughout the experiment.

The first set of results (figure 7.6) shows the agent's behavior when the state transition probabilities and intrinsic reward are initialized as shown in table 6.1. In this case, for each action the agent believes that there is a 50% chance of success, which causes the agent to transition to the intended goal state, and a 50% chance of failure, which leaves the agent in the initial state. With this initialization, when the agent tries any state-action for the first time, success and failure produce identical intrinsic rewards. In other words, success and failure are equally interesting.

The second set of results (figure 7.7) shows the agent's behavior subject to

[2]If there are multiple least-tried state-actions (for example when none have been tried), the LT agent selects a random one from the least tried set.

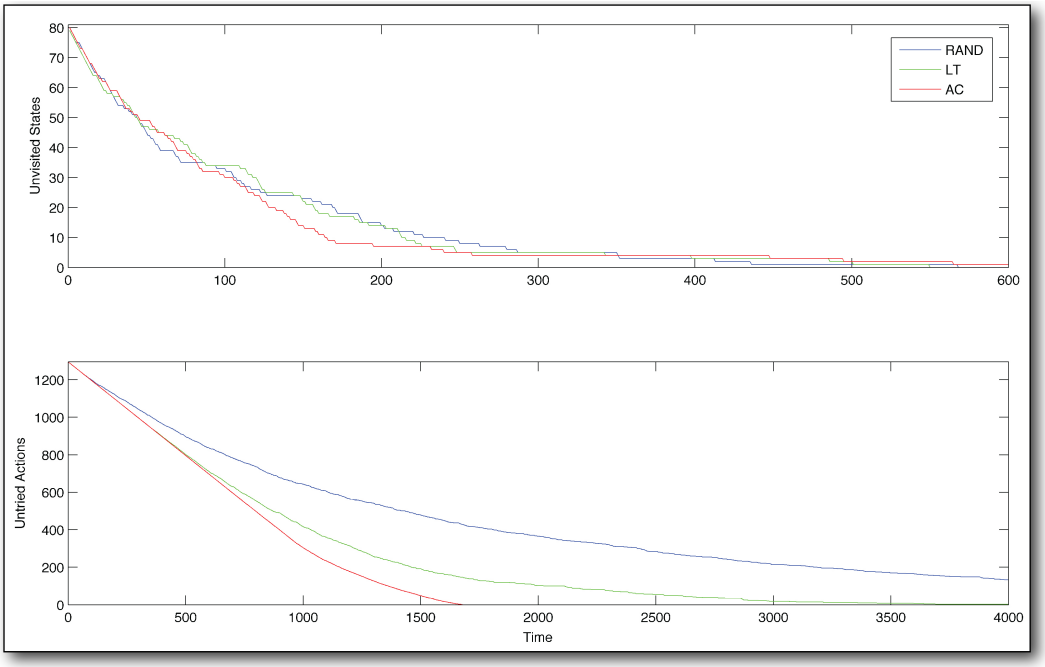


Figure 7.5. state-action space coverage during early learning: The policy based on Artificial Curiosity (AC) explores the state-action space most efficiently, compared to policies based on random exploration (RAND) and always selecting the least tried state-action (LT). Time is measured in state transitions.

a different initialization, which contains an additional observation of the goal state, as shown in table 7.1. This causes the agent to believe that actions are 60% likely to succeed and 40% likely to fail, making a failure more interesting (in terms of intrinsic reward) than a success.

All of the experimental results, comprising different initial states, different initializations of state transition probabilities and intrinsic rewards, and different discount factors, share the same general features.

### 7.2.3 State Space Coverage

The distributions of visits over the set of states is not uniform, but it has a distinct shape, which is repeated in each set of results. Qualitative analysis of the state space shows that the more often visited states are those, which have more neighbors, and therefore more actions which lead into them. These can be thought of as being near the middle of the state space.

| Observation | $T$ | $P$ | $R = D_{KL}(P\|\|T)$ |
|---|---|---|---|
| - | {} | {} | - |
| $s_i$ | {1} | {2} | 0 |
| $s_g(a)$ | {2,1} | {2,2} | 0.0589 |
| $s_g(a)$ | {2,2} | {2,3} | 0.0201 |

Table 7.1. 60-40 initialization of state transition probabilities

### 7.2.4 Uniformity of State-Action Selection

The distributions of tries over state-actions are fairly uniform. On the one hand this could be expected, extrapolating from the hypercube result, however on the other hand one might ask why the actions belonging to the more frequently visited states are not selected more frequently. The answer is of course that the more frequently visited states are so because they have more state-actions learning into them; analogously, they have more state-actions leading out, and that is why the distribution of state-action tries can remain so uniform even when some states are visited more frequently than others.

### 7.2.5 Anomalies in State-Action Distribution

There are some clearly defined features in the distributions of state-action tries, which break its uniformity. These spikes and dips, most of which clearly repeat themselves across different trials, are discussed here.

#### Hard to Reach States

One of the most obvious features is the set of three apparent holes at state-actions 122-125, 281-285, and 419-422, the state-actions available to the agent at states 24, 51, and 78, respectively, which are the least frequently visited states. The state-actions are not selected often, because the agent is not often in the appropriate initial state to do so. This is most obvious in the data from the 50-50 initialization with $\gamma = 0.5$ (figure 7.6 - top).

Another interesting feature is the spikes apparent at state-actions 107, 264, and 405, which have goal states 24, 51, and 78, respectively, precisely the hard-to-get-into states that have the high-valued state-actions. The agent is trying fairly hard to get into these less-frequently-visited states, but it still does not succeed very frequently compared to the rate at which it visits other states. The reason for that is because some other state-actions, which have the same goal

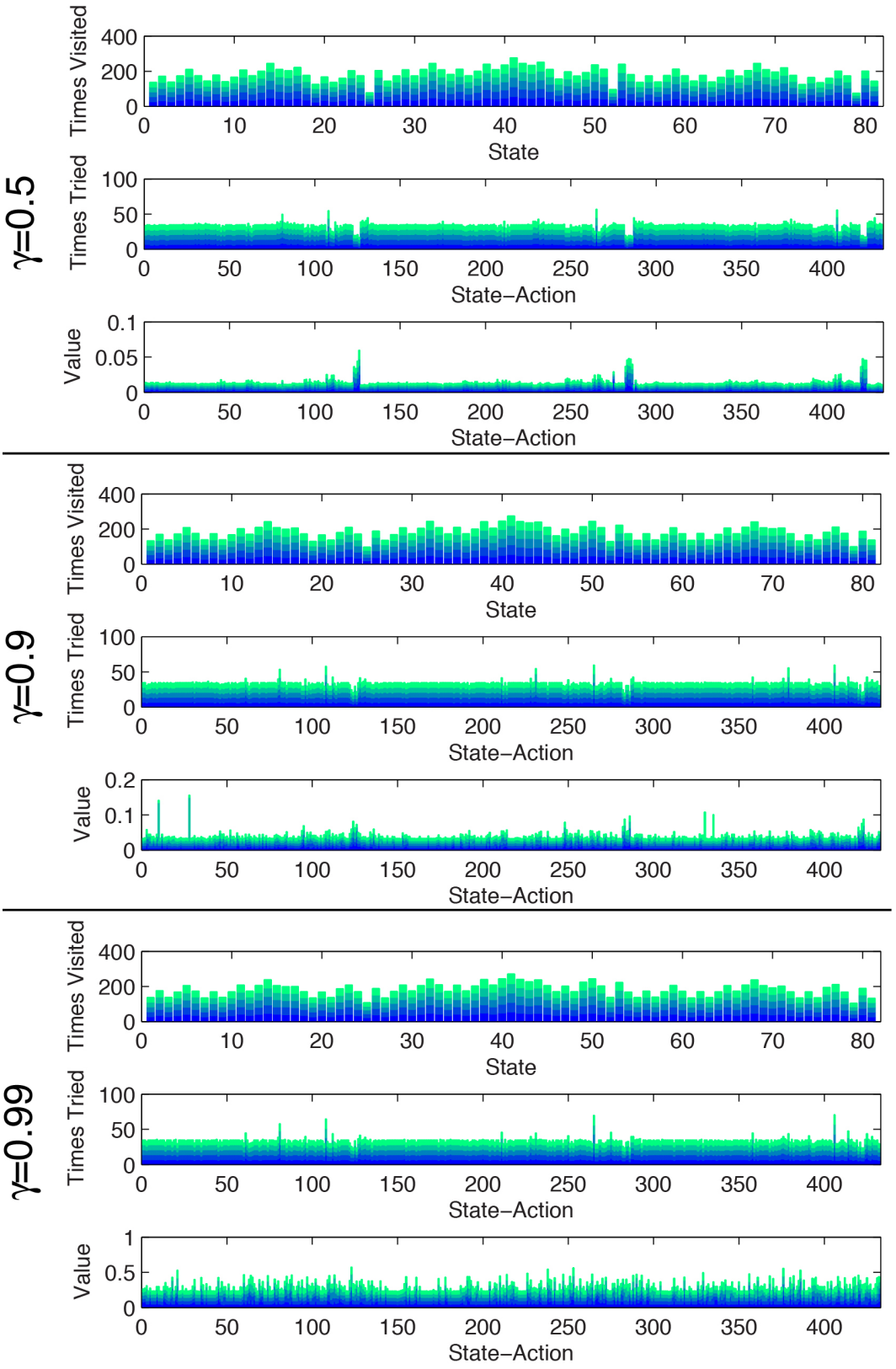


Figure 7.6. State-action history and value function for the rank-3 hyper-lattice (7.4) with 50-50 initialization, after 2987 exploratory actions. Each color represents data from an individual randomly initialized experiment.

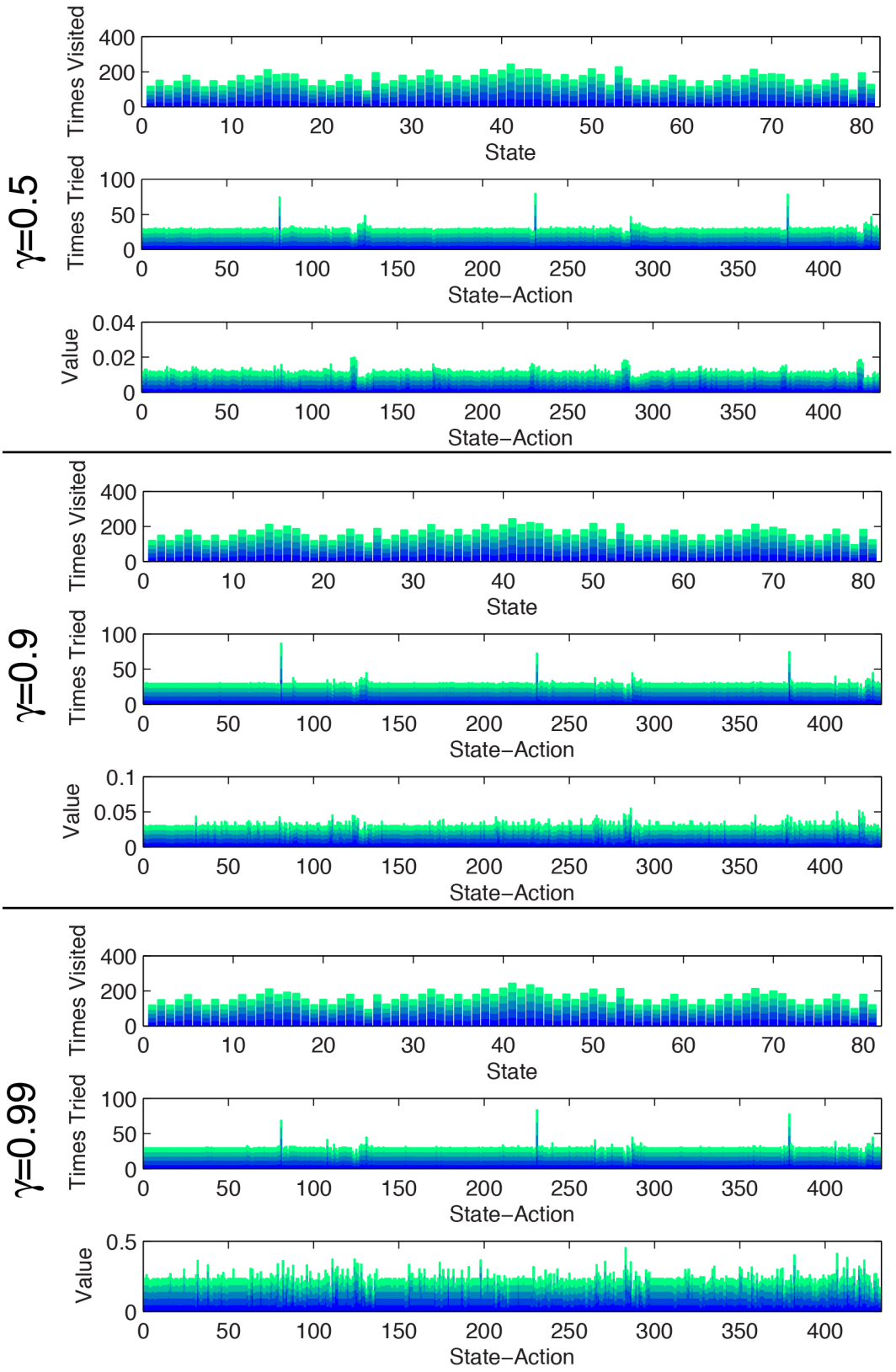


Figure 7.7. State-action history and value function for the rank-3 hyper-lattice (7.4) with 60-40 initialization (table 7.1) after 2651 actions. Each color represents data from an individual randomly initialized experiment.

states, are failing. Exactly which state actions are failing is not very obvious in the data from the 50-50 initialization, because failed actions and successful ones are equally interesting to the agent.

The state-action groups, 122-125, 281-285, and 419-422 each constitute a large bump in the value function. However, that value must be propagated through adjacent state-actions, in order to pull the agent into states 24, 51, and 78, such that the agent can then take one of the high valued actions. The holes in the distribution of state-action tries are in large part caused by the the low discount factor, $\gamma = 0.5$, which is responsible for the limited propagation of the high value of state-actions 122-125, 281-285, and 419-422, into the surrounding state-action space. In fact, as one looks down figure 7.6 at the data associated with $\gamma = 0.9$ and $\gamma = 0.99$, the holes become less and less pronounced as the spikes on actions 107, 264, and 405 grow taller.

### Failing State-Actions

Which state-actions fail to make states 24, 51, and 78 difficult for the agent to enter? The answer can easily bee seen in the data associated with the 60-40 initialization (figure 7.7), where the agent is more interested in failed actions than successful ones. Prominent peaks in the distribution of state-action tries are visible for all $\gamma$ at state-actions 80, 230, and 378. These are the infeasible actions, which are shown in red and circled in figure 7.4.

Qualitative analysis of the experiments showed that these failures occurred when the agent was repelled from the linear constraints, which are specified in the XML and protect the robot from cable length infeasibilities. A close inspection of the state transition probabilities associated with these actions shows that they do not fail deterministically. Instead they sometimes succeed and sometimes fail, and the ratio of success to failure varies from one run to the next, which is not surprising considering the relatively small number of tries belonging to each run.

The fact that static constraints do not necessarily lead to deterministic state transitions is quite interesting. It indicates a certain lack of precise repeatability of actions, and I speculate that this apparent stochasticity comes from some combination of the following:

- Discretization of continuous dynamics
- Noise in the sensory-motor apparatus
- Complex robot dynamics not modeled in the planning/control system

- Residual momentum in the system when the robot's position is considered to have settled

Due to this lack of repeatability, a *plan first, act later* approach, such as PRM planning, will never work robustly in practice. Plans will sometimes fail at runtime, and not necessarily in a repeatable manner. In fact the lighter and more flexible robots get, and the more they are controlled by complex computer networks, the more pronounced these problems will become, which is an important motivation for continuing to develop more robust solutions, such as the MDP motion planning presented here.

### 7.2.6 Spurious Peaks

Lastly, there is one feature in the experimental data, which is not part of a trend that spans different experiments or experimental trials. In the value function for the 50-50 data, with $\gamma = 0.9$ (figure 7.6, middle), there are spurious peaks at state-actions 9, 27, 334, and 329. These state-actions have an artificially high value, apparently for no reason, and almost all of the (cumulative) value is from only one trial. The spikes at 9 and 27 are generated by data from the fourth trial (medium green), and those at 334 and 329 come from the fifth (light green).

Extensive qualitative analysis of the trials in question revealed that the affected state-actions were not particularly interesting until the end of the experiment, when suddenly, they became very interesting, and their value shot up accordingly. This occurred because the actions, which had always terminated deterministically in the expected state, suddenly brought the robot somewhere else. These particular experiments were done using the iCub simulator, which made it safer and easier to run batches of experiments over days and nights, and the observed effect therefore arises from numerical instabilities within the Open Dynamics Engine (ODE). However similar weirdness (for lack of a better word) has also been observed in experiments with the real hardware, and the reason for that remains somewhat mysterious.

Whatever the source of the sporadic, strange behavior of the robot and/or simulator, the curious agent responds exactly as an adaptive control system should. It becomes interested in the affected state-actions and tries them out a few times in an effort to improve its model of the robot.

## 7.3 Planning Around Self Collisions and Shoulder Constraints

Having verified that the curious agent responds sensibly to state-actions that do not work as expected, I began to experiment with larger state-action spaces. Here, I present two rank 4 hyper-lattices, with 256 states and 1536 state-actions. One is smaller than the other, with state-centers at permutations of 20, 40, 60, and 80 per cent of each joint's range of motion (figure 7.8). The geometrically larger hyper-lattice is stretched out to cover a larger volume of the configuration space, with state-centers at permutations of 12.5, 37.5, 62.5, and 87.5 per cent (figure 7.9).

A qualitative analysis of the exploration of both rank 4 hyper-lattices revealed that many self collisions occur between the iCub's arm and body. Thanks to the dynamic collision avoidance provided by MoBeE, some of these collisions do not affect the outcome of the relevant state-actions, because the robot's trajectory is deflected in a way that it still can reach the goal state. However other state-actions do fail to bring the robot to the desired goal state, and some states turn out to be unreachable, as they lie entirely in infeasible regions of configuration space.

The rank 4 lattices have significantly more states than do the previous rank 3 ones, and therefore they take much longer to explore. For this reason, and also because the previous experiments verified that the learning system was behaving sensibly, I did not run batches of redundant experiments, nor did I vary the transition probability initialization or $\gamma$. Instead I used the 50-50 initialization with $\gamma = 0.9$, with the intention of learning practical MDP planners to apply in reaching experiments. Following are some observations concerning the resulting data.

For the smaller hyper lattice (figure 7.8), the unreachable states are 60, 124, 188, 189, and 252. These result in blocks of state-actions, which maintain high values throughout the experiment, 334-337, 746-750, 1162-1166, 1167-1172, and 1518-1521 respectively. Since they are never tried, their rewards never decay. As the experiment runs, the agent explores the rank 4 hyper-lattice in a mostly uniform manner, but spends considerable time trying to get into the hard-to-reach (or impossible to reach) states where there are high-valued untried actions.

Also of interest are states 61, 125, and 253, which were visited very infrequently compared to the other states, 2, 3, and 2 times respectively. Consequently, many of the state-actions there too remain untried, despite many

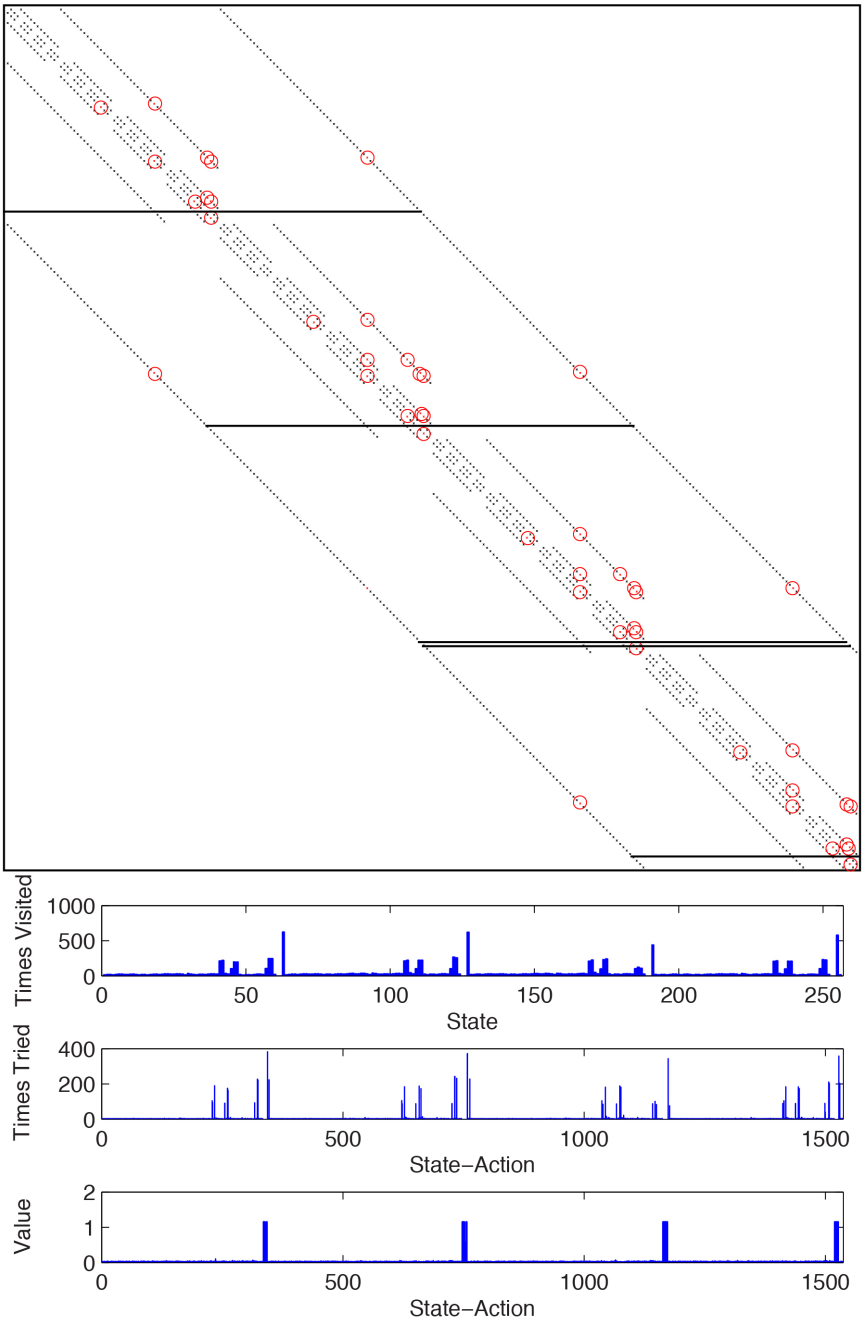


Figure 7.8. Rank 4 hyper-lattice with 256 states, centered at 20, 40, 60, and 80 per cent of each joint's range of motion. The connectivity is shown graphically (top) where $*$ indicates that there exists an action, which is available to the agent in state *row* and intends to bring it to state *column*. Each state is connected to its nearest equidistant neighbors. The circled state-actions, shown in red, are interesting, as they do not necessarily terminate in the intended state. Rows of state-actions, which have been stricken through with horizontal lines, belong to unreachable states and have therefore not been tried. The distributions of states visited and actions selected are shown (bottom) along with the value function, after the agent has taken 14597 actions. The 50-50 initialization was used with $\gamma = 0.9$.

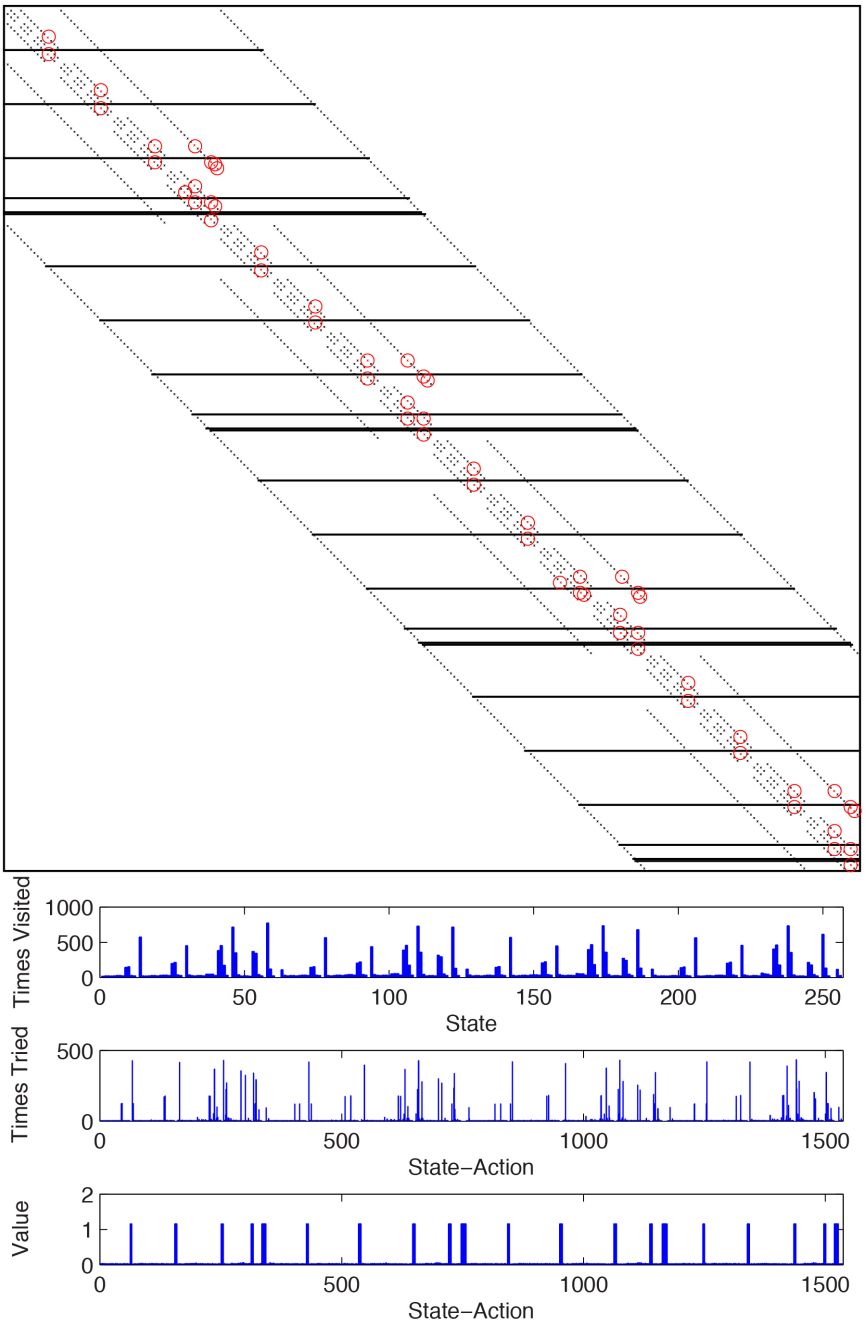


Figure 7.9. Rank 4 hyper-lattice with 256 states, centered at 12.5, 37.5, 62.5, and 87.5 per cent of each joint's range of motion. The connectivity is shown graphically (top) where $*$ indicates that there exists an action, which is available to the agent in state *row* and intends to bring it to state *column*. Each state is connected to its nearest equidistant neighbors. The circled state-actions, shown in red, are interesting, as they do not necessarily terminate in the intended state. Rows of state-actions, which have been stricken through with horizontal lines, belong to unreachable states and have therefore not been tried. The distributions of states visited and actions selected are shown (bottom) along with the value function, after the agent has taken 26963 actions. The 50-50 initialization was used with $\gamma = 0.9$.

hundreds of tries to enter these states. For example:

- State-action 322, which was intended to take the agent from state 57 to state 61 and was tried 228 times, actually terminated in state 57 once, in state 61 twice, and in state 62 225 times.
- State-action 731, which was intended to take the agent from state 121 to state 125 and was tried 244 times, actually terminated in state 121 3 times, in state 125 3 times, and in state 126 238 times.
- State-action 1506, which was intended to take the agent from state 249 to state 253 and was tried 213 times, actually terminated in state 249 2 times, in state 253 2 times, in state 254 208 times, and in state 238 once.

This is further evidence that the discretized dynamical systems within MoBeE, which protect the robot from harm and facilitate real-time exploratory behavior, do not produce deterministic state transitions, even though the geometric constraints on the robot are static. It therefore demonstrates the utility of the MDP based approach to planning presented here.

The experiment with the geometrically larger hyper-lattice caused the robot to get closer to its joint limits, and therefore many more arm/body self collisions were produced. The experiment proceeded analogously to the smaller one, however there were many more unreachable states. In fact they were far too many to discuss individually, but the data are tabulated in figure 7.9. A qualitative analysis of both experiments lead me to believe that these rank 4 hyper-lattices would be of sufficient complexity to facilitate coarse planning for reaching experiments.

## 7.4 Discovering the Table with a Multi-Agent RL System

In the second experiment, both of the iCub's arms and its torso are controlled, 12 DOF in total. A hypercube in 12 dimensions has 4096 vertices, and a rank 3 hyper-lattice has 531,441 vertices. Clearly, uniform sampling in 12 dimensions will not yield a feasible RL problem. Therefore, the problem must be parallelized. Three curious agents are employed, controlling each arm and the torso separately, not having access to one another's state. The state-action spaces for the arms are the rank 3 hyper-lattice described in section 7.2, with each state being connected to its 16 nearest neighbors. The state-action space for the 3D

torso is defined in an analogous manner (25%, 50%, and 75% of each joint's range of motion), resulting in a 3D lattice with 27 vertices, which are connected to their $2^3 = 8$ nearest neighbors, yielding $27 \cdot 8 = 216$ state-actions.

The iCub is placed in front of a work table, and all 3 learners begin exploring (figure 7.10). The three agents operate strictly in parallel, each having no access to any state information from the others, however they *are* loosely coupled through their effects on the robot. For example, the operational space position of the hand (and therefore whether or not it is colliding with the table) depends not only on the positions of the joints in the arm, but also on the positions of the joints in the torso. Thus, we have three interacting POMDPs, each of which has access to a different piece of the complete robot state, and the most *interesting* parts of the state-action spaces are where the state of one POMDP affects some state transition(s) of another.

When the torso is upright, each arm can reach all of the states in its state space, but when the iCub is bent over at the waist, the shoulders are much closer to the table, and some of the arms' state-actions become infeasible, because the robot's hands hit the table. Such interactions between the learners produce state-transition distributions, like the one shown in figure 7.11, which are much richer than those from the previous experiments. These state-actions are the most interesting because they generate the most slowly decaying intrinsic reward of the type shown in table 6.4. The result is that the arms learn to avoid constraints as in the first experiment, but over time, another behavior emerges. The iCub becomes interested in the table, and begins to touch it frequently. Throughout the learning process, it spends periods of time exploring, investigating its static arm constraints, and touching the table, in a cyclic manner, as all the intrinsic rewards decay over time.

In figure 7.12, the distribution of tries over the state-action space is tabulated for each of the three learners after 18,000 state transitions, or a little more than two full days of learning. As in the previous experiment, we see that the curious agent prefers certain state-actions, selecting them often. Observing the behavior of the robot during the learning process, it is clear that these frequently chosen state-actions correspond to putting the arm down low, and leaning forward, which result in the iCub's hand interacting with the table. Furthermore, the distribution of selected state-actions for the right arm and the left arm are very similar. This is to be expected, since the arms are mechanically identical and their configuration spaces have been discretized the same way. It is an encouraging result, which seems to indicate that the variation in the number of times different state-actions are selected does indeed capture the extent to which those state-actions interfere with (or *are* interfered with by) the other learners.

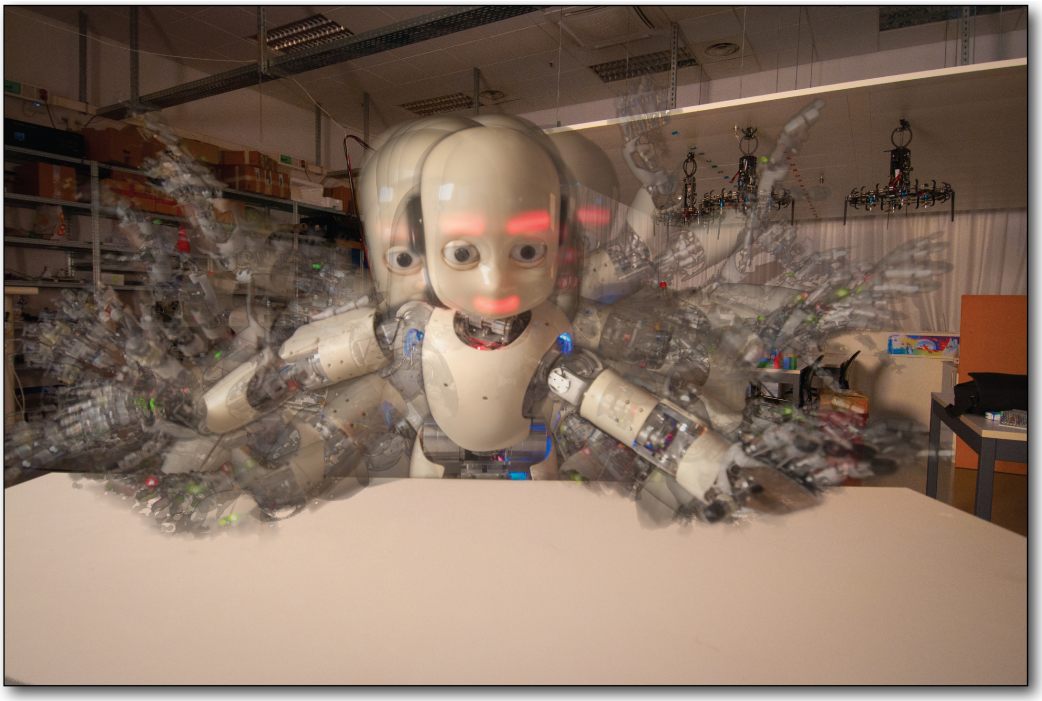

Figure 7.10. Autonomous exploration: This composite consists of images taken every 30 seconds or so over the first hour of the experiment described in section 7.4.1. Although learning has just begun, we already begin to see that the cloud of robot poses is densest (most opaque) near the table. Note that the compositing technique as well as the wide angle lens used here create the illusion that the hands and arms are farther from the table than they really are. In fact, the low arm poses put the hand or the elbow within 2cm of the table.

The emergence of the table exploration behavior is quite promising with respect to the goal of using MDP based motion planning to control an entire humanoid *intelligently*. An intractable configuration space was partitioned into several loosely coupled RL problems, and with only intrinsic rewards to guide their exploration, the learning modules coordinated their behavior, causing the iCub to explore the surface of the work table. Although the state spaces were generated using a coarse uniform sampling, and the object being explored was large and quite simple, the experiment nevertheless demonstrates that MDP motion planning with artificial curiosity can empower a humanoid robot with many DOF to explore its environment in a structured way and build useful, reusable models.

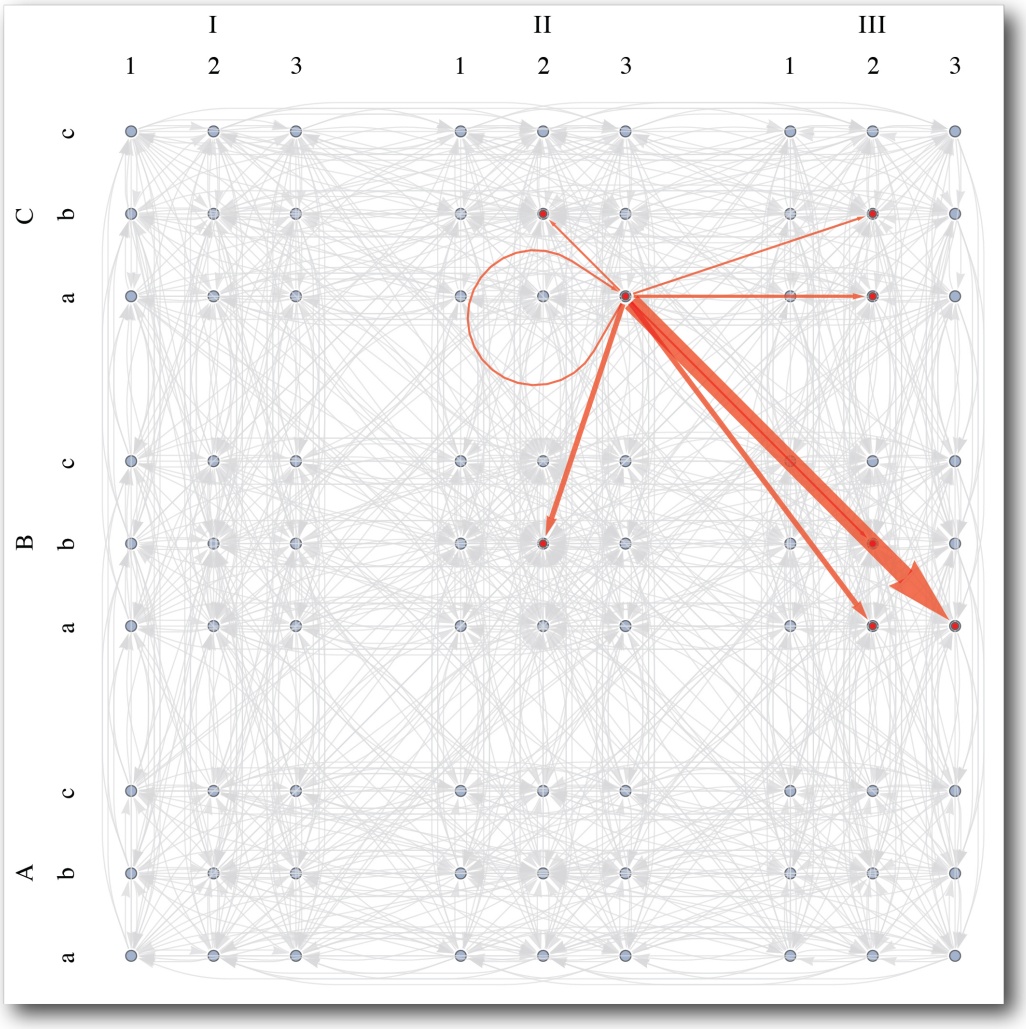


Figure 7.11. State space and transition distribution for an interesting arm action in multi-agent system: The 4D state space is labeled as follows: shoulder flexion/extension (1,2,3), arm abduction/adduction (a,b,c), lateral/medial arm rotation (I,II,III), elbow flexion/extension (A,B,C). The red arrows show the distribution of next states resultant of an interesting state-action, which causes the hand to interact with the table. Each arrow represents a state transition probability and the weight of the arrow is proportional to the magnitude of that probability. Arrows in gray represent boring state-actions. These work as expected, reliably taking the agent to to the intended goal state, to which they point.

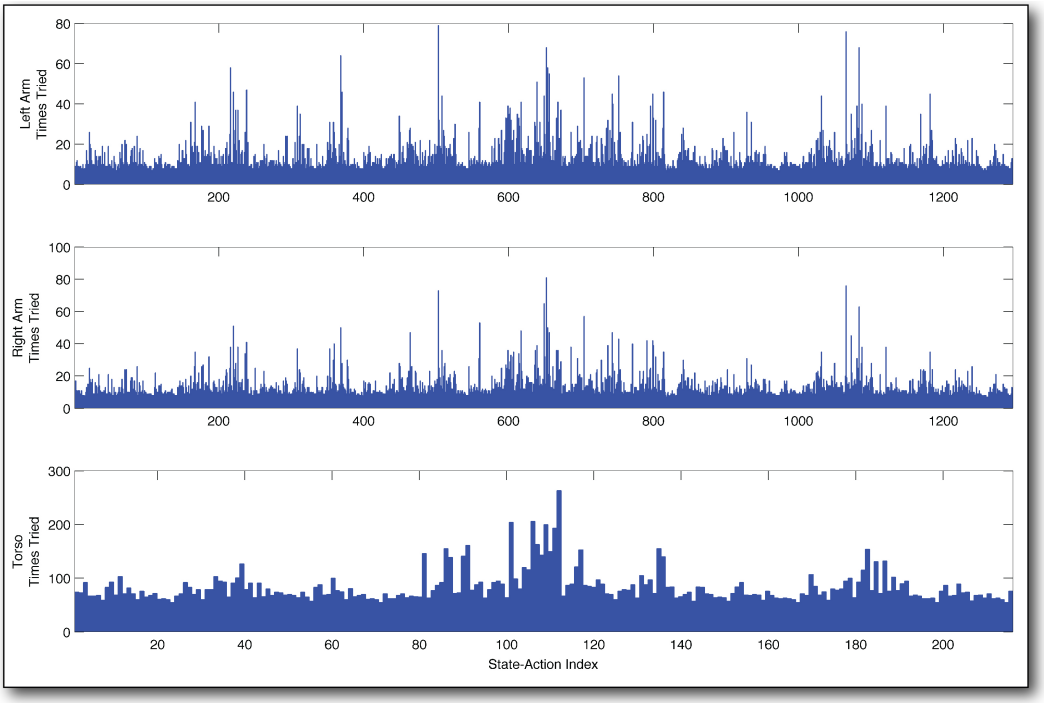


Figure 7.12. Frequency of actions taken by three curious agents in parallel: The most interesting actions are selected much more often than the others. They correspond to moving the arm down and leaning the torso forward. This results in the iCub robot being interested in the table surface. Note the similarity in the behavior of the two arms.

### 7.4.1 Planning in a Dynamic Environment

There is an alternative way to view the multi-agent experiment. Because the arm does not have access to the torso's state, the experiment is exactly analogous to one in which the arm is the only learner and the table is a dynamic obstacle, moving about as the arm learns. Even from this alternative viewpoint, it is none the less true that some actions will have different outcomes, depending on the table configuration, and will result in state transition distributions like the one shown in figure 7.11. The key thing to observe here is that while the curious agent is interested in interacting with the dynamic obstacles, if one were to turn it off and exploit the planner by placing an external reward at some goal, removing the intrinsic rewards, and recomputing the value function, then the resulting policy/plan will try to avoid the unpredictable regions of the state-action space, where state transition probabilities are relatively low. In other

words, training an MDP planner in an environment with dynamic obstacles, produces policies that plan around regions where there tend to be obstacles.

# Chapter 8

# Learning To Reach

In the model learning experiments of the previous chapter, the implementation of actions was designed to facilitate motion planning. The actions simply set an attractor in configuration space via the MoBeE framework at the Voronoi center of a region of configuration space, which defines a state. The robot then moved toward the attractor according to the transient response of the dynamical system within MoBeE. The result was that the MDP planner exploited a kind of best effort position control with reactive constraint avoidance to function as an enhanced version of a PRM planner. However, the RL framework is in principal capable of much more.

In addition to position control, the MoBeE framework supports force control in both joint space and operational space, and as far as the RL implementation is concerned, actions can contain arbitrary control code. Therefore, a curious agent for the iCub can benefit from different action modalities, implemented through MoBeE's dynamical system. My work is the first such application of a multi-modal action repertoire to RL on a humanoid robot that I am aware of.

The power of the reach actions is that they can access arbitrary real-valued points in workspace. They therefore compliment the MDP planner, allowing it to focus on on coarse planning, bringing the robot to a suitable pre-reach pose. Once this is done, a reaching state-action can be be executed to close the remaining error distance to real-valued targets in workspace. Thus the MDP reach planner can contain relatively few states, but it can still reach an infinite number of target points in the workspace.

Before the addition of the reach actions, the MDP planner worked as a PRM planner does, in the sense that one would provide a goal state by putting a reward there, and the planner would work out how best to get the reward by computing the value function. The reach actions change the task somewhat. The

input to the planner should now be a reach target, a real valued workspace point in or on some geometric primitive, and the RL system should work out which is the best reaching state-action for the job, put a reward there, and plan a path by computing the value function. To execute the reach, the agent should first go to a good pre-reach state by selecting appropriate 'position move' actions. Only when it is in such a state, which offers favorable initial conditions to the reach action, should it actually execute the reach[1].

Perhaps the simplest way to formalize the above is to view each novel reach target as an individual RL problem. The agent must explore the state space and try reaching actions to learn the reward matrix for each problem. It should store a history of actions taken and rewards received for each problem it sees, such that these reward matrices can be recalled, and the learner can become more and more competent at solving reaching problems.

## 8.1 Planning Around Arbitrary Obstacles

The MDP was learned to represent the ever-present features of the robot's configuration space. Thus, it allows motion planning, which is respectful of self-collisions, but the addition of obstacles[2] in the workspace changes the topology of the configuration space. It introduces infeasible regions, holes, around which the agent must plan, but the relevant information is not in the MDP.

If nothing is done to represent these topological changes to the configuration space, the motion planning fails in a scenario like the following: The agent finds itself at some initial state, $s_i$, and chooses an action, $a_{ig}$, according to the greedy policy, which according to the model, has a high probability of taking the agent to its goal state $s_g$. However $a_{ig}$ tries to move the arm/hand through the target object, which was not present when the MDP was learned, and this is obviously not allowed by MoBeE. Thus $a_{ig}$ terminates (with high probability) with the agent still in $s_i$. At this point the agent again selects $a_{ig}$ according to the greedy policy and is thus stuck in a cyclic behavior, which prevents it from effectively climbing the gradient of the value function.

Topological changes to the configuration space resultant of workspace targets/obstacles must be represented. Once could consider building the miss-

[1] A particularly appealing aspect of this approach to reaching is that it explicitly avoids inverse kinematics.

[2] In order to reach to a target object, the agent should not touch it, except with its palm and/or fingertips at the very end of the motion. Therefore reach targets can actually be seen as obstacles until the reach action completes, and an eventual grasping action takes over to actually interact with the object.

ing topological information directly into the MDP, which would entail learning how each potential target/obstacle affects the state transition probabilities for each action. However, such an approach would essentially mean that each RL problem requires its own Markov model, and that is problematic since there is a potentially infinite number of problem instances (reach target locations in workspace).

Luckily, the kinematic mapping is continuous. Therefore, if some target/obstacle at a workspace point, $x^*$, affects a particular region of the configuration space, then an infinitesimal change in $x^*$ produces an infinitesimal change in the geometry of the affected region of configuration space. This encourages the belief that for a *novel* reach target, $x^*$, a reasonably good estimate of the set of affected state transitions can be constructed by using the sets of state transitions affected by reach targets near $x^*$.

The effects of target/obstacle geometries on state transition probabilities are not all equivalent. Sometimes being pushed laterally into an adjacent state does not affect the agent's ability to climb the value function. Other times it does, as in the above example, causing the agent to cyclicly repeat a sequence of actions. In either case, it is not clear that learning the probabilities of these occurrences is of primary importance to reach learning. Instead, robustly moving the robot to try the high-valued reaching state-action is paramount.

In light of this realization, one could simply use negative rewards to discourage the agent from making potentially problematic state transitions. This way, the the MDP would represent the robot itself, and all the information relevant to the RL problems, namely which reaching state-actions are best and which state transitions are to be avoided, would be contained within problem specific reward matrices.

The mechanism at work in the experiments presented here is as follows: During reach learning, the MDP planner is exploited to plan paths to sensible pre-reach states. The state transition probabilities are not updated. Positive rewards are associated with the reaching state-actions (these will be covered in section 8.2), while 'successful' state transitions are generally not rewarded. Thus the agents plans to move through state space, ascending the value function, in order to find a good state from which to initiate a reaching action. 'Failure' of a state transition is defined by a very simple heuristic, which is surprisingly powerful. Any state action $(s, a, s')$, for which $s'$ does not have a higher value than $s$, is considered failed, and a large negative reward (-10 in the following experiments) is associated with $(s, a)$ in the current problem's reward matrix.

As the agent tries to reach different pre-reach states, each state-action develops a history, of length $n$, which lists the previously tried reach targets, $x_i^*$, and

the corresponding (negative) rewards gotten, $r_i(x_i^*)$. From this information, the expected reward can be estimated for each novel goal point, $x_{n+1}^*$, according to:

$$e_{n+1} = e(x_{n+1}^*) = \frac{\sum_{i=1}^{n} \frac{1}{1+|x_{n+1}^* - x_i^*|} r_i}{\sum_{i=1}^{n} \frac{1}{1+|x_{n+1}^* - x_i^*|}} \tag{8.1}$$

The estimate is computed prior to each execution of each state-action, and it too is appended to the history of the relevant state-action, $H_{s,a}$, which comprises the tuples:

$$H_{s,a} = \{\{x_1, e_1, r_1\}, \{x_2, e_2, r_2\} \ldots \{x_n, e_n, r_n\}\} \tag{8.2}$$

Notice that when the history is empty, ($n = 0$) the estimated reward is undefined:

$$e_1 = \frac{\sum \varnothing}{\sum \varnothing} = \frac{0}{0} \tag{8.3}$$

Therefore, $e_1$ must be defined explicitly: $e_1 = 0$. This optimistic initialization means that state transitions are expected to work, receiving zero reward.

Notice also, that once a state transition has failed, the responsible state-action always estimates a negative reward, though it is small, provided that the current reach target, $x_{n+1}^*$, is far from those that generated negative rewards in the past. Therefore, the agent prefers paths that are always collision free, which maximizes the robustness of the reach planner.

## 8.2 Incorporating Dynamic Reaches

In the experiments presented here, reaches are implemented in a manner very similar to algorithm 5. The only difference is that rather than generating a trajectory offline, they force MoBeE's dynamical system in realtime.

Algorithm 5 was written for clarity and notational precision, and it computes a piecewise linear, positional trajectory. This is done offline, using only a forward kinematic model. At each time step, the desired $\Delta\mathbf{x}$ is used to compute the $\Delta\mathbf{q}$, which is applied directly to update the position of the robot.

Contrastingly, the reach action used here gets the actual position of the robot from the hardware/simulator at each time step. The desired $\Delta\mathbf{x}$ is still used to

compute the $\Delta\mathbf{q}$, but $\Delta\mathbf{q}$ is interpreted as a *direction* and used to force MoBeE's dynamical system away from the attractor, $q^*$, of the currently active state.

These dynamic reaches seek to put a marker (section 2.3.1), with workspace position $x$, on a target point, or 'workspace attractor,' $x^*$. The reaching begins as $x$, is forced toward $x^*$. When the robot stops moving, or a timeout occurs, the reward is computed as a function of the residual error according to equation 8.4. Then the forcing stops, and MoBeE's attractor dynamics pulls the robot pose back toward the center of the active state, $q^*$ in equation 5.7. Finally, when the robot pose settles (for the second time) the action terminates.

$$r = r(x^*) = \frac{1}{1 + |x - x^*|} \tag{8.4}$$

As the agent tries to reach different goal points, each reaching state-action develops a history, as described above for state transition actions. Also similarly to state transition actions, rewards are predicted by reaching state-actions, however the computation differs slightly from equation 8.1.

For a state transition action, $e_1 = 0$, is optimistic. No news is good news from the path planner. Reach actions, however, generate positive rewards, and an optimistic initialization must therefore be positive. In fact $e_1 = 1$ would do nicely as equation 8.4 shows that the actual rewards generated satisfy $r \in (0, 1]$.

Even with an optimistic initialization however, the reward estimator in equation 8.1 suffers from an unfortunate problem. If some novel reach target, $x_{n+1}$ is very far from all the others tried, then the prediction will be low. Even if the action does very well on a few problems, the reward predictor is always pessimistic with regard to far away problems. This is likely a bad thing. In fact, since the learner is dealing with sets of many different RL problems, it would be nice to have a predictor that is not only temporally optimistic (when it has no experience), but also spatially optimistic (when asked about a problem very different from those with which it has experience). Therefore, reaching state-actions employ the following alternative predictor:

$$e_{n+1} = e(x^*_{n+1}) = \frac{1 + \sum_{i=1}^{n} \frac{1}{1+|x^*_{n+1} - x^*_i|} r_i}{1 + \sum_{i=1}^{n} \frac{1}{1+|x^*_{n+1} - x^*_i|}} \tag{8.5}$$

Notice the optimism of this expression. When the agent has no experience at all, $e_1 = 1$. When the agent has experience very far from the goal point, $x^*_{n+1}$, the sums make a small contribution and $e_{n+1} \approx 1$ (though $e_{n+1} < 1$). Only when the agent has a certain amount of experience near $x^*_{n+1}$ do the sums dominate

the expression and provide a realistic (as opposed to optimistic) reward estimate. The estimated reward can be used to generate policies to reach to novel workspace goals/targets, and the prediction error associated with the estimate (and/or its derivatives) can be used to define an intrinsic reward. In the following experiments, I use a very naive intrinsic reward, which is simply equivalent to the prediction error:

$$r_{intrinsic} = e(x^*) - r(x^*) \tag{8.6}$$

Here, $e_i$ is the reward predicted by equation 8.5 for some state-action reaching to $x_i^*$, and $r_i$ is the actual reward computed according to equation 8.4. Finally, since both reward predictions and rewards themselves are written to the history, intrinsic reward can be predicted straightforwardly, according to:

$$e_{intrinsic,n+1} = e(x_{n+1}^*) = \frac{1 + \sum_{i=1}^{n} \frac{1}{1+|x_{n+1}^* - x_i^*|} r_{intrinsic,i}}{1 + \sum_{i=1}^{n} \frac{1}{1+|x_{n+1}^* - x_i^*|}} \tag{8.7}$$

The reach learning approach described here is incremental and suitable for online applications. One can feed the system novel reach targets, which constitute new RL problems, and it can predict rewards for each state-action to generate a policy that reflects the best sequence of actions to take given the currently available knowledge of the task. The *best* thing to do can be measured in terms of exploration (equation 8.7) or exploitation (equation 8.5), and as the system gains experience, so improves the accuracy of its predictions.

## 8.3 The Reach Learning Task

A prerequisite to the learning discussed here is a model (an MDP) of the robot's configuration space, as described in chapter 6 and demonstrated in chapter 7. In these experiments, I have used the rank 3 hyper-lattice of section 7.2, which offers a reasonably wide variety of pre-reach poses, yet is significantly smaller than the rank 4 hyper-lattice. The smaller size of the rank 3 hyper-lattice facilitate quicker experiments, and therefore I have been able to run more of them, which is important due to the stochastic nature of the reach planner.

The MDP motion planner is enhanced by the addition of a dynamic reaching action, which is based on operational space control via MoBeE (8.2) and can be called from any state in the model. The task of the reach learning agent is to

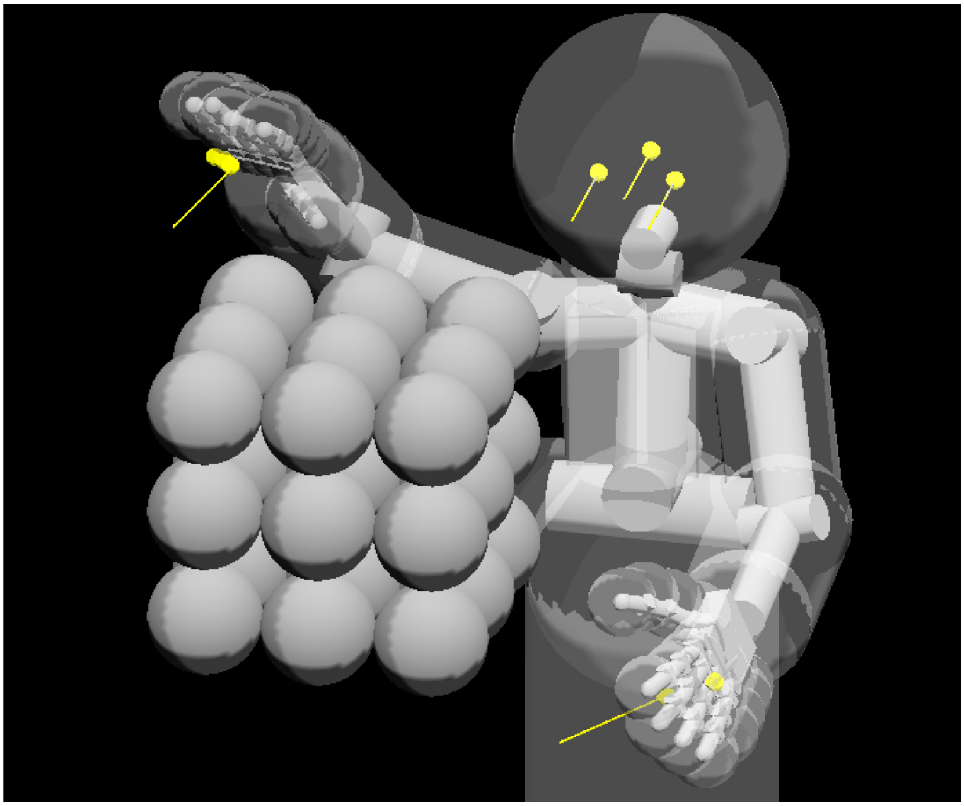

Figure 8.1. The set of 27 reach target objects (RL problems), arranged in a grid, which roughly covers the workspace of the iCub's right arm, without the aid of torso motion.

move around the state-action space and reach to workspace targets in order to discover which initial conditions (pre-reach poses) are most suitable for which targets.

In order to compare different exploration strategies, I have chosen a set of RL problems comprising 27 reach targets, arranged in a grid, which roughly covers the workspace of the iCub's right arm, without the aid of torso motion. Each reach target lies at the center of a sphere, 10cm in diameter, and the set of these reach targets is shown in figure 8.1.

### 8.3.1 The Problem-Try

To describe the reach learning task, I must first introduce the notion of *problem-tries*, which are essentially runs of policies on a problem. For each reach target (RL problem), the learner generates a policy consisting of a sequence of state

transition actions, which generate rewards, $r \leq 0$, followed by a reach, which generates a positive reward. Once that positive reward has been gotten by the learner, the policy is considered complete, and the *try counter* for the relevant problem is incremented. Thus, a problem-try is the execution of a policy that culminates in a reach toward a target, which defined the RL problem at hand.

### 8.3.2 Exploration

With the notion of the problem-try so defined, the reach learning proceeds according to the following steps:

1. The least tried RL problem is selected and inserted into the MoBeE model. If multiple problems have the same number of tries, as is the case initially, a random problem is selected from the set of equally tried problems.

2. A reward matrix is estimated for the chosen problem. State transition actions always estimate rewards according to equation 8.1. Reach rewards are estimated in three different ways in order to create different exploration strategies.

   - A random exploration policy estimates a reward of 1 for a random reaching state-action and 0 for all the others.
   - A greedy optimistic policy estimates rewards according to equation 8.5
   - An intrinsically motivated policy estimates rewards according to equation 8.7

3. A policy is generated by value iteration and executed on the robot. If a reach action is executed, generating a positive reward, the policy is considered to have terminated gracefully. Otherwise, if some number of actions is called (10 in the subsequent experiments) the policy is considered to have timed out. In either case, the problem-try is considered finished.

### 8.3.3 Exploitation

Periodically throughout reach learning, the process is paused, and the performance of the agent is evaluated over the entire set of reach problems. The actions taken and rewards received are written to separate history files, such that they do not pollute the internal state of the learning machine. The evaluation process proceeds according to the following steps:

1. The robot is put in a randomly selected state.
    - A random state, $s$, is selected.
    - A reward matrix is initialized to 0 and the action, $a$, is rewarded, which intends to bring the agent to $s$, that is for which $s_g(a) = s$.
    - A policy is generated by value iteration, which the agent follows until a positive reward is generated, or until a timeout is exceeded (10 actions).
2. The set of reaching problems is shuffled, and each one is tried sequentially.
    - A greedy, optimistic reward matrix is estimated according to equations 8.1 and 8.5.
    - A policy is generated by value iteration, which the agent follows until a reach action is tried, generating a positive reward, or until a timeout is exceeded (10 actions).
3. The rewards given for each problem-try are summed, and the evaluation returns the cumulative reward gotten on the set of 27 reach problems.

## 8.4 Experimental Results

In this section I present the results of 15 reach learning experiments, each of which execute 6400 problem-tries. 5 experiments employed random exploration and ran for 207 hours, 5 were greedy and optimistic and ran for 109 hours, and 5 were intrinsically motivated and ran for 113h. All of the experiments used the discount factor, $\gamma = 0.9$, for exploration as well as the exploitation episodes. Visualizations of selected reaches, generated by one of the reach planners presented here, are provided in appendix A.

### 8.4.1 Cumulative Reward Over the Set of Reaching Problems

The performance of the learner was evaluated after 100, 200, 400, 800, 1600, 3200, and 6400 total problem-tries, and each evaluation was run 5 times, to mitigate the effect of the robot's initial (random) pose on the cumulative reward. Thus, there were 25 total evaluations each of random exploration, greedy optimistic exploration, and intrinsically motivated exploration, after 100, 200, 400, 800, 1600, 3200, and 6400 total problem-tries, respectively. The results are tabulated in figure 8.2, which clearly shows that:

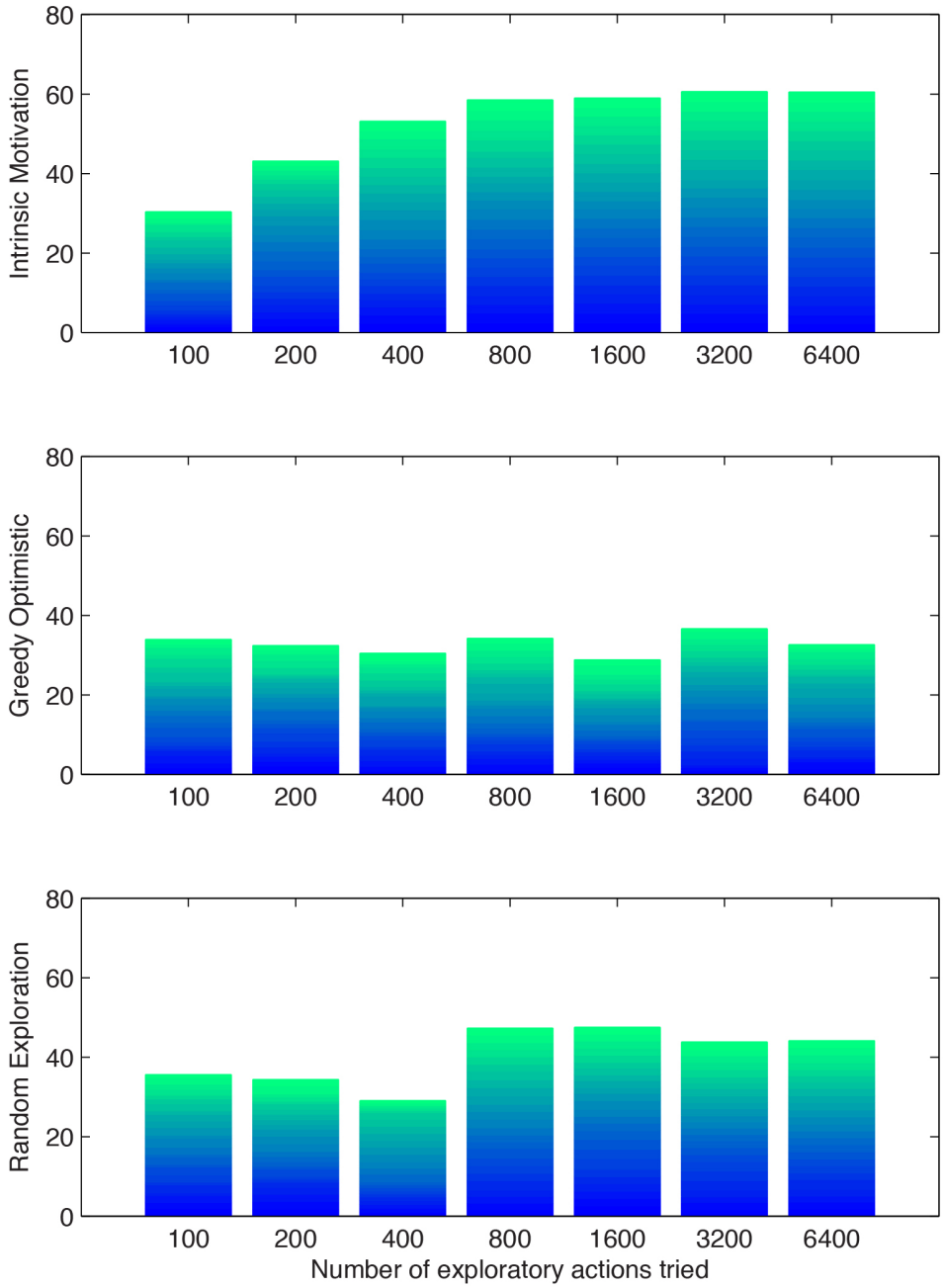


Figure 8.2. Stacked bar charts show the cumulative reward gotten (y-axis) by the reach planner at different stages of learning (x-axis) for three different exploration strategies (section 8.3). Colors indicate results from different (randomly initialized) evaluations.

1. The intrinsically motivated explorer is less competent than the others initially, after 100 problem-tries.

2. However, in contrast to the other exploration strategies, it steadily improves the agent's competency over time.

3. After about 800 problem tries, intrinsically motivated exploration has produced a significantly more competent agent than random exploration or greedy optimistic exploration has.

### 8.4.2 Distribution of Reaching State-Actions Tried During Exploitation Episodes

Additional insight into these exploration strategies can be gained by looking at the distributions of reaching actions the agents selected in order to generate the rewards, tallied in figure 8.2. Comparing the three sequences of state-action distributions for the intrinsically motivated agent (figure 8.4), the greedy optimistic one (8.5), and the one that explores randomly (8.6), the 'quality' of the distributions appears to be well correlated with the amount of reward these agents were able to accumulate during the exploitation phase of the experiments (described in section 8.3.3).

Comparing figure 8.4 with figure 8.2 (top), it is easy to see how the intrinsically motivated agent achieves such a smooth and consistent improvement in its performance over time. After 100 problem tries, the distribution of reaching actions selected is quite broad, and there is little overlap between the sets of reaching state-actions chosen in the different experiments (represented by colors). As the agent gains experience, the predictors improve, and through that improvement, they begin to agree with one another. When the inexperienced agent (figure 8.4 - top) is made to solve the set of reaching problems, it produces broad action selection distributions, which vary from one experiment to the next. However as it becomes more experienced (figure 8.4 - middle), the agent begins to focus on certain state-actions, and those become more consistent from one experiment to the next as the predictors begin to agree with one another. Once the agent is well experienced (figure 8.4 - bottom), it has settled on a few state-actions, its favorite reaches, which give it the best access to the set of target geometries. The two best poses are pictured in figure 8.3.

Keep in mind that the distributions of state-actions selected are the result of greedy policies for reward matrices generated by the predictions of equation 8.5. Therefore, if in the exploitation episode after $n$ exploratory actions, there is a

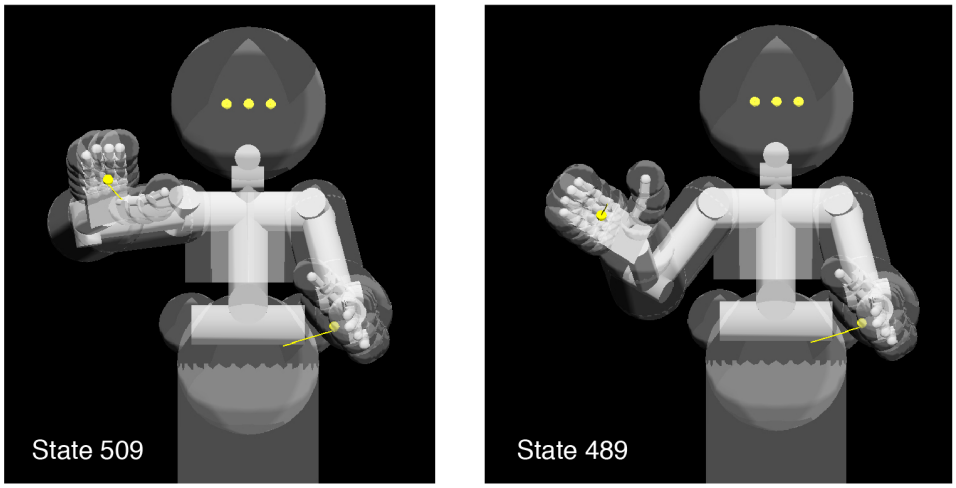


Figure 8.3. The two pre-reach poses most frequently selected during exploitation episodes, which benefitted from intrinsically motivated exploration.

multicolored peak in the state-action distribution at state-action $m$, it can be said that after $n$ exploratory problem-tries, state-action $m$ predicted a high reward for at least some of the RL problems in several separate randomly initialized experiments.

Agreement between experiments is a key feature to look for in these plots. One would not expect to see very much such agreement early in the learning process because the reward predictors are not yet very good. When exploration begins, with zero problem-tries, both the intrinsically motivated learner and the greedy optimistic one predict $r = 1$ for every reaching state-action. Therefore, the sequence of state-actions tried early on is left to chance, and the state of the inexperienced predictors can vary wildly from one experiment to another.

In contrast to the intrinsically motivated agent, the greedy, optimistic agent seems to learn almost nothing (figure 8.5). The distributions of state-actions selected are broad, and they vary not only from one experiment to another but also from one evaluation to another as the same agent gains experience (pick a color and look at the distributions from top to bottom). Greediness is well known to prevent the agent from exploring the entire state-action space. That can obviously prevent the agent from finding the largest rewards, instead settling for some small reward, which it can reliably get. I would have expected this to cause the greedy, optimistic agent to converge on a bad policy, but it does not. Instead, each evaluation on the set of reaching tasks generates a completely different distribution of selected reaches. It would take another view of the

learning process (figure 8.4.3) to fully understand why this agent's performance on exploitation was so stochastic.

Random exploration (figure 8.6) falls somewhere between the other two strategies. It does not show the clear convergence to the preferred state-actions that the intrinsically motivated exploration does, however it does appear to make progress, learning to prefer the state-actions to the right of the distribution, which tend toward holding the hand high.

Initially, as with the other exploration strategies, random exploration produces broad state-action distributions, and there is no consensus between the different experiments. As learning progresses, peaks begin to emerge (200 and 400 problem-tries), demonstrating that the predictors begin to agree with one another across different experiments.

In contrast to the intrinsically motivated experiments however, random exploration does not smoothly increase the competency of the agent. In fact, the cumulative reward gotten by the randomly exploring agent on the exploitation evaluation actually decreases from 100 to 200 to 400 problem-tries. The different experiments' predictors may agree with one another, but they are not always correct. It appears that random exploration does not do as good a job at training the predictors as intrinsically motivated exploration does.

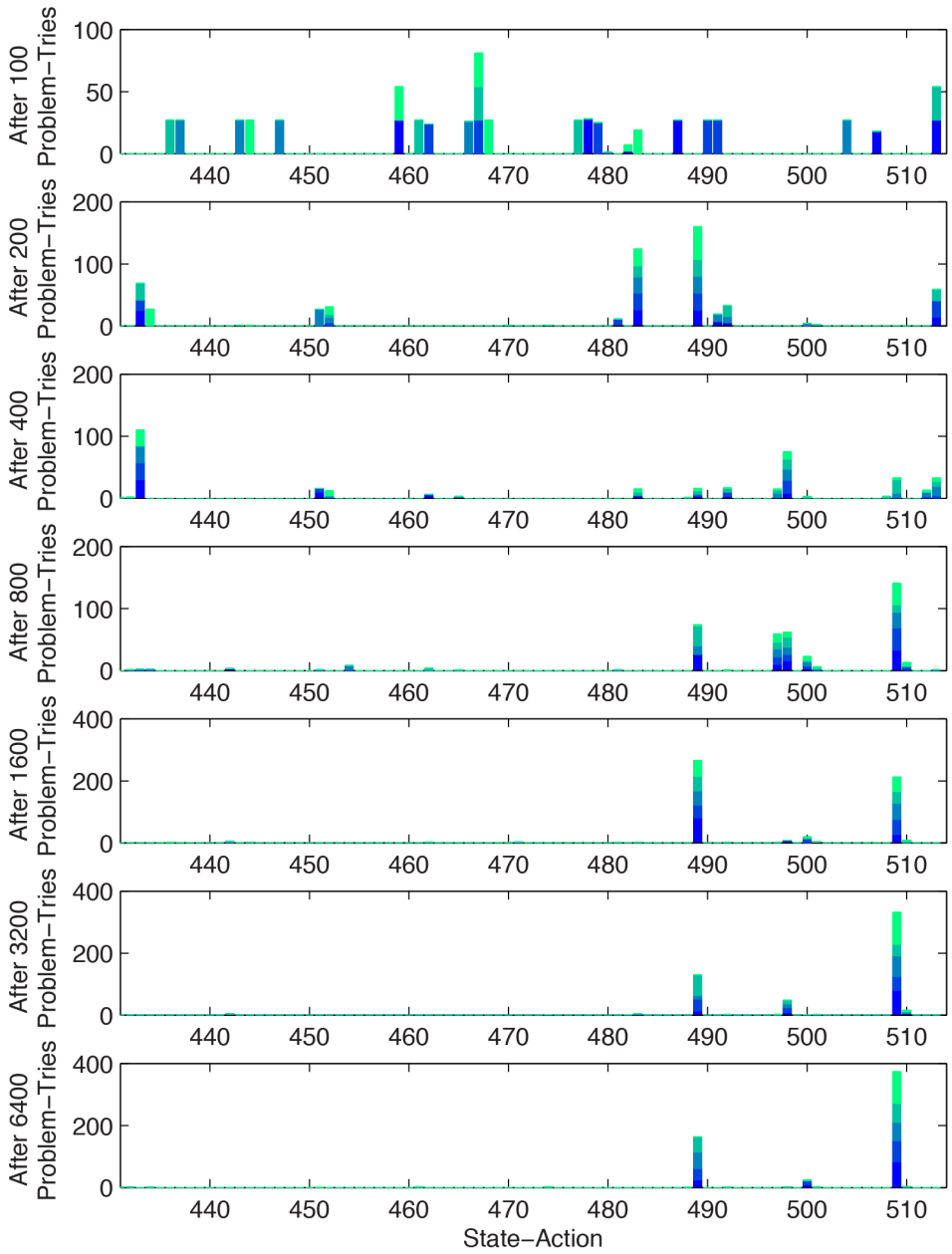


Figure 8.4. Intrinsic Motivation - Distribution of reaching state-actions tried during exploitation episodes. Colors indicate results from different (randomly initialized) evaluations. In every experiment, as learning progresses, the agent settles on the same few actions, which best solve the set of reaching problems.

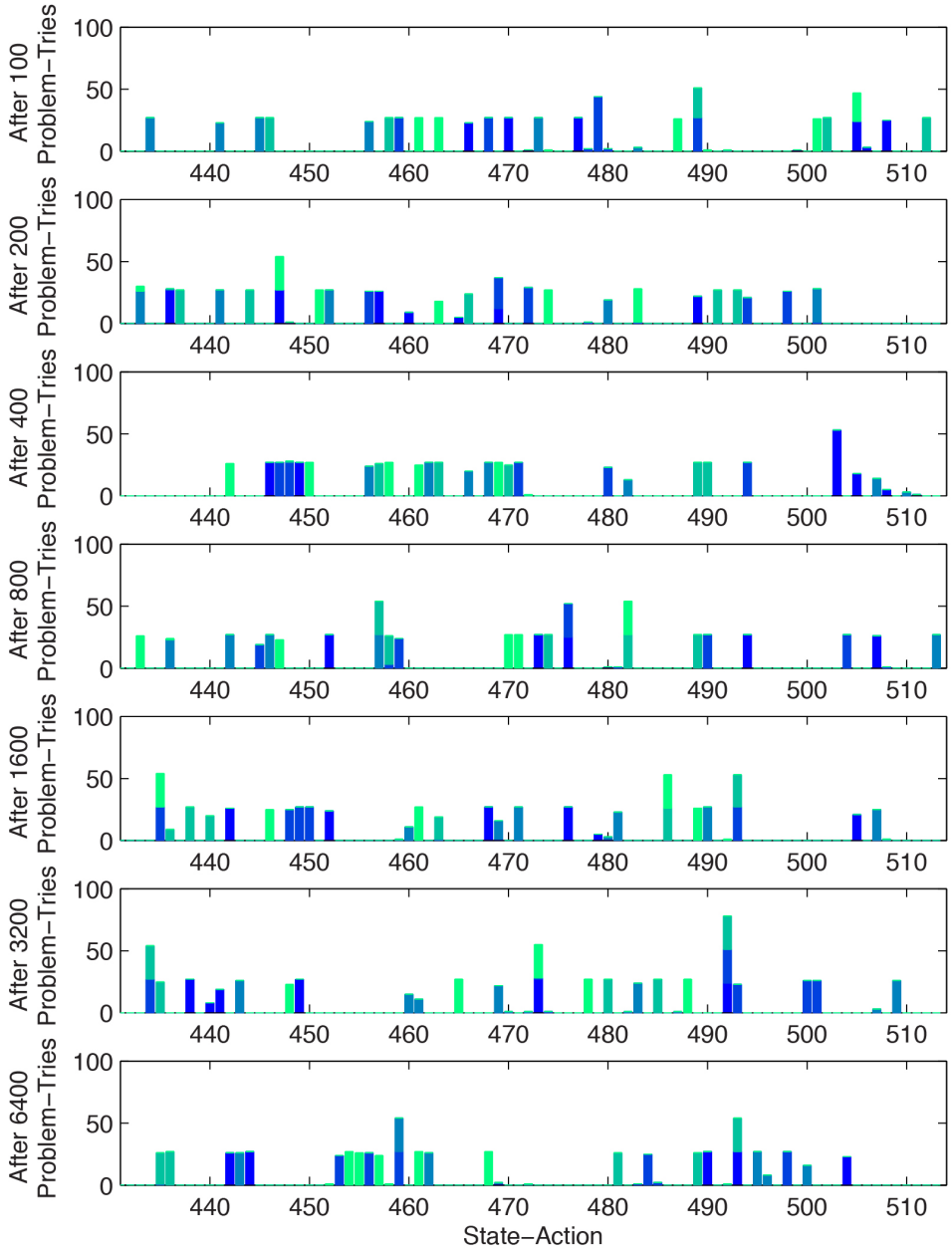


Figure 8.5. Greedy, Optimistic Exploration - Distribution of reaching state-actions tried during exploitation episodes. Colors indicate results from different (randomly initialized) evaluations. As learning progresses, the experiments do not agree on which reaching actions best solve the set of problems, and the distributions remain widely dispersed over most of the state space.

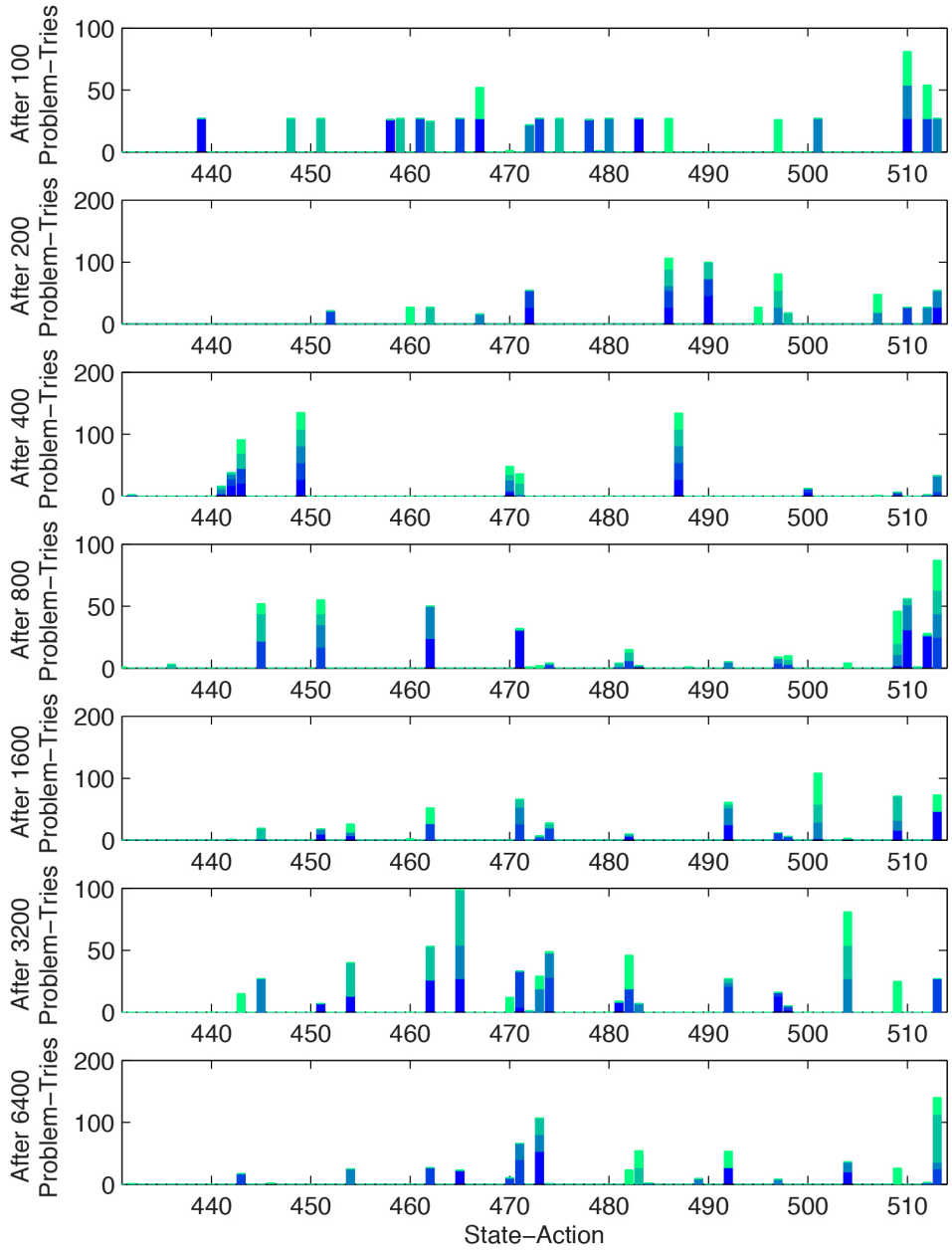


Figure 8.6. Random Exploration - Distribution of reaching state-actions tried during exploitation episodes. Colors indicate results from different (randomly initialized) evaluations. As learning progresses, the experiments do not agree on which reaching actions best solve the set of problems, although the distributions all seem to tend toward the right hand side of the state space, equates to initiating reaches from a high hand position, as the agent gains experience.

### 8.4.3 Exploratory Problem-Try Distributions

The distributions of exploratory problem tries generated by the intrinsically motivated agent look fairly good, as one might expect since the agent does well on the learning task. Particularly after 800 and 1600 problem-tries, the agent seems to be sampling state-action/problem pairs fairly, throughout the space. By 3200 problem tries, the agent has tried almost all possible combinations (there are $81 \cdot 27 = 2187$ possible), and there are a few state actions that have become *interesting* to the agent, particularly regarding certain problems. Although they have been tried several times, the rewards generated by these state-action/problem pairs remain difficult to predict. So far, this is exactly the behavior that one would hope to see from the intrinsically motivated agent.

An important feature emerges in the exploratory problem-try distribution for the intrinsically motivated agent after 6400 problem-tries. The bright points in the previous distribution have been stretched into vertical lines, indicating that the state-actions, which had been interesting when tried on certain problems, have become interesting on all problems. This implies that the predictor may not be able to capture local features of the reward function. In fact, this is not very surprising since the predictor employed is so simple.

In the early stages of learning (up to about 800 problem-tries), the greedy, optimistic agent's sampling behavior and that of the intrinsically motivated agent seem quite similar according to these plots. However, there is an important difference, which becomes obvious after 1600 problem-tries. The familiar vertical lines appear. The agent has found the best state-actions for some problems, and the predictors begin to think that these state-actions are best for all problems. As the agent continues to act, there is little to no further exploration, and it spends all of its time oscillating between the states, which it thinks provide the best pre-reach poses. In fact the two red lines in the distributions for 3200 and 6400 problem tries, correspond to state-actions 489 and 509, the very state-actions that the intrinsically motivated agent used to rack up so much cumulative reward on the problem set.

Apparently, the greedy, optimistic agent found the same best state-actions as the intrinsically motivated agent did, and it took about the same amount of time. One can see the faint beginnings of the vertical lines in the distribution for the greedy, optimistic agent after 800 problem-tries. At that level of experience the exploitation episodes, which had benefitted from the intrinsically motivated

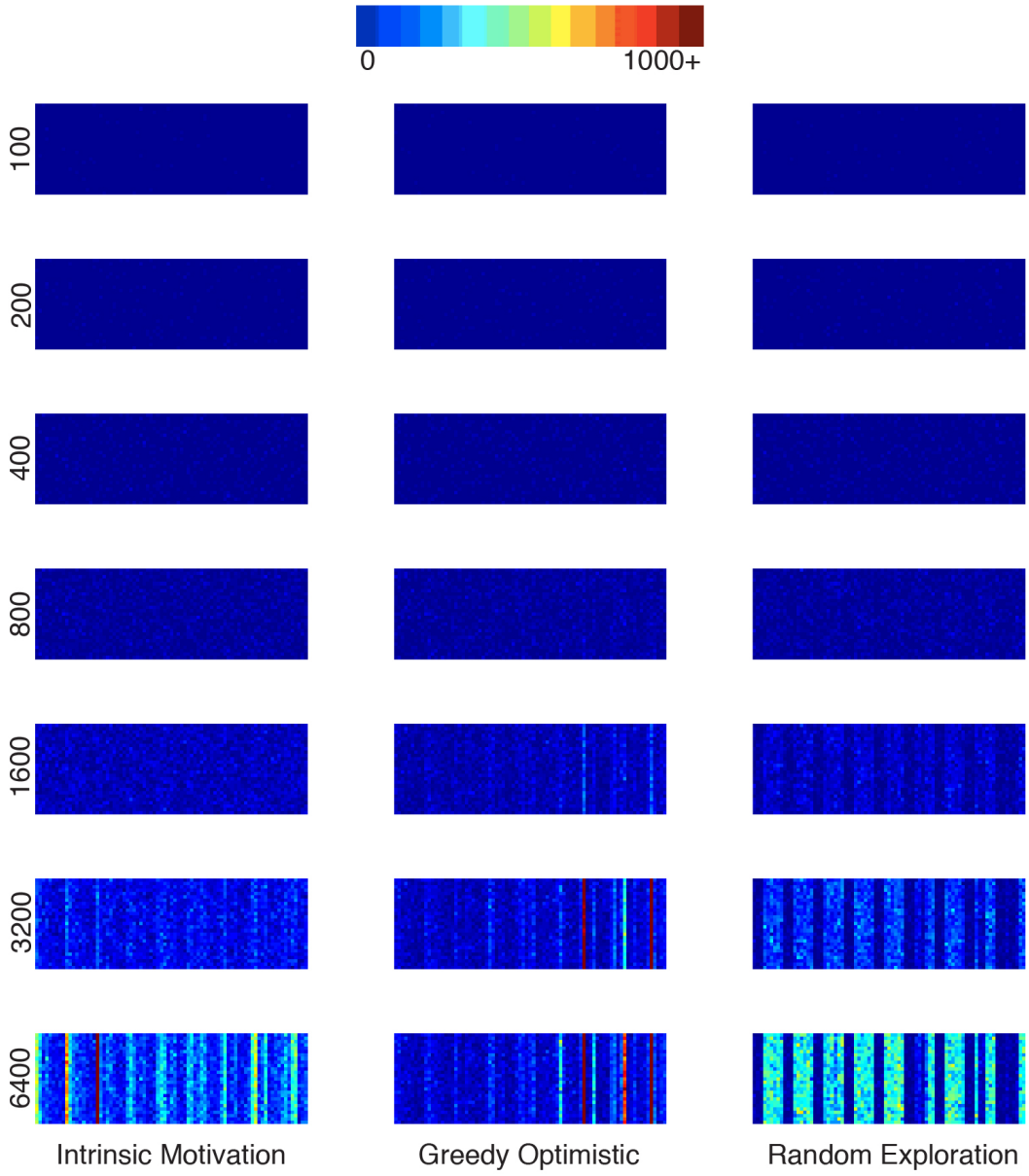


Figure 8.7. Problem Tries During Exploration - Each pixel in these images shows how often a state-action (column) was tried on an RL problem (row), for three different exploration strategies (bottom labels) as the respective agents gain experience (left labels indicate total number of exploratory problem tries. The data are averaged over 5 experimental trials.

exploration began to generate large cumulative rewards. Why then did the greedy, optimistic agent fail to do so?

The reason for the poor performance of the greedy, optimistic agent becomes clear, looking at its exploratory distribution after 1600 problem-tries. There are large regions, which are dark blue. Despite the spatial optimism of equation 8.5, the rewards predicted for problems far from the ones already in the history were not sufficiently large to overcome the discount factor, and consequently, the agent preferred to stay in the neighborhood of the best known state-actions. Still, the spatial optimism was not entirely insignificant. Some reward was indeed predicted for these unexplored regions of the state-action/problem space. Therefore, when the exploitation episodes were initialized with the agent in a random state, it was often quite optimistic about nearby state-actions. This led to seemingly unrelated distributions of state-action tries, each in the neighborhood of a random state, selected to initialize an exploitation episode.

The greedy, optimistic agent knew something about where the good state-actions were, but it knew little about the bad ones, *and* remained optimistic about them. Therefore it tended to exploit when I wanted it to explore and explore when I wanted it to exploit.

Lastly, the state-action distributions for the random exploration also contain some interesting and unexpected features. For the first 800 exploratory problem-tries, the sampling appears to be well distributed, however thereafter appear the familiar vertical structures in the image. This time however, they are not lines, but bands, defined by adjacent regions of the state-action/problem space being sampled or not. The strange thing is that the less sampled regions do contain scattered samples, which appear to be left over from the early stages of exploration. Why would some state-actions be tried early on and then never again? To find the answer, I had to look to the history of negative rewards, given for failed state transition actions.

### 8.4.4 Failing State Transitions

Path planning around the target objects is done by learning which state transition actions are affected by each target. Transitions that do not cause the agent to ascend the value functions are given a large negative reward, such that they are immediately avoided, as described in section 8.1. Figure 8.8 shows the history of such rewards for the same 5 experiments with each of the 3 different agents, which have been discussed above.

Immediately apparent is the fact that the randomly exploring agent accumulates many more negative rewards than do the other two agents. Upon reflec-

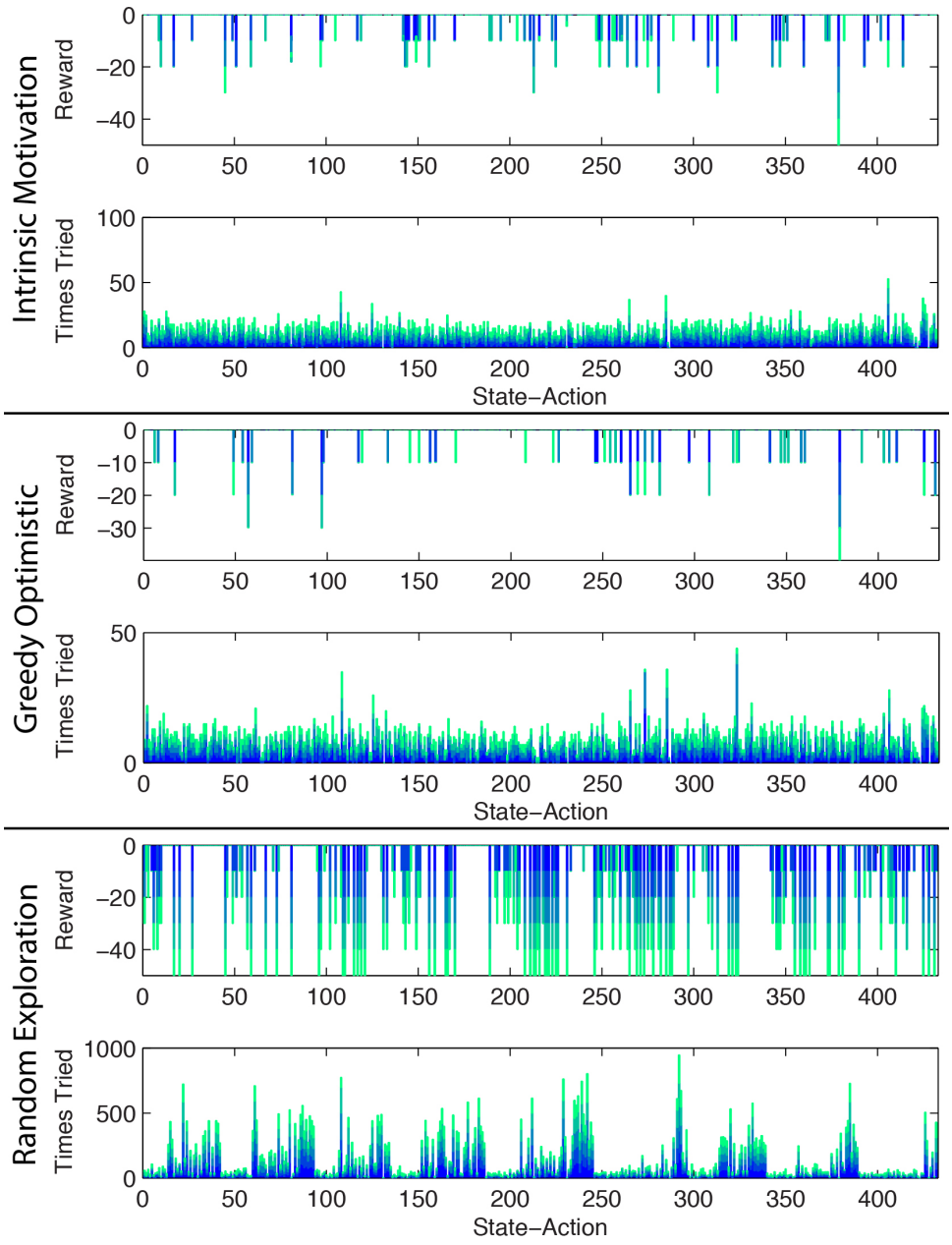


Figure 8.8. Distributions of negative rewards for failed state transitions. State transitions which do not ascend the value function are considered failed. Colors represent results of different experiments. The randomly exploring agent accumulates many more negative rewards than do the other two agents.

tion, the reason for this is simple. When a reward is placed on a random reaching state-action, it may very well be on the other side of the state space. At the very least, it is unlikely to be immediately adjacent to the current state. Therefore, the randomly exploring agent makes many unnecessary trips back and forth across the state space, executing many more state transitions per reaching action. This is why running the 5 experiments of 6400 problem tries takes about twice as long for the random explorer (207 hours) than it does for the other two agents (109 hours for greedy, optimistic, and 113 hours for intrinsic motivation).

Traversing the state space many times during reach learning is not necessarily a bad thing. After all it should give a more complete picture of which state transitions are affected by which target geometries. However, in this case, due to the limitations of my overly simplified predictor, it leads to the bands shown in figure 8.7 (bottom-right).

In the same way that the predictor is overly optimistic regarding good reaches[3], it is overly pessimistic for bad state transitions.

A state transition that fails and generates a negative reward, predicts a smaller negative reward for nearby problems and a much smaller negative reward for far away problems, according to equation 8.1. How small these rewards are though, is relative. If they are large enough, and if the agent accumulated enough of them, they can conspire to block off parts of the state space entirely. This is what prevents the random explorer from accessing regions of state-action/problem space where it had previously been.

## 8.5 Discussion

Perhaps the most pervasive problem with my RL system throughout the experiments presented above is that the predictors proposed in equations 8.1, 8.5, and 8.7 are rather inadequate. In retrospect, this is quite clear. The true reward landscape for each reaching state-action over the workspace (the space of possible RL problems) is likely to contain peaks and valleys resultant of the kinematic mapping as well as interference between the hand/arm and the target object. If reward predictions are made for reach targets based on other targets, which are too far away, important features of the reward landscape may be missed, and predictions my be very wrong indeed.

I had made an effort to address this issue, with the spatially optimistic version of the weighted average (equation 8.5), and preliminary experiments (not pre-

[3]State-actions, which predict high reward for one problem soon predict high reward for all problems in both the intrinsically motivated and greedy, optimistic cases.

sented here) showed that it did help the exploration process significantly. However the more through battery of experiments presented in this chapter showed that:

1. The greedy, optimistic explorer was not sufficiently interested in trying state-actions out on novel problems to train the predictors well.

2. Generally speaking, all of the predictors discussed here suffered from the problem that once they began to predict a large reward (positive or negative alike) for a single state-action/problem pair, they quickly began to do so for that state-action over all problems as the agent gained more and more experience.

It is possible that the capacity of the greedy, optimistic agent to explore could be improved by parameter tuning. If future reward were discounted less, or if spatial optimism were stronger in the predictors, then perhaps the greedy, optimistic agent could be made to explore more like the intrinsically motivated one. Parameter tuning, however makes a brittle solution, which is prone to be sensitive to things like the units used to measure joint angles and workspace distances, as well as the particular kinematics of the robot, and the sequence of reach targets fed into the system during learning.

From a theoretical standpoint, a much more appealing solution (than tuning rewards and discount factors) is intrinsically motivated exploration, which alleviates much of the need for parameter tuning. Moreover, the experiments presented here show that it has real benefits in an applied setting. The intrinsically motivated agent's exploratory behavior is characterized by two very important features:

1. It spends time learning about what it is not good at (predicting low rewards generated by reaches that do not work very well).

2. Its opportunism (in that way it is similar to the greedy optimistic agent) results in a structured traversal of the state-action space.

Because of these two characteristics, intrinsically motivated exploration generates a much better set of training data for the predictors than does random or greedy, optimistic exploration. Therefore, in the experiments presented here, the intrinsically motivated agent is able to get the most out of a relatively bad predictor.

Many of the problems with the predictors themselves could perhaps be mitigated by tuning the weighting terms, or limiting the number of 'neighbor' problems who's rewards are allowed into the sum. However my intuition is that ultimately, the proposed system (particularly the path planning through negative reward prediction) would benefit greatly from a more powerful function approximator for reward prediction, such as a neural network.

# Chapter 9

# Conclusion

My accomplishment in this work has been to conceive of, design, implement, and test a approach to motion synthesis, which is unprecedented in the robotics literature in that it is simultaneously robust to changing constraints, and yet able to re-plan motions quickly, without the need to re-initialize an expensive search.

Initially, I had set out to unify motion planning and reactive control, and RL gave me the abstract formalism necessary to do that. I think that my effort was quite successful in that I was able to learn planners that do not 'break' in the traditional sense of 'planner failure'. Moreover, in a variety of different experimental contexts, I was able to do that learning both in simulation and on real hardware, over thousands of cumulative hours of autonomous operation. After all of the development and experimentation presented in this dissertation, I am convinced that RL is a promising tool for learning reusable models in robotics. However, it remains under-utilized, perhaps because it is misunderstood by many engineers.

In my opinion, the crux of the problem with applying RL to manipulators/humanoids is figuring out the right representations of states, actions, rewards, value functions, and the like. This requires not only knowledge of RL but also a deep domain-specific knowledge. I would therefore argue that the future of developmental robotics may depend on better and/or more fruitful cooperation between roboticists/engineers and AI researchers, who in my experience often operate under different assumptions, have different expectations, and/or speak different professional languages.

Real world reaching is a very hard problem, and while my approach has not solved it for good and all, I do believe that I have made significant progress beyond the state of the art, particularly with respect to robustness and autonomy. Moreover, mine is the first multiple-query planner that can plan reaches *both to*

*and around* arbitrary workspace objects. In my opinion, it justifies the following, important conceptual claim: The combination of a coarse, discrete, preemptive motion planner with a gradient based reactive control strategy leads to much more powerful and robust motion synthesis than either method could do alone.

To the best of my knowledge, this work constitutes the first application of RL to motion planning for a real, physical manipulator with many DOF. As such, it represents only the beginning of what will be possible in years to come. In the remainder of this, the final chapter of my dissertation, I will reflect on some of the things I've accomplished in my time at IDSIA and where this work might lead in years to come.

## 9.1 MoBeE

To prototype different robotics experiments efficiently and effectively requires good system level engineering. This has led to the emergence of many open source projects in robotics[Gerkey et al., 2003; Diankov and Kuffner, 2008; Jackson, 2007; Quigley et al., 2009; Fitzpatrick et al., 2008; Metta et al., 2006; van den Bergen, 2004; The Boost Graph Library; The CGAL Project]. My contribution to the robotics community is MoBeE [Frank et al., 2012a,b], a solid, reusable toolkit for prototyping behaviors on the iCub humanoid robot.

MoBeE represents the state-of-the-art in humanoid robotic control and is similar in conception to the control system that runs DLR's Justin [De Santis et al., 2007; Dietrich et al., 2011]. In the open source spirit of the iCub itself, I contributed the MoBeE source to the iCub repository last year, and I am pleased to report that it has been adopted by other groups. In fact, MoBeE received it's first external citation this year at IROS [Pathak et al., 2013], and if emails to me from other groups are any indication, there will be some more users in the years to come.

Internally at IDSIA, MoBeE has facilitated research activities on topics such as, pure (task relevant) roadmap planning [Stollenga et al., 2013], adaptive roadmap planning [Frank et al., 2012a,b], feature learning for RL [Kompella et al., 2011], learning object localization through robot interaction [Leitner et al., 2012b], and other vision related things [Leitner et al., 2013a,b, 2012a,e,d,c]. Additionally, we have published two videos:

- 'Towards Intelligent Humanoids' (http://vimeo.com/51011081)
- 'Task Relevant Roadmaps' (http://www.youtube.com/watch?v=N6x2e1Zf_yg)

The latter won 'Best Student Video' at the annual AI conference held by the Association for the Advancement of Artificial Intelligence (AAAI) in 2013.

## 9.2 IM-CLeVeR Project

The model learning part of my RL concept [Frank et al., 2013], presented here in chapters 6 and 7, was part of IDSIA's final demonstrations for the EU project Intrinsically Motivated Cumulative Learning Versatile Robots (IM-CLeVeR). I presented the robotics contribution for IDSIA at the final project review meeting for IM-CLeVeR, which was evaluated as 'excellent' by the EU appointed review committee.

The essence of my work on model learning is in a novel application, namely to motion planning in a high DOF configuration space. I observed that many approaches to motion planning in robotics are impractical in the real world because they are based on unrealistic assumptions, or alternatively, they solve different sub-problems related to real-world motion planning. Then I developed an approach to motion planning which simultaneously draws on the different approaches in the robotics literature, by leveraging the very abstract formalism of RL.

Most of the approaches to motion synthesis in robotics fall into two categories. So called *planning algorithms*, search the configuration space, which typically takes a long time, and they assume that the workspace is static, which is often unrealistic over the 'long time' that is required for the search. Contrastingly, *reactive control* assumes that the workspace is dynamic, and can pursue goals while avoiding obstacles, even when everything is moving. However, if one places a large, static obstacle in the way of a reactive controller, it may never find its way around, because it cannot search.

My research posed the simple question, 'What if the sequence of steps comprising a *plan* were not piecewise trajectories, but reactive dynamical systems?' This would of course create all kinds of problems for the planning algorithms usually employed in robotics, but it turned out that planning around arbitrary constraints using a series of dynamical systems could be formulated as an RL problem.

## 9.3 Reaching Experiments

In the wake of the IM-CLeVeR project, I have developed the RL system further, such that it provides not only point-to-point motion planning in configuration space, but can also exploit the motion planner to generate reaches to target objects in the workspace. I accomplished this in part by adding an additional action modality, namely to reach to an object in the workspace, using a naive operational space control based on the Jacobian matrix. However, for the agent to learn from which states it should reach to which target objects, I had to cope with (what is at least from the perspective of a roboticist) a persistent problem with the formulation RL.

In RL, a *task* is defined by a reward function. It is easy to imagine how a reward function might be defined around, say, the agent reducing the error distance between the robot's palm and a target object. However things get complicated when the target object is moved, which at first glance seems to change the reward function, and thus define a new RL problem. The de facto solution to this issue throughout most of the RL literature is build the position of the target object into the state of the system, however this causes the dimensionality of the state space to explode, and there are infinitely many real-valued target object locations in the workspace.

As a roboticist, it seems quite natural to me that each motion planning query should constitute a new RL problem, so I chose to embrace the fact, rather than trying to build the target locations into the state representation. The problem was then to avoid starting from scratch to learn each new policy. To that end, I introduced a mechanism, which could predict reward functions for novel RL problems, based on known reward functions of similar problems. That way, I could exploit the model-based RL of my MDP motion planner to quickly generate an approximate policy that incorporates knowledge already in the system. Moreover, the predictor created the opportunity to carry the concept of intrinsic motivation all the way through the system, and use it not only to explore the model, but also to explore the set of externally rewarded RL problems.

To the best of my knowledge, the notion of tackling a set of related RL problems by predicting reward functions and applying model-based RL is a novel one. I am certain at least that I am the first to apply it to reach planning for robotics. Although experimental evidence showed my predictors to be somewhat inadequate for the task at hand, the approach as a whole seems to me to be quite promising, since the intrinsically motivated RL agent was able to achieve fairly good performance on the set of reaching tasks, which are very, very hard.

In the robotics literature, one often hears about a task called *pick and place*,

which essentially means getting/putting objects from/at different locations in the workspace. It usually takes place at a work table, restricting the resting positions of the objects involved to a 2D plane, and the space above the table is usually assumed to be free of obstacles. In this scenario, a heuristic like reach down from above can work very well, and good performance on the task can be achieved by many approaches in practice. My reaching task is much, much harder. Motions must be planned to move the hand from arbitrary poses (there is no home pose) around the target objects at arbitrary 3D locations.

I am not aware of any approach in the robotics literature that can solve this task robustly in practice[1]. Still, my RL agent does a pretty good job, even for want of a better predictor (see appendix A). I am confident that with the addition of a more powerful function approximator, the RL system would perform much better on the reaching tasks.

## 9.4 Future Work

Throughout my work, I have tried to focus on the whole behavioral control system more than its individual parts, in an effort to connect the dots between related, but until now, separate threads of research, path planning, reactive control, and RL. Now that I have shown what the resulting system can do, I would like to address some of its shortcomings in terms of its constituent parts.

### 9.4.1 Simultaneity of Learning

In the work present here, the MDP motion planners were learned first, then reach planning was done atop the MDP planner. I did it this way primarily to facilitate clean experiments, through which I could validate my RL implementation and report the resulting data clearly. However, now that I have demonstrated both model learning and reach learning, there is no reason that they cannot be done simultaneously.

The only ambiguity in learning the MDP and the reach predictors at the same time arises when an action fails to help the agent climb the value function. Is it a model update or a negative reward associated with the current RL problem? The answer can easily be extracted from the simulated tactile feedback provided by MoBeE. If the simulated skin (force fields) feels something, and if that something is not the robot's own body (self-collisions cause pairs of tactile sensations,

[1] One could plan the motions offline, but moving the target object just a little may invalidate the plan.

whereas collisions with workspace objects do not), then it is a negative reward, otherwise, it is a model update.

This simultaneous learning is appealing from the perspective of anthropomorphism and developmental robotics, and it would make the whole learning process much more autonomous.

### 9.4.2 Parallel Behaviors

The reaching experiments, which were done using the actual iCub hardware, required a way of locating the target object. To this end it was extremely helpful to control the iCub's head and eyes, as that usually produced better localization results. I built a simple reactive control module to do that, and another one to facilitate grasping by closing and reopening the hand. Other such reactive modules could be developed to aid the reach planner, even if they if they operate in parallel with it. For example, I had always had at the bottom of my 'to do' list the idea to implement a reactive controller to keep the palm normal pointed at the target object.

Another excellent application of this would be to use the torso to increase the reach of the arm. All of the reach learning happens in the frame of reference of the iCub's shoulder, and the reach planner can in principle cope with moving objects. Therefore I should be able to control the torso without disturbing the operation of the reach planner. After learning, I essentially have a map of the reachability of the workspace relative to the shoulder. I could build a reactive control module, which puts a target object in a 'good' location relative to the shoulder, such that the dynamic reach works better.

### 9.4.3 Dynamical Systems

All of the attractors I use for control in configuration space and operational space alike are point attractors. I did this because tuning dynamical systems for control of a complex robot like the iCub is a difficult and time consuming task, and the simpler they are the easier it is. Also, I was unsure how the different forcing functions would interact one another (attractors and repulsive forces) and with the RL agent, as I could not find a precedent for a system like mine in the literature.

Now that the integrated reach learning system is built and tested, I think it could benefit from better actions to implement both state transitions and reaching. The reactive control literature is full of all kinds of fancy vector functions,

many of them borrowed from fluid mechanics, which can be used to force dynamical systems and steer robots. I would like to look into how they might apply to my RL system.

### 9.4.4 Reward Predictors

As I mentioned in the previous chapter, weighted averaging makes a poor predictor, and the reach learning would benefit from something more powerful. Perhaps a neural network or a support vector machine would do the job, but I lack expertise in this field, and would need to research the matter to determine what kind of function approximator would best suit my purposes.

### 9.4.5 State-Action Space

The MDP motion planning experiments and subsequently the reach learning ones all employed very simple state spaces consisting of 4D hyper-lattices. This was done primarily to facilitate debugging. The hyper-lattices meant that I could (more or less) picture the 4D geometry of the state space, and that was very helpful in determining whether or not agents were functioning correctly during experiments.

The work on TRMs however, showed what can really be done with roadmap data structures. In the future, MDP motion/reach planning could be greatly enhanced by a more carefully selected state space, based on a roadmap. This opens up a whole plethora of possibilities.

One could simply exploit the TRM technique to optimize the locations of the states, prior to any learning atop the roadmap. Depending on the objective function, this could already be a big improvement over a simple ad-hoc hyper-lattice. However there is no reason to keep the optimization so separate from the learning.

Through model learning, it was discovered that certain states were unreachable. Similarly, reach learning showed that there were certain states, from which reach actions worked well to a wide range of target locations. Both kinds of states mark important features in the configuration space. Perhaps the TRM approach could be more tightly integrated with the RL, working on the state space by building and pruning to develop the most interesting regions and make the boring ones more sparse.

The connectivity of the state space could also be reworked throughout the learning process. Shortcuts could be constructed between frequently visited, far away states, and seldom used or often infeasible transition actions could be

removed. All of this dynamic maintenance of the state-action space would be facilitated by intrinsic motivation. Additions to the state-action space would be interesting, and they would remain so until predictable.

### 9.4.6 Hierarchies of agents

The experiment 'Discovering the Table' (section 7.4.1) is promising with respect to the goal of extending the multi-agent MDP motion planning to hierarchies of agents. The *interesting* (most frequently selected) state-actions, constitute each agent's ability to interact with the others. Therefore they are exactly the actions that should be considered by a parent agent, whose job it would be to coordinate the different body parts. It is my strong suspicion that all state-actions, which are not interesting to the current system, can be compressed as 'irrelevant' in the eyes of such a hypothetical parent agent. However to develop the particulars of the communication up and down the hierarchy remains a difficult challenge, and the topic of ongoing work.

# Appendix A

# Visualizations of Selected Reach Policies

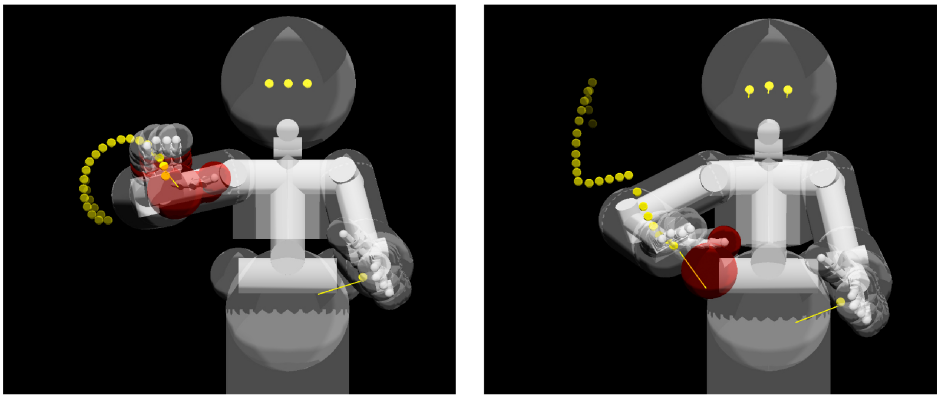

Figure A.1. These reaches were generated by the agent, which benefitted from intrinsically motivated exploration, after the learning experiments presented in section 8.4. The robot was put in a random state, a random reach target was selected (from the set of problems on which the agent has trained), and the RL system generated a policy. The yellow trails mark the history of hand poses throughout the execution of the policy.

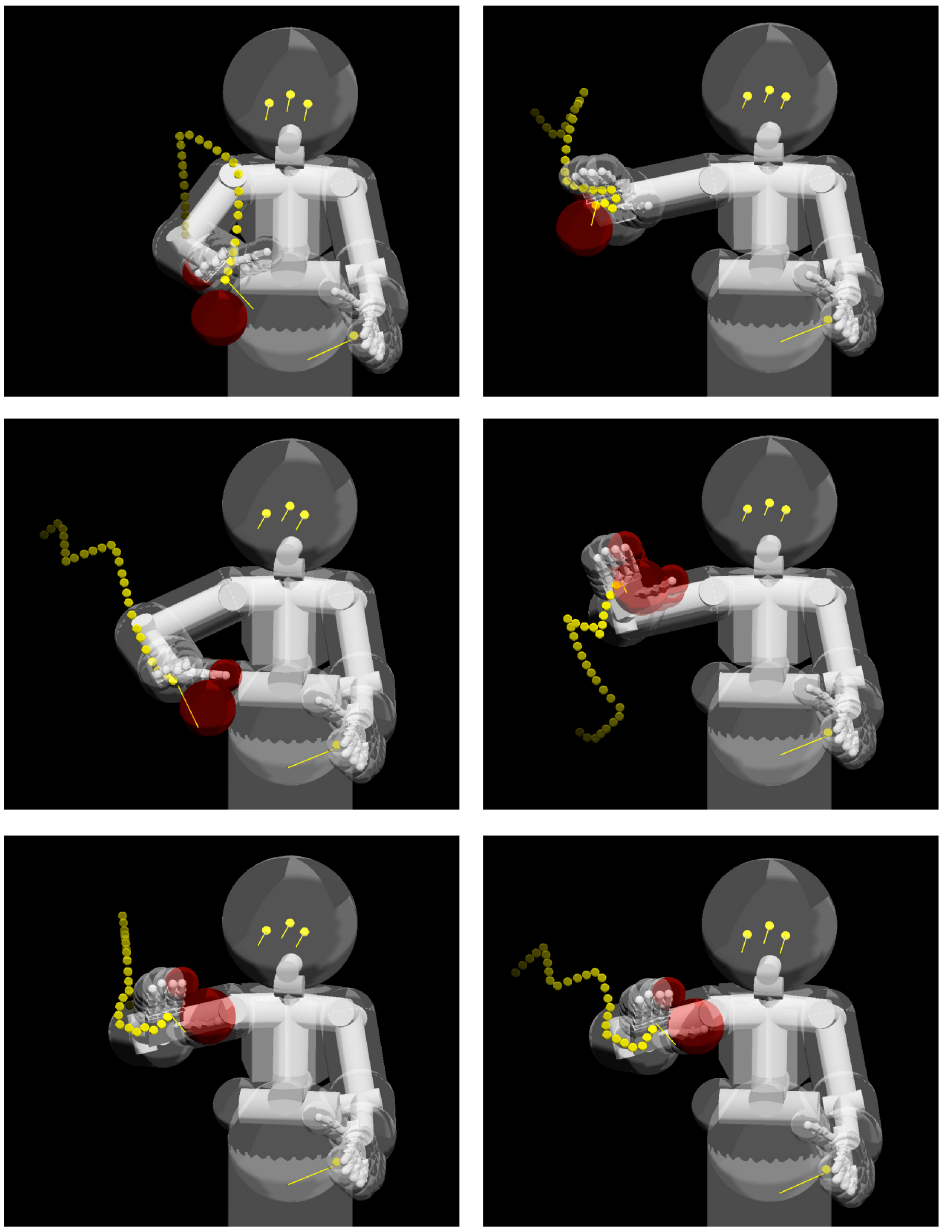

Figure A.2. Example reaches continued.

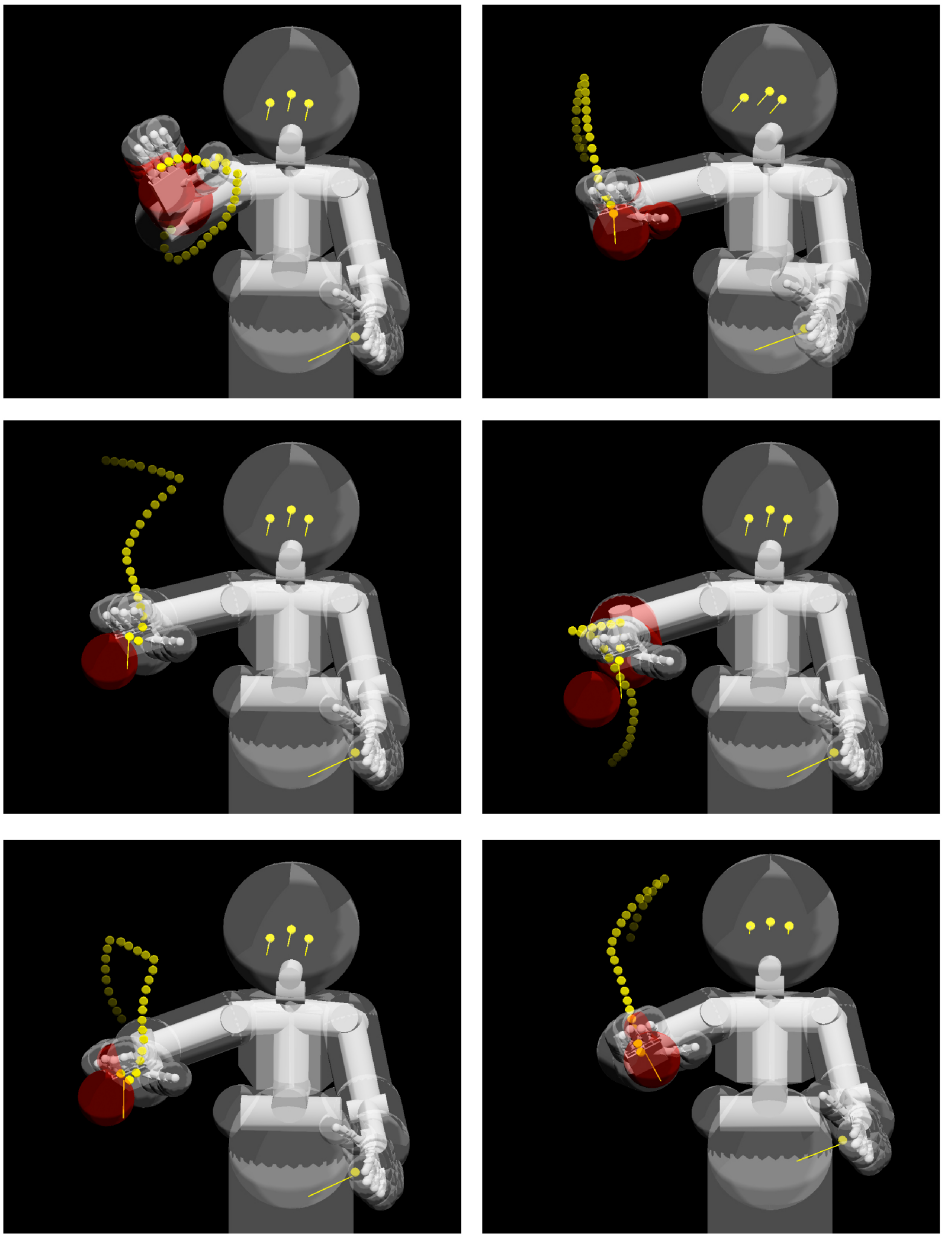

Figure A.3. Example reaches continued.

# Appendix B

# XML Specification - Katana Model

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ZeroPositionKinematicTree>
<ZeroPositionKinematicTree version="1.0" robotName="katana400">

<bodypart name="arm">
  <link z="-.2">
    <box width=".12" depth=".012" height=".1" pz="-0.2"/>
    <motor minPos="6.65" maxPos="352.64" home="6.65">
      <joint minPos="6.65" maxPos="352.64" z="1" radius="0">
        <link z="1" radius="0.04" length="0.19">
          <motor minPos="-15.75" maxPos="124.25" home="124">
            <joint minPos="-15.75" maxPos="124.25" y="1" radius="0.03" length="0.1">
              <link x="-1" radius="0.02" length="0.19">
                <motor minPos="52.7" maxPos="302.69">
                  <joint minPos="52.7" maxPos="302.69" y="1" radius="0.03" length="0.1">
                    <link x="1" radius="0.02" length="0.139">
                      <motor minPos="63.5" maxPos="293.5">
                        <joint minPos="63.5" maxPos="293.5" y="-1" radius="0.03" length="0.1">
                          <motor minPos="8.5" maxPos="350.5">
                            <joint minPos="8.5" maxPos="350.5" x="-1" radius="0">
                              <link x="-1" radius="0.02" length="0.185">

  <!-- MANIPULATOR -->
    <motor minPos="-121.58" maxPos="8.7" home="-50">
      <joint minPos="90" maxPos="180" y="-1" radius="0">
        <link z="-.13" radius="0">
          <box width=".01" depth=".13" height=".03"  px="-.01" pz=".066"/>
        </link>
      </joint>
      <joint minPos="90" maxPos="180" y="1" radius="0">
```

```
        <link z=".13" radius="0">
          <box width=".01" depth=".13" height=".03" px="-.01" pz="-.066"/>
        </link>
      </joint>
    </motor>
  <!-- / MANIPULATOR -->

                              </link>
                            </joint>
                          </motor>
                        </joint>
                      </motor>
                    </link>
                  </joint>
                </motor>
              </link>
            </joint>
          </motor>
        </link>
      </joint>
    </motor>
  </link>
</bodypart>

</ZeroPositionKinematicTree>
```

# Bibliography

S. Amari. Natural gradient works efficiently in learning. *Neural computation*, 10(2):251–276, 1998.

J.M. Badger, S.W. Hart, and J.D. Yamokoski. Towards autonomous operation of robonaut 2. 2011.

P. Baerlocher and R. Boulic. An inverse kinematics architecture enforcing an arbitrary number of strict priority levels. *The visual computer*, 20(6):402–417, 2004.

A. Balestrino, G. De Maria, and L. Sciavicco. Robust control of robotic manipulators. In *Proceedings of the 9th IFAC World Congress*, volume 5, pages 2435–2440, 1984.

A.G. Barto, R.S. Sutton, and C.W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5): 834–846, 1983.

A.G. Barto, S. Singh, and N. Chentanez. Intrinsically motivated learning of hierarchical collections of skills. In *Proceedings of the 3rd International Conference on Development and Learning (ICDL 2004)*, pages 112–19, 2004.

R.E. Bellman. On the theory of dynamic programming. In *Proceedings of the National Academy of Sciences*, 1952.

R.E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ., 1957.

D. Berenson, S.S. Srinivasa, D. Ferguson, and J.J. Kuffner. Manipulation planning on constraint manifolds. In *IEEE Int. Conf. on Robotics and Automation (ICRA)*, pages 625–632. IEEE, 2009.

D. Berenson, S. Srinivasa, and J. Kuffner. Task space regions a framework for pose-constrained manipulation planning. *The Int. Journal of Robotics Research*, 30(12):1435–1460, 2011.

N. E. Berthier. The syntax of human infant reaching. In *Unifying Themes in Complex Systems: Proceedings of the Eighth Int. Conf. on Complex Systems*, volume Volume VIII of *New England Complex Systems Institute Series on Complexity*, pages 1477–1487. NECSI Knowledge Press, 2011.

N. E. Berthier, R. K. Clifton ad D. D McCall, and D. J. Robin. Proximodistal structure of early reaching in human infants. *Experimental Brain Research*, 127:259–269, 1999.

D.P. Bertsekas and J.N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-hall, Englewood Cliffs, NJ, 1989.

D.P. Bertsekas and J.N. Tsitsiklis. *Neuro-dynamic programming*. Athena Scientific, 1996.

O. Brock and O. Khatib. Real-time re-planning in high-dimensional configuration spaces using sets of homotopic paths. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, volume 1, pages 550–555. IEEE, 2000.

R.A. Brooks. Intelligence without representation. *Artificial intelligence*, 47(1):139–159, 1991.

S.R. Buss and J. Kim. Selectively damped least squares for inverse kinematics. *Journal of Graphics, GPU, and Game Tools*, 10(3):37–49, 2005.

E. Chinellato, M. Antonelli, B. J. Grzyb, and A. P. del Pobil. Implicit sensorimotor mapping of the peripersonal space by gazing and reaching. *TAMD*, 3(1):43–53, 2011.

Eris Chinellato, Beata J Grzyb, Nicoletta Marzocchi, Annalisa Bosco, Patrizia Fattori, and Angel P Del Pobil. Eye-hand coordination for reaching in dorsal stream area v6a: Computational lessons. In *Bioinspired Applications in Artificial and Natural Computation*, pages 304–313. Springer, 2009.

N. Courty and E. Arnaud. Inverse kinematics using sequential monte carlo methods. *Articulated Motion and Deformable Objects*, pages 1–10, 2008.

S. Cousins. Ros on the pr2 [ros topics]. *Robotics & Automation Magazine, IEEE*, 17(3):23–25, 2010.

A. De Santis, A. Albu-Schaffer, C. Ott, B. Siciliano, and G. Hirzinger. The skeleton algorithm for self-collision avoidance of a humanoid manipulator. In *Advanced intelligent mechatronics, 2007 IEEE/ASME international conference on*, pages 1–6. IEEE, 2007.

J. Denavit and R.S. Hartenberg. A kinematic notation for lower-pair mechanisms based on matrices. *Trans. of the ASME. Journal of Applied Mechanics*, 22:215–221, 1955.

A.S. Deo and I.D. Walker. Adaptive non-linear least squares for inverse kinematics. In *Robotics and Automation, 1993. Proceedings., 1993 IEEE International Conference on*, pages 186–193. IEEE, 1993.

R. Diankov and J. Kuffner. Openrave: A planning architecture for autonomous robotics. Technical Report CMU-RI-TR-08-34, Carnagie Mellon University, Robotics Institute, 2008.

A. Dietrich, T. Wimbock, H. Taubig, A. Albu-Schaffer, and G. Hirzinger. Extensions to reactive self-collision avoidance for torque and position controlled humanoids. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 3455–3462. IEEE, 2011.

M. Diftler, J. Mehling, M. Abdallah, N. Radford, L. Bridgwater, A. Sanders, S. Askew, D. Linn, J. Yamokoski, F. Permenter, B. Hargrave, R. Platt, R. Savely, and R. Ambrose. Robonaut 2: The first humanoid robot in space. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*, 2011.

E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1): 269–271, 1959.

A. D'Souza, S. Vijayakumar, and S. Schaal. Learning inverse kinematics. In *Intelligent Robots and Systems, 2001. Proceedings. 2001 IEEE/RSJ International Conference on*, volume 1, pages 298–303 vol.1, 2001. doi: 10.1109/IROS.2001.973374.

M.S. Dutra, I.L. Salcedo, and L.M.P. Diaz. New technique for inverse kinematics problems using simulated annealing. In *Int. Conf. on Engineering Optimization*, pages 01–05, 2008.

V.V. Fedorov. Theory of optimal experiments. 1972.

P. Fitzpatrick, G. Metta, and L. Natale. Towards long-lived robot genes. *Robotics and Autonomous Systems*, 56(1):29–45, 2008. ISSN 0921-8890.

M. Frank, A. Förster, and J. Schmidhuber. Reflexive collision response with virtual skin - roadmap planning meets reinforcement learning. In *International Conference on Agents and Artificial Intelligence (ICAART)*, pages 642–651, 2012a.

M. Frank, J. Leitner, M. Stollenga, G. Kaufmann, S. Harding, A. Förster, and J. Schmidhuber. The modular behavioral environment for humanoids and other robots (mobee). In *9th International Conference on Informatics in Control, Automation and Robotics (ICINCO)*, 2012b.

M. Frank, J. Leitner, M. Stollenga, A. Förster, and J. Schmidhuber. Curiosity driven reinforcement learning for motion planning on humanoids. *Frontiers in Neurorobotics*, 7(25), 2013. ISSN 1662-5218. doi: 10.3389/fnbot.2013.00025. URL `http://www.frontiersin.org/neurorobotics/10.3389/fnbot.2013.00025/abstract`.

B.P. Gerkey, R.T. Vaughan, and A. Howard. The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems. In *International Conference on Advanced Robotics*, pages 317–323, 2003.

T. Glasmachers, T. Schaul, S. Yi, D. Wierstra, and J. Schmidhuber. Exponential natural evolution strategies. In *12th annual conference on Genetic and Evolutionary Computation*, pages 393–400. ACM, 2010.

A.A. Goldenberg, B. Benhabib, and R.G. Fenton. A complete generalized solution to the inverse kinematics of robots. *Robotics and Automation, IEEE Journal of*, 1(1):14–20, 1985.

G. Gordon and E. Ahissar. Reinforcement active learning hierarchical loops. In *Neural Networks (IJCNN), The 2011 International Joint Conference on*, pages 3008–3015. IEEE, 2011.

G. Gordon and E. Ahissar. A curious emergence of reaching. In *Advances in Autonomous Robotics, Joint Proceedings of the 13th Annual TAROS Conference and the 15th Annual FIRA RoboWorld Congress*, pages 1–12. Springer Berlin Heidelberg, 2012. doi: 10.1007/978-3-642-32527-4_1. Bristol, UK, August 20-23.

M. Grüttner, F. Sehnke, T. Schaul, and J. Schmidhuber. Multi-Dimensional Deep Memory Atari-Go Players for Parameter Exploring Policy Gradients. In *Proceedings of the International Conference on Artificial Neural Networks ICANN*, pages 114–123. Springer, 2010.

K.C. Gupta. Kinematic analysis of manipulators using the zero reference position description. *The International Journal of Robotics Research*, 5(2):5, 1986.

N. Hansen, S. Müller, and P. Koumoutsakos. Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (cma-es). *Evolutionary Computation*, 11 (1):1–18, 2003.

P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, 1968.

K. Hauser, V. Ng-Thow-Hing, and H. Gonzalez-Baños. Multi-modal motion planning for a humanoid robot manipulation task. *Robotics Research*, pages 307–317, 2011.

C. Hecker, B. Raabe, R.W. Enslow, J. DeWeese, J. Maynard, and K. van Prooijen. Real-time motion retargeting to highly varied user-created morphologies. In *ACM Transactions on Graphics (TOG)*, volume 27, page 27. ACM, 2008.

A. Hemami. A more general closed-form solution to the inverse kinematics of mechanical arms. *Advanced robotics*, 2(4):315–325, 1987.

S. Hespos, G. Gredebäck, C. von Hofsten, and E. S. Spelke. Occlusion is hard: comparing predictive reaching for visible and hidden objects in infants and adults. *Cognitive Science*, 33: 1483–1502, 2009.

X. Huang and J. Weng. Novelty and reinforcement learning in the value system of developmental robots. In *Lund University Cognitive Studies*, pages 47–55, 2002.

I. Iossifidis and G. Schöner. Autonomous reaching and obstacle avoidance with the anthropomorphic arm of a robotic assistant using the attractor dynamics approach. In *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, volume 5, pages 4295–4300. IEEE, 2004. doi: http://dx.doi.org/10.1109/ROBOT.2004.1302393.

I. Iossifidis and G. Schöner. Reaching with a redundant anthropomorphic robot arm using attractor dynamics. *VDI BERICHTE*, 1956:45, 2006.

L. Itti and P.F. Baldi. Bayesian surprise attracts human attention. *Advances in neural information processing systems*, pages 547–554, 2005.

J. Jackson. Microsoft robotics studio: A technical introduction. *IEEE Robotics & Automation Magazine*, 14(4):82–87, 2007.

L.P. Kaelbling, M.L. Littman, and A.W. Moore. Reinforcement learning: A survey. *arXiv preprint cs/9605103*, 1996.

M. Kalakrishnan, S. Chitta, E. Theodorou, P. Pastor, and S. Schaal. Stomp: Stochastic trajectory optimization for motion planning. In *IEEE Int. Conf. on Robotics and Automation (ICRA)*, pages 4569–4574. IEEE, 2011.

M. Kallmann, Y. Huang, and R. Backman. A skill-based motion planning framework for humanoids. In *Robotics and Automation (ICRA), 2010 IEEE Int. Conf. on*, pages 2507–2514. IEEE, 2010.

G. Kaufmann. A flexible and safe environment for robotic experiments : a sandbox and testbed for experiments intended for the humanoid robot icub. Master's thesis, Università della Svizzera italiana (USI), 2010.

M. Kauschke. Closed form solutions applied to redundant serial link manipulators. *Mathematics and Computers in Simulation*, 41(5):509–516, 1996.

K. Kazerounian, K. Latif, C. Alvarado, et al. Protofold: A successive kinetostatic compliance method for protein conformation prediction. *Journal of Mechanical Design*, 127:712, 2005a.

K. Kazerounian, K. Latif, K. Rodriguez, C. Alvarado, et al. Nano-kinematics for analysis of protein molecules. *Journal of Mechanical Design*, 127:699, 2005b.

O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *The international journal of robotics research*, 5(1):90, 1986.

J.O. Kim and P.K. Khosla. Real-time obstacle avoidance using harmonic potential functions. *Robotics and Automation, IEEE Transactions on*, 8(3):338–349, 1992.

V. Kompella, L. Pape, J. Masci, M. Frank, and J. Schmidhuber. Autoincsfa and vision-based developmental learning for humanoid robots. In *Humanoid Robots (Humanoids), 2011 11th IEEE-RAS International Conference on*, pages 622–629. IEEE, 2011.

V.R. Kompella, M. Luciw, M. Stollenga, L. Pape, and J. Schmidhuber. Autonomous learning of abstractions using curiosity-driven modular incremental slow feature analysis. In *Proc. Joint Int'l Conf. Development and Learning and Epigenetic Robotics (ICDL-EPIROB-2012)*, San Diego, 2012.

Y. Kusuda. Toyota's violin-playing robot. *Industrial Robot: An International Journal*, 35(6):504–506, 2008.

J.C. Latombe, L.E. Kavraki, P. Svestka, and M. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4):566–580, 1996.

S.M. LaValle. Rapidly-exploring random trees: A new tool for path planning. Technical report, Computer Science Dept., Iowa State University, 1998.

S.M. LaValle. *Planning algorithms*. Cambridge Univ Pr, 2006.

C.S.G. Lee and M. Ziegler. Geometric approach in solving inverse kinematics of puma robots. *Aerospace and Electronic Systems, IEEE Transactions on*, (6):695–706, 1984.

J. Leitner, P. Chandrashekhariah, S. Harding, M. Frank, G. Spina, A. Förster, J. Triesch, and J. Schmidhuber. Autonomous learning of robust visual object detection and identification on a humanoid. In *Development and Learning and Epigenetic Robotics (ICDL), 2012 IEEE International Conference on*, pages 1 –6, nov. 2012a. doi: 10.1109/DevLrn.2012.6400826.

J. Leitner, S. Harding, M. Frank, A. Förster, and J. Schmidhuber. Learning spatial object localization from vision on a humanoid robot. *International Journal of Advanced Robotic Systems*, 9: ISBN: 1729–8806, 2012b. doi: 10.5772/54657.

J. Leitner, S. Harding, M. Frank, A. Förster, and J. Schmidhuber. icvision: A modular vision system for cognitive robotics research. In *5th International Conference on Cognitive Systems (CogSys)*, Feb 2012c.

J. Leitner, S. Harding, M. Frank, A. Förster, and J. Schmidhuber. Towards spatial perception: Learning to locate objects from vision. In Joanna Szufnarowska, editor, *Proceedings of the Post-Graduate Conference on Robotics and Development of Cognition*, pages 20–23, September 2012d.

J. Leitner, S. Harding, M. Frank, A. Förster, and J. Schmidhuber. Transferring spatial perception between robots operating in a shared workspace. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 1507 –1512, oct. 2012e. doi: 10.1109/IROS.2012.6385642.

J. Leitner, S. Harding, P. Chandrashekhariah, M. Frank, A. Förster, J. Triesch, and J. Schmidhuber. Learning visual object detection and localisation using icVision. *Biologically Inspired Cognitive Architectures*, (0):–, 2013a. ISSN 2212-683X. doi: http://dx.doi.org/10.1016/j.bica.2013.05.009. URL `http://www.sciencedirect.com/science/article/pii/S2212683X13000443`. proof.

J. Leitner, S. Harding, M. Frank, A. Förster, and J. Schmidhuber. Humanoid learns to detect its own hands. In *Congress on Evolutionary Computing (CEC)*. IEEE, 2013b. accepted.

J. Leitner, S. Harding, M. Frank, A. Förster, and J. Schmidhuber. Artificial neural networks for spatial perception: Towards visual object localisation in humanoid robots. In *International Joint Conference on Neural Networks (IJCNN)*, 2013c. accepted.

T.Y. Li and Y.C. Shie. An incremental learning approach to motion planning with roadmap management. *Journal of Information Science and Engineering*, 23(2):525–538, 2007.

D.V. Lindley. On a measure of the information provided by an experiment. *The Annals of Mathematical Statistics*, pages 986–1005, 1956.

M. Luciw, V. Graziano, M. Ring, and J. Schmidhuber. Artificial curiosity with planning for autonomous perceptual and cognitive development. 2:1–8, 2011.

M. Lungarella, G. Metta, R. Pfeifer, and G. Sandini. Developmental robotics: a survey. *Connection Science*, 15(4):151–190, 2003.

G. Metta, P. Fitzpatrick, and L. Natale. YARP: Yet Another Robot Platform. *International Journal of Advanced Robotic Systems*, 3(1), 2006.

G. Metta, G. Sandini, D. Vernon, L. Natale, and F. Nori. The icub humanoid robot: an open platform for research in embodied cognition. In *Proceedings of the 8th Workshop on Performance Metrics for Intelligent Systems*, pages 50–56. ACM, 2008.

RC Miall and Daniel M Wolpert. Forward models for physiological motor control. *Neural networks*, 9(8):1265–1279, 1996.

J. Mugan and B. Kuipers. Autonomous learning of high-level states and actions in continuous environments. *IEEE Transactions on Autonomous Mental Development*, 4(1):70–86, 2012.

Y. Nakamura and H. Hanafusa. Inverse kinematic solutions with singularity robustness for robot manipulator control. *Journal of dynamic systems, measurement, and control*, 108(3):163–171, 1986.

A. Needham, T. Barrett, and K. Peterman. A pick-me-up for infants' exploratory skills: early stimulated experiences reaching for objects using 'sticky mittens' enhances young infants' object exploration skills. *Infant Behavior and Development*, 25:279–295, 2002.

H. Ngo, M. Luciw, A. Foerster, and J. Schmidhuber. Learning skills from play: Artificial curiosity on a katana robot arm. In *Proc. of the 2012 International Joint Conference of Neural Networks (IJCNN)*, Brisbane, Australia, 2012.

F. Nori, L. Natale, G. Sandini, and G. Metta. Autonomous learning of 3d reaching in a humanoid robot. In *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on*, pages 1142–1147. IEEE, 2007.

P.Y. Oudeyer, F. Kaplan, and V.V. Hafner. Intrinsic motivation systems for autonomous mental development. *Evolutionary Computation, IEEE Transactions on*, 11(2):265–286, 2007.

L. Pape, C.M. Oddo, M. Controzzi, C. Cipriani, A. Förster, M.C. Carrozza, and J. Schmidhuber. Learning tactile skills through curious exploration. *Frontiers in Neurorobotics*, 6, 2012.

S. Pathak, L. Pulina, G. Metta, and A. Tacchella. Ensuring safety of policies learned by reinforcement: Reaching objects in the presence of obstacles with the icub. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2013.

A. Perez, S. Karaman, A. Shkolnik, E. Frazzoli, S. Teller, and M.R. Walter. Asymptotically-optimal path planning for manipulation using incremental sampling-based algorithms. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 4307–4313. IEEE, 2011.

J. Peters and S. Schaal. Learning to control in operational space. *The International Journal of Robotics Research*, 27(2):197, 2008a.

J. Peters and S. Schaal. Reinforcement learning of motor skills with policy gradients. *Neural Networks*, 21(4):682–697, 2008b.

J. Peters and S. Schall. Natural actor critic. *Neurocomputing*, 71(7-9):1180–1190, 2008.

J. Peters, S. Vijayakumar, and S. Schaal. Reinforcement learning for humanoid robotics. In *Proc. of IEEE/RSJ International Conference on Humanoid Robotics*, 2003.

J. Piaget and M.T. Cook. The origins of intelligence in children. 1952.

M.L. Puterman. *Markov decision processes: discrete stochastic dynamic programming*, volume 414. John Wiley & Sons, 2009.

M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng. ROS: an open-source Robot Operating System. In *International Conference on Robotics and Automation, Open-Source Software workshop*, 2009.

T. Rückstieß, M. Felder, and J. Schmidhuber. State-dependent exploration for policy gradient methods. In *ECML/PKDD (2)*, pages 234–249, 2008a.

T. Rückstieß, M. Felder, and J. Schmidhuber. State-Dependent Exploration for policy gradient methods. In W. Daelemans et al., editor, *European Conference on Machine Learning (ECML) and Principles and Practice of Knowledge Discovery in Databases 2008, Part II, LNAI 5212*, pages 234–249, 2008b.

W. Schenck, H. Hoffmann, and R. Möller. Learning internal models for eye-hand coordination in reaching and grasping. In F. Schmalhofer, R. M. Young, and G. Katz, editors, *EuroCogSci*, pages 289–294. Lawrence Erlbaum Associates, 2003.

J. Schmidhuber. A possibility for implementing curiosity and boredom in model-building neural controllers. In J. A. Meyer and S. W. Wilson, editors, *International Conference on Simulation of Adaptive Behavior: From Animals to Animats*, pages 222–227. MIT Press/Bradford Books, 1991a.

J. Schmidhuber. Curious model-building control systems. In *Proceedings of the International Joint Conference on Neural Networks, Singapore*, volume 2, pages 1458–1463. IEEE press, 1991b.

J. Schmidhuber. Developmental robotics, optimal artificial curiosity, creativity, music, and the fine arts. *Connection Science*, 18(2):173–187, 2006.

J. Schmidhuber. PowerPlay: Training an Increasingly General Problem Solver by Continually Searching for the Simplest Still Unsolvable Problem. *Frontiers in Psychology*, 2013. doi: 10.3389/fpsyg.2013.00313.

R. Schneider. *Convex bodies: the Brunn–Minkowski theory*, volume 151. Cambridge University Press, 2013.

G. Schöner and M. Dose. A dynamical systems approach to task-level system integration used to plan and control autonomous vehicle motion. *Robotics and Autonomous Systems*, 10(4): 253–267, 1992.

F. Sehnke, C. Osendorfer, T. Rückstieß, A. Graves, J. Peters, and J. Schmidhuber. Policy gradients with parameter-based exploration for control. In *Proceedings of the International Conference on Artificial Neural Networks ICANN*, 2008.

F. Sehnke, A. Graves, C. Osendorfer, and J. Schmidhuber. Multimodal Parameter-exploring Policy Gradients. In *2010 Ninth International Conference on Machine Learning and Applications*, pages 113–118. IEEE, 2010a.

F. Sehnke, C. Osendorfer, T. Rückstieß, A. Graves, J. Peters, and J. Schmidhuber. Parameter-exploring policy gradients. *Neural Networks*, 23(4):551–559, 2010b.

L. Sentis and O. Khatib. A whole-body control framework for humanoids operating in human environments. In *IEEE Int. Conf. on Robotics and Automation (ICRA)*, pages 2641–2648. IEEE, 2006.

R.K. Srivastava, B.R. Steunebrink, and J. Schmidhuber. First experiments with powerplay. *Neural Networks*, 41(0):130 – 136, 2013. ISSN 0893-6080. doi: http://dx.doi.org/10.1016/j.neunet.2013.01.022. URL `http://www.sciencedirect.com/science/article/pii/S0893608013000373`. <ce:title>Special Issue on Autonomous Learning</ce:title>.

O. Stasse, A. Escande, N. Mansard, S. Miossec, P. Evrard, and A. Kheddar. Real-time (self)-collision avoidance task on a hrp-2 humanoid robot. In *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, pages 3200–3205. IEEE, 2008.

M. Stollenga, L. Pape, M. Frank, J. Leitner, A. Förster, and J. Schmidhuber. Task-relevant roadmaps: A framework for humanoid motion planning. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2013.

J. Storck, S. Hochreiter, and J. Schmidhuber. Reinforcement driven information acquisition in non-deterministic environments. In *Proceedings of the International Conference on Artificial Neural Networks, Paris*, volume 2, pages 159–164. Citeseer, 1995.

H. Sugiura, M. Gienger, H. Janssen, and C. Goerick. Real-time collision avoidance with whole body motion control for humanoid robots. In *Intelligent Robots and Systems, IEEE/RSJ International Conference on*, pages 2053–2058. IEEE, 2007.

G. Sun and B. Scassellati. Reaching through learned forward models. In *IEEE-RAS/RSJ Int. Conf. Humanoid Robots (Humanoids 2004)*, pages xx–yy, 2004. accepted for publication.

R.S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3: 9–44, 1988.

R.S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proc. of the International Conference on Machine Learning*, pages 216–224, 1990.

R.S. Sutton and A.G. Barto. *Reinforcement learning: An introduction*, volume 1. Cambridge Univ Press, 1998.

R.S. Sutton, D. Precup, S. Singh, et al. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1):181–211, 1999.

R.S. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Proc. of Advances in Neural Information Processing Systems (NIPS)*, 2000.

S. Takagi. Toyota partner robots. *Journal of the Robotics Society of Japan*, 24(2):62, 2006.

The Boost Graph Library. Bgl, the boost graph library. url=http://www.boost.org/libs/graph/, accessed 2012.

The CGAL Project. Cgal, the computational geometry algorithms library. url=http://www.cgal.org/, accessed 2012.

E. Thelen, D. Corbetta, K. Kamm, J. P. Spencer, K. Schneider, and R. F. Zernicke. The transition to reaching: mapping intention and intrinsic dynamics. *Child Development*, 64(4):1058–1098, 1993.

D. Tolani, A. Goswami, and N.I. Badler. Real-time inverse kinematics techniques for anthropomorphic limbs. *Graphical models*, 62(5):353–388, 2000.

G. van den Bergen. *Collision detection in interactive 3D environments*. Morgan Kaufmann, 2004. ISBN 155860801X.

C. von Hoftsen. An action perspective on motor development. *TICS*, 8(6):266–272, 2004.

C.W. Wampler. Manipulator inverse kinematic solutions based on vector formulations and damped least-squares methods. *IEEE Transactions on Systems, Man and Cybernetics*, 16(1): 93–101, 1986.

L. Wang and C. Chen. A combined optimization method for solving the inverse kinematics problems of mechanical manipulators. *Robotics and Automation, IEEE Transactions on*, 7(4): 489–499, 1991.

C.J.C.H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, 1989.

C.J.C.H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992.

J. Weng, J. McClelland, A. Pentland, O. Sporns, I. Stockman, M. Sur, and E. Thelen. Autonomous mental development by robots and animals. *Science*, 291(5504):599–600, 2001.

D.E. Whitney. Resolved motion rate control of manipulators and human prostheses. *IEEE Transactions on man-machine systems*, 1969.

D. Wierstra, A. Förster, J. Peters, and J. Schmidhuber. Recurrent policy gradients. *Logic Journal of IGPL*, 18(2):620–634, 2010.

R.J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(23), 1992.

W.A. Wolovich and H. Elliott. A computational technique for inverse kinematics. In *The 23rd IEEE Conference on Decision and Control*, volume 23, pages 1359–1363. IEEE, 1984.

J. Zhao and N.I. Badler. Inverse kinematics positioning using nonlinear programming for highly articulated figures. *ACM Transactions on Graphics (TOG)*, 13(4):313–336, 1994.

M.A. Zohdy, M.S. Fadali, and N.K. Loh. Robust control of robotic manipulators. In *American Control Conference*, pages 999–1004. IEEE, 1989.