8-2013

Gesture-Based Robot Path Shaping

Paul Yanik
Clemson University, pyanik@wcu.edu

Gesture-Based Robot Path Shaping

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Engineering

---

by
Paul M. Yanik
August 2013

Accepted by:
Dr. Ian D. Walker, Committee Chair
Dr. Keith E. Green
Dr. Richard E. Groff
Dr. Richard R. Brooks

Abstract

For many individuals, aging is frequently associated with diminished mobility and dexterity. Such decreases may be accompanied by a loss of independence, increased burden to caregivers, or institutionalization. It is foreseen that the ability to retain independence and quality of life as one ages will increasingly depend on environmental sensing and robotics which facilitate *aging in place*. The development of ubiquitous sensing strategies in the home underpins the promise of adaptive services, assistive robotics, and architectural design which would support a person's ability to live independently as they age. Instrumentation (sensors and processing) which is capable of recognizing the actions and behavioral patterns of an individual is key to the effective component design in these areas.

Recognition of user activity and the inference of user intention may be used to inform the action plans of support systems and service robotics within the environment. Automated activity recognition involves detection of events in a sensor data stream, conversion to a compact format, and classification as one of a known set of actions. Once classified, an action may be used to elicit a specific response from those systems designed to provide support to the user. It is this response that is the ultimate use of recognized activity. Hence, the activity may be considered as a command to the system. Extending this concept, a set of distinct activities in the form of hand and arm gestures may form the basis of a command interface for human-robot

interaction. A gesture-based interface of this type promises an intuitive method for accessing computing and other assistive resources so as to promote rapid adoption by elderly, impaired, or otherwise unskilled users.

This thesis includes a thorough survey of relevant work in the area of machine learning for activity and gesture recognition. Previous approaches are compared for their relative benefits and limitations. A novel approach is presented which utilizes user-generated feedback to rate the desirability of a robotic response to gesture. Poorly rated responses are altered so as to elicit improved ratings on subsequent observations. In this way, responses are honed toward increasing effectiveness. A clustering method based on the Growing Neural Gas (GNG) algorithm is used to create a topological map of reference nodes representing input gesture types. It is shown that learning of desired responses to gesture may be accelerated by exploiting well-rewarded actions associated with reference nodes in a local neighborhood of the growing neural gas topology. Significant variation in the user's performance of gestures is interpreted as a new gesture for which the system must learn a desired response. A method for allowing the system to learn new gestures while retaining past training is also proposed and shown to be effective.

Dedication

To Wesley

Acknowledgments

The amalgam of people, events, and experiences which has lent influence to the undertaking and completion of this work is richly complex. A complete listing of those who have guided me along the way is not possible. However, it is with great thanks that I acknowledge those whose encouragement, support, instruction and collaboration have principally allowed me to complete this journey.

Foremost, I wish to thank my advisor, Ian Walker. Dr. Walker was the first person I encountered at Clemson University when I initially sat in a lecture hall in fall of 2007. That day, and in every meeting with him since, he has been a model of unflagging support and commitment to my progress toward this degree while allowing me utmost freedom to pursue my areas of interest. In our many conversations, he has always sought to listen and understand before offering what is routinely advice of pivotal importance. He has also served as a mentor to me through his manner of teaching and working with students. As I have worked as a university instructor throughout the course of my degree, his example has helped shape my own teaching style, and my facility with student interaction. It is my hope that as I continue to teach and advise students, that I will carry forward at least a measure of what he has imparted to me.

I would like to thank the members of my committee (past and present): Bob Schalkoff, Stan Birchfield, Richard Groff, Richard Brooks, and especially Keith Green.

His vision for the ART project and its integration of gesture has informed my work at every step. My discussions with these gentlemen were often the source of critical advice which helped me to consider many challenges with fresh eyes. Their suggestions allowed me to advance my research and to improve my thesis. I also wish to thank Johnell Brooks for her support of my research and for guiding me through the human aspects of my experimentation.

Thanks also to my friends and collaborators in the robotics lab including Apoorva Kapadia, Joe Manganelli, Tony Threatt, Jessica Merino, Linnea Smolentzov, Henrique Houayek, Tarek Mohktar, Ninad Pradhan, Bryan Willimon and Bryan Peasley. Through many long work sessions and great conversations, their friendship and gifted insight have shaped my thinking and research and helped me over the many hurdles endemic to being a graduate student. They always made the lab an enjoyable place to be. It has been a privilege to know and work with you all.

Completing an endeavor such as this while holding down family and work responsibilities has indeed been challenging. I could not have done so without the support of my colleagues in the Department of Engineering and Technology at Western Carolina University. Everyone single person in this group has contributed to my efforts through their advice, tutelage, loans of books, juggling of my schedule, and continuous moral support. Though I cannot name everyone here, special thanks goes to Ken Burbank, James Zhang, Brian Howell, Chip Ferguson and Bob Adams. These gentlemen were instrumental in making WCU a place where I could grow as both a teacher and a researcher.

I wish to thank my parents, Albert and Margaret Yanik, and my brothers and sisters who knew this was possible even when I wasn't so sure. They never failed to inquire of my progress, to express their pride in my efforts, and to encourage me along. Their support meant more than they could know. Thanks also to Bob Satterwhite

and Dallas Satterwhite who came to the rescue at home so many times and stepped into the void when I simply couldn't keep up.

Finally, I wish to thank my wife Wesley and my daughters Owen and Barrett. Wesley shared my vision to complete this work and made it *our* dream. She accepted the long hours in the office and lab with grace and was always ready to listen to my half-baked ideas, my worries, and my aspirations. I could not have attempted this without you. Owen and Barrett never let me depart or arrive home without hugs and kisses; they decorate my work space with cardboard robots and drawings designed to help me with my *homework*. The three of you are the best part of my life, the stars in my heaven.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Mobility decreases as we age. Such reduction, whether gradual or sudden, may ultimately impair one's ability to perform essential Activities of Daily Living (ADLs). A reduced capacity to conduct ADLs may be associated with diminished quality of life, decreased independence, a higher burden to caregivers, or even institutionalization [25]. Thus, for older adults with the goal of *aging in place*, it is vital to ensure that they have the ability to perform ADLs independently [109].

As the population of the United States ages, their desire to retain a level of independence in the face of diminished mobility and health will increasingly draw upon assistive technologies to facilitate essential ADLs. The work described in this thesis is motivated by a dearth of technologies that might provide adequate support of these important functions. Effective design, deployment, and use of such technologies are seen as critical to promoting an improved quality of life and prolonged independence for the user. The *Assistive Robotic Table* (ART) project begun at Clemson University seeks to develop an intelligent class of assistive devices and services which are highly integrated into the built environment. In so doing, the environment becomes an adaptive partner to facilitate aging in place for users whose ability levels

are changing.

1.1 Research Contributions

In this thesis, a contribution is made in the area of Human-Robot Interaction (HRI) which aims to address common limitations of interface designs in this space, particularly as applied to assistive devices for aging in place. To this end, the HRI problem is dissected into its constituent parts and a thorough literature review is conducted which examines the relative strengths and weaknesses of current approaches in the major problem areas. Extending the state of the art, experimentation to develop a novel and effective end-to-end interface framework for HRI is presented and discussed.

Toward the creation of such an interface, gesture at the scale of hand/arm gesticulation is explored as a candidate mode of interaction. Given the importance of gesture at this scale as a means of human communication (section 1.3), research in this area promises an intuitive user experience that caters to impaired or otherwise unskilled individuals, a target population for technologies designed to facilitate aging in place. With this overarching goal in mind, key features of the interface include the following.

- Sensing must be ambient and non-intrusive. Sensor types which encumber the user to carry or wear specialized sensing devices or apparel are avoided. Vision-based sensors (i.e. cameras) which expose the user to potential loss of privacy are also avoided.

- Gestures are freely formed according to the user's own movement style and capability. Recognition of gesture should not require the user to match a specified

gesture choreography. Thus, the system will make use of human instruction on line to acquire gesture within a manageable number of observations and to learn the desired response.

- The gesture-based vocabulary of the interface shall be extensible on line. The user may *teach* new gestured commands at any time.

Successful application of the work described in this thesis is envisioned to form the basis of a comprehensive system of adaptive robotic and architectural components to support independent living for individuals whose capabilities and needs are changing over potentially long periods of time.

1.2 Thesis Layout

The remainder of this chapter contains a review of relevant literature. The review can be roughly divided into two main categories. First, background information is given on the broader areas of architectural robotics (section 1.3.1) and HRI (section 1.3.2) which motivate the ART project and which provide the vision underpinning the work presented in this thesis. Second, the problems of gesture recognition and the mapping of gesture to a robotic response are examined. In considering these problems, related work on the topics of sensor device types, data representation, pattern recognition, and machine learning is discussed.

The remainder of this section contains a brief synopsis of experiments undertaken in pursuit of the research goals stated above. Chapters 2 through 4 contain detailed descriptions of these experiments and their results. Chapter 5 contains final conclusions and prospects for future work in this area.

1.2.1 Activity Recognition

Chapter 2 describes a study which is focused on classification of representative motions related to manual dexterity. Specifically, the work seeks to determine whether a non-vision based sensor paradigm could be used to construct a useful classifier of human motion based on a holistic (pixel-based) representation. A comparison is made between the efficacy of non-vision based and vision based (i.e. cameras) sensors. A data representation based on self similarity of raw sensor readings (or image pixels) is used. Results of this study informed the selection of sensor platform, data representation, and comparison method used in subsequent work.

1.2.2 Gesture Recognition

In chapter 3, the broader problem of generalized human activity recognition is narrowed to that of gesture recognition and mapping of gesture to actuated robot response. Specific methods for arm-scale gesture sensing, representation, pattern recognition are selected for their respective strengths in the common facets of the gesture recognition problem. Further, the use of Q-Learning is explored as a means of avoiding strict classification of gesture in favor of gesture learning as defined by expected rewards from human user. The chapter lays out an end-to-end framework for on line definition of a gesture based command vocabulary with the user as teacher in an HRI context.

1.2.3 Gesture Vocabulary Augmentation

Chapter 4 considers the problem of adding new gestures on line. It is assumed that, in practice, the ART appliance will come equipped with a baseline level of learning which includes a knowledge of common essential gesture commands. However,

building on this will require the user to train ART in real time. In this case, concerns emerge regarding the number of iterations which the user is required to execute, and the stability of past learning as new gestures are encountered (the *Stability-Plasticity Dilemma* [22]). These concerns are central to the practical deployment of ART. The chapter presents findings that support the design decisions of chapter 3 and which show promise for the eventual incorporation of ART in a real world setting.

1.3 Related Work

This section contains a review of related literature in two major categories. Sections 1.3.1 and 1.3.2 contain background information on the broader areas of architectural robotics and HRI. Concepts in these areas form the basis for the broader ART application and motivate the work described in this thesis. Section 1.3.3 discusses past work directly related to recognition of human which informs the experimentation of chapters 2 and later.

1.3.1 Architectural Robotics

Heretofore, architects and environmental designers have attempted to accommodate those with reduced mobility or physical impairment through the use of Universal Design Principles (UDPs) and *smart home* technologies. UDPs help to ensure that the environment does not confound an individual's efforts to complete tasks by making the environment safe, clean, legible and barrier-free for all occupants regardless of ability [38, 52, 82]. These strategies facilitate resident mobility and independence. However, the majority of current implementations are *static* and of low fidelity. Most often, user accommodation is delivered solely in the form of systems associated with the built environment such as the placement of furniture and fixtures.

Smart homes aim to extend awareness, increase control over systems, and enhance the security, healthfulness and safety of the environment through sensing, inference, communication technologies, decision-making algorithms and appliance control [24, 31, 51, 61]. However, the real-time processing of occupant activity via high fidelity implementations has historically been costly in terms of computing and memory requirements and has often relied on technologies considered intrusive of people's privacy (e.g. cameras) or which represent an encumbrance to the user such as with wearable devices. As a result, smart homes have frequently utilized low fidelity implementations. Practical occupant sensing in smart homes is typically event-based, consisting of on/off sensor activations such as room changes, door openings/closings, appliance actuations, etc.

Advances in computing speed and storage make possible implementations of increasing capability. A logical progression for the use of high fidelity sensing and processing of user activity may be seen in their central importance to assistive robotics. As Green and Walker describe [105], the notion of assistive robotics frequently conjures images of a self-contained *humanoid servant* in which all robotic and intelligence challenges have been addressed. Finding this to be an unlikely possibility in the near term while seeking to move beyond the conventional static smart home, this research envisions an environment containing robotic components that take advantage of the capabilities and higher level thinking of the user to operate in a collaborative manner, working *with* rather than *for* the user.

With the support of clinicians and staff of the Roger C. Peace Rehabilitation Hospital of the Greenville Hospital System University Medical Center (GHS), previous investigations underpinning the ART project have been performed [18, 19, 97, 96, 104] at the hospital's *home+* lab (Figure 1.1a). This work has examined possible forms and use models for assistive robotics in home and hospital settings. The vision sketched

from these investigations is shown in Figure 1.1b and includes intelligent appliances such as an enhanced over-the-bed table, and an automated assistant/storage and retrieval system for personal items, and continuum robotic morphing surfaces [57, 74] capable of providing therapy and comfort to the user.

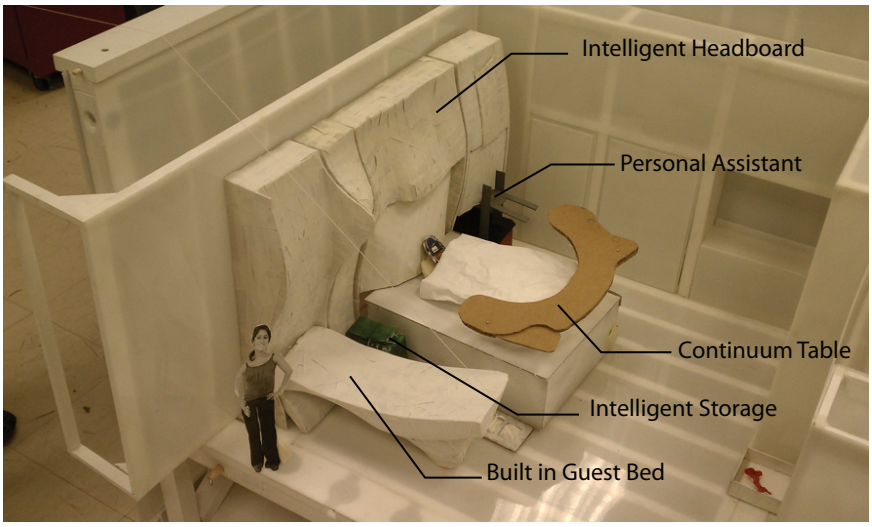
ART and the larger home+ architectural robotic ecosystem aim to improve recovery outcomes and quality of life for elders aging in place through development of collaborative interaction with the home environment. Current work focuses on ART's ability to facilitate a natural, companionable and social relationship between human and robot [70, 95] through a system of user-defined behavioral rewards to the machine. To this end, the use of high fidelity sensing to create a non-verbal communication loop (Figure 1.2) between a patient and ART is under way. One aspect of this communication loop, the use of gestured command, is a primary focus of this thesis.

The high fidelity sensing employed in efforts such as these is expected to allow for learned inference of user action and intention through persistent monitoring. Further, degradation in the abilities of the user may be tracked over time so as to adaptively inform the robot's assistive action plans. With knowledge of typical user activity patterns the environment could respond to gestured commands or detect infrequent needs such as assistance with reach, weight transference, or ambulation [118]. Effectively implemented, assistive robotic components would facilitate the afore mentioned aims of smart home technologies.

1.3.2 Human-Robot Interaction

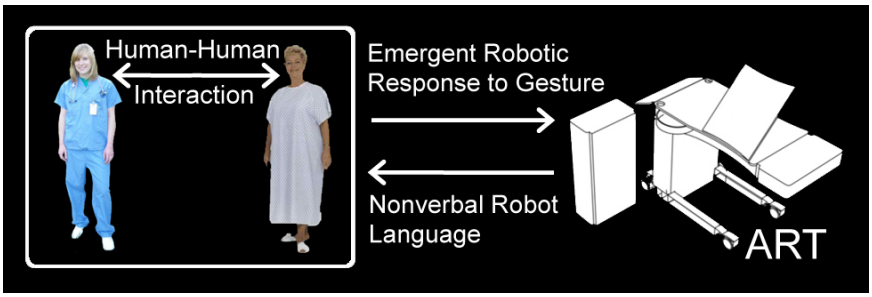
As ART seeks to improve the quality of living for its human users, the conventional view of service robotics as merely labor saving devices must be reconsidered.

(a)

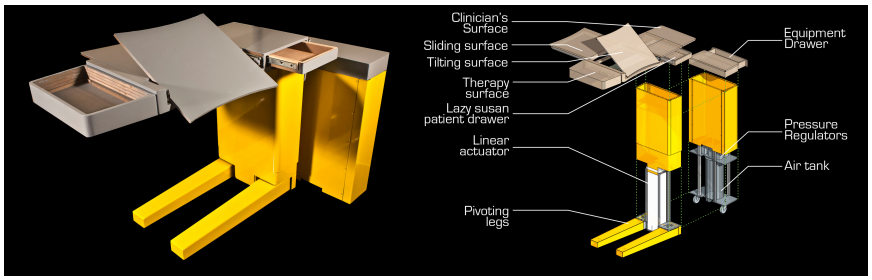


(b)

Figure 1.1: (a) The home+ lab. (b) The envisioned robotic ecosystem.



(a)



(b)

Figure 1.2: (a) The non-verbal communication loop of the *Assistive Robotic Table* being developed at Clemson University. The focus of this work is on the emergent (learned) response of this device to the user. (b) A recent project artifact.

In the vision which motivates this thesis, architecural robotics engages the resident as a fellow agent; a part of the complete environment in which the overall healthfulness, productivity, security, and enjoyment of the living space are the results of a novel collaborative and adaptive partnership between the robotic components and users. To this end, the underlying intelligence of ART may seek to optimize the Human-Robot Interaction (HRI) partnership by not only assessing typical measures of success (e.g. task completion), but also through learned accommodation of the needs, preferences, and constraints of the user.

Research suggests that when considering the possibility of an in-home service robot, most people do not expect nor desire that such devices will fulfill human capacities [29, 93]. That is, robots need not physically resemble humans nor possess virtues best ascribed to humans such as creative thinking, judgment, or friendship. Moreover, what is desired is for robots to assume an assistive role, particularly where there may be gaps in human ability such as with memory intensive or perceptual tasks.

1.3.2.1 Baseline Capabilities

To accomplish such tasks and to facilitate higher level functions, ART will possess a framework of baseline intelligent faculties. Navigation, perception, management of system resources, object manipulation, and social considerations [90] are capabilities which must be present prior to any attempts at learned adaptation to a user.

Navigation encompasses collision free movement as well as social aspects of motion planning. Trajectories must be smooth, understandable, natural [4], and adaptive [11] to the human resident. Movements must not occur too quickly so as to induce surprise nor may they pass uncomfortably close to the user. The robot may

not violate conventional human expectations for timing and sensibility [1, 36]. Perception may utilize conventional pervasive sensing techniques to localize both humans and robotic components. Advanced implementations should utilize collected sensor data to interpret the context of human activities and of the environment [46, 49]. Robotic components should also manage resources (both internally and within the environment) through real time situational awareness. The respective capabilities of all agents can thus be brought to bear in collaborative tasks. Object manipulation is a conventional task asked of robotic agents. Taxomonies of manipulation include tasks typically associated with the hand such as grasping, pushing, and fetching/carrying [90]. Initially, this is expected to occur in the form of intelligent storage as described described in section 1.3.1 and shown in Figure 1.1.

Implementation of these baseline capabilities will utilize conventional approaches for low level sensing of room fixture usage (appliances, cabinetry), localization of the human user, robotic components and selected objects, and gathering of performance data. Upon them, higher level inference and context awareness will be built as briefly described in section 1.3.2.2 below.

1.3.2.2 Higher Level Functions

Once implemented the capabilities described above may be used to execute higher level functions involving the estimation of human intentions so that human and robotic agents may collaborate [8] to complete tasks at hand. Through the course of such interactions, the environment might learn the activity patterns, preferences and limitations of the human resident. This will determine when behavioral anomalies represent rare occurrences or long term behavioral changes and adapt accordingly.

In order for the proposed adaptive capabilities of ART to best achieve the objective of creating an optimal living environment, performance metrics would be

developed on which to base a paradigm of reinforcement learning [92]. Typically, the evaluation of human-robot collaboration is reduced to task-oriented effectiveness metrics such as time per task, error/damage/collision counts, task completion coverage, or situational completion quality [26, 80].

Generalized assessment of the efficacy of the human-robot relationship may be determined through the common metrics of neglect tolerance and interface efficiency. Neglect tolerance is a measure of the robot's ability to remain on task over time despite a lack of human intervention. Indirectly, this is a measure of both user trust, and the inherent intelligence of the robot. Interface efficiency is a measure of the effort required for the human to gain situational awareness, decide on a course of action, and command the robot. Indeed, since users will be primarily untrained, elderly, and possibly disabled, the need for a clean and easily understood interface will be key to efficient human-robot performance [98]. Although not a primary focus of our research, neglect tolerance will serve as a fundamental bellwether of our progress in improving system intelligence. Interface efficiency is a primary consideration of this thesis. Indeed, the development of an intuitive user experience with a small number of training episodes (in a machine learning context) is the focus of experimentation described in chapters 3 and 4.

1.3.2.3 Broader Impacts

Ultimately, the goals of ART and of architectural robotics as described in section 1.3.1 will require a new class of innovative performance metrics to characterize the overall quality of the environment in areas such as healthfulness, conduciveness to creativity, and support of social interaction. These metrics, along with conventional environmental sensor data, may inform a higher level of the reinforcement learning infrastructure. Use of such metrics may appear on the surface to be counter to

conventional success criteria. For example, a robot which *engages* and assists the human agent to undertake partial performance of tasks for themselves during a period of impaired mobility might reduce the healing time of the user although it may also slow the average execution time per task. Hence, the ultimate goal of such a system can be viewed as the improvement of human performance rather than simply that of increased usability of the robotic components [20].

1.3.3 Human Gesture Recognition

Automated recognition of human gesture is an active area of research. Work in this space finds application in areas such as entertainment, healthcare, security, and comfort. Gesture may occur in various forms. These may include hand and arm gesticulation, pantomime, sign language, static poses of the hand and body, or language-like gestures which may replace words during speech. Of these, hand and arm gesticulation account for 90% of gestured communication [76]. With the goal of creating an intuitive interaction paradigm between humans and robotic or computing agents, exploration of gesture at the scale of hand/arm gesticulation is warranted.

Efforts at automated gesture recognition generally involve a common set of considerations and problems to be addressed. These include some combination of sensor platform, data representation, pattern recognition and machine learning. This section discusses previous approaches to these problems and their relative benefits and drawbacks compared to the methods applied in the experimentation of chapters 2 through 4.

1.3.3.1 Sensing

In order for gestures to be detected and classified, the motion or pose of the actor must be sensed. Typical sensor strategies include wearable devices such as data gloves or body suits which are instrumented with magnetic field tracking devices or accelerometers, or vision-based techniques involving one or more cameras [76]. Still other approaches involve Infra Red (IR) motion or proximity sensors. In this section, examples of prevalent sensor types and applications of each are described along their relative strengths and drawbacks.

Wearable devices provide almost immediate detection of gesture motion without the need for image preprocessing. Further, such devices are not susceptible to visual occulusion as are cameras or other *line of sight* platforms [12]. Jin et al. [54] use a glove-based orientation sensor to extract static hand positions to be used as commands. Lementec and Bajcsy [69] use wearable (arm) orientation sensors for sensing arm gesture models composed of Euler angles. These are intended for use in an Unmanned Aerial Vehicle (UAV) and implemented as a lab simulation. Zhou et al. [121] use Micro-Electro-Mechanical System (MEMS) accelerometer data to characterize hand motions including *up*, *down*, *left*, *right*, *tick*, *circle* and *cross*. Yan, et al. [114] uses a *shape tape* system consisting of bend sensing optical fibers and orientation sensors to extract the 3D orientation of various points on an actor's arms and torso. Wearable sensors are also used in [113, 121, 122], and others. Typically, however, the usefulness of wearable devices for measuring gestured motion is accompanied by the acknowledgment that such devices may limit user motion and often require a wired connection to a computer. Thus, they present inherent impediments to practical application [76].

IR proximity sensors are used by Cheng et al. [23] to create a reliable gesture

recognition system for a touchless mobile device interface. The method uses the pair-wise time delay between a passing user's hand and two IR proximity sensors. This system detects gestures of *swipe right*, *swipe left*, *push* and *pull*. Rhy et al. [32] propose a computer control interface design using a proximity sensor to extract hand commands to a GUI. The mechanism is scaled as a mouse replacement. Such coarse assessment of motion is not sufficiently descriptive to support an extensive vocabulary of gestures. However, as shown in [117, 118] and discussed in chapter 2, an array of IR motion sensors can provide sufficiently rich data to allow for accurate classification of gross motions.

Much of the work in gesture recognition is performed using video image sequences due to the richness of information and cost effectiveness available with cameras. A recent thorough discussion of vision-based and other sensor types for the purpose of gesture recognition is given by [12]. Vision based approaches may suffer from disadvantages associated with latency, occlusion, or lighting. Further, since most video sequences represent a 3D to 2D projection, a loss of information is inherent in the processing of data [76]. And, although the presence of cameras in an individual's personal environment is becoming more common, they are often considered intrusive of privacy in certain scenarios [9, 30].

With the limitations of these various sensor types in mind, the experimentation described in chapter 3 utilizes an RGB-D depth sensing system. The Microsoft Kinect [75, 83] is a current model RGB-D sensor (Figure 3.2a) which uses *near-IR* technology to project an IR light pattern on to the subject. The projection produces a dense *point cloud* of sensor readings which is used to construct a depth image [12]. In this way, the Kinect provides a rich, real-time, 3D data stream that preserves user anonymity. Although the Kinect does provides a conventional RGB camera output, it is not used in this research. As Figure 3.2a shows, this output is physically covered during data

collection to ensure that the data stream does not contain information that identifies the user. The Kinect also functions well in across varied lighting conditions (including total darkness) where conventional cameras would be ineffective. For these qualities, the Kinect RGB-D sensor platform is selected for the experimentation described in chapter 3 and later.

1.3.3.2 Data Representation

Given a sensor input data stream, a compact data representation must be computed. Representations may be roughly divided into feature-based (parametric) versus holistic (nonparametric) forms. Parametric representations extract features related to the physical geometry and kinematics of the actor such as limb lengths and joint angles. Spatial information about the actor's performance of the gesture is preserved.

Holistic representations utilize statistics of the motion performed that are drawn from the sensor signal (typically in (x, y, t) space). Hence, with regard to the frequently employed visual images of motion, these can also be characterized as pixel-based representations [10]. Whether parametric or holistic, however, the problem of data representation can, in general, be defined as one of feature selection. That is, some vector of characterizing numerical features is extracted from sensor data and applied to a classifier.

Motion History Images (MHI) have been used to form a visual template of motion that preserves directional information [15, 16, 59]. Histograms of Oriented Gradients (HOGs) are used in [28] to generate regional descriptors of single frame images for human detection. Periodic motions such as walking or running may be recognizable solely from the movement of lighted feature points placed on the actor's body [55]. This phenomenon is exploited by Benabdelkader et al. [10] and Cutler

and Davis [27] through the concept of self-similarity. In this approach, the locations of features (e.g. edges) in an image sequence are seen to generate a repeating pattern from which a motion descriptor may be generated. The set of features is tracked through the course of an image sequence. The summed distances of features between image pairs is computed. Performing this summation exhaustively across all image pairs forms a Self-Similarity Matrix (SSM).

SSMs and HOGs are combined to produce view-invariant representations for non-periodic motions in [56] and [118]. A detailed description of this approach is given in chapter 2. Experimental results show that recurrences in both IR spatial sensor data and in video data can produce robust discriminants. Although these representations possess strong discriminative qualities, they tend to be of high dimension and require either compression or excessive computation.

In the experimentation described in chapters 3 and 4, the concept of Dynamic Instants (DIs) advanced by Rao et al. [84] is extended to three dimensions. DIs are defined as the extrema (or discontinuities) of acceleration in an actor's motion. They include motion starts, stops, and rapid changes in speed or direction. Rao shows this representation to be view-invariant, with DI features being visible (unless occluded) regardless of the vantage point of the sensor. The Kinect (section 1.3.3.1) allows direct extraction of a third dimension without the need for 2D image processing. The representation used for gesture recognition in this research combines the five most significant DIs in (x, y, z) space along with their frame number over a 5 second interval at 30 Hz sampling. This is described further in section 3.2.

1.3.3.3 Pattern Recognition

In order to classify gestures, the feature vector is typically sorted into one of a known gallery of types. Numerous classification methods have been introduced

including Hidden Markov Models (HMM), Finite State Machines (FSM), clustering techniques such as Nearest Neighbor (kNN) and C-means, and various types of artificial neural networks including Multilayer Perceptron (MLP) networks, Time Delay Neural Networks (TDNN) [76], neural networks based on Adaptive Resonance Theory (ART) [44], Neural Gas (NG) [71], and Growing Neural Gas (GNG) [39].

Hidden Markov models have well established success in the classification of gestures and of generalized motion and are used in numerous research efforts. Notably, these include [110] and [112]. A survey of such approaches can be found in [77]. The authors note that HMM approaches may inaccurately assume that observation parameters may be approximated by a mixture of Gaussian densities. Further, HMMs often have poorer discriminative outcomes than neural networks.

Bobick and Wilson [17] use finite state machines to classify gestures collected from video images. Lee et al. [68] seek to classify video motion sequences as whole-body gestures by mapping sequences of estimated poses to gestures. PCA is used for visualization; an EM-based (Expected Maximum) Gaussian Mixture Model is used for clustering of poses. Frolova et al. [41] classify planar decimal digits traced in free air with high accuracy by storing hand trajectories. The Most Probable Longest Common Subsequence (MPLCS) algorithm is used to classify trajectories by comparison with a probabilistic template based on variations within a Gaussian Mixture Model. Prasad and Nandi [81] explore the effectiveness of several methods for vectorizing and clustering gesture motion data including: hierarchical, mean shift, k-means, fuzzy c-means and Gaussian mixture. Schlömer et al. [89] use k-means to determine clusters in basic hand/arm gestures generated using a wiimote controller including *square*, *circle*, *roll*, *Z*, and *tennis swing*. Wachs et al. [103] use fuzzy C-means clustering to achieve highly accurate recognition of twelve static hand gestures as the basis for a telerobotic command interface. And, although the focus of Knox's

work in [63] is user-guided machine learning (see section 1.3.3.4) the author uses kNN to determine the probable state of a robot from sensor data in order to conduct state-action selection as the basis for a user reward function. The experimentation described in this research is compared with a kNN approach. Unlike Knox, however, we assume that the sensed gesture is the state, rather than the sensed position of the robot.

Zhu and Sheng [122] use wearable sensors to detect both hand gestures and simple ADLs. Neural networks are used for gesture spotting. HMMs are used for classification. Varkonyi-Koczy and Tusor [102] use Circular Fuzzy Neural Networks (CFNN) to classify static hand postures for their iSpace intelligent environment. CFNNs are seen to have reduced training time. Sequences of hand postures are composed into hand gestures. Yang and Ahuja [115] use Time Delay Neural Networks (TDNN) to classify sequences of motion trajectories in hand motion for American Sign Language (ASL). Conventional neural networks are used in [78] and [86] for their ability to generate responses in real time while also being robust in the presence of temporally inconsistent input patterns. Alexander et al. [2] use a neural network based on Adaptive Resonance Theory to recognize static hand gestures. Networks employing Adaptive Resonance Theory are seen to possess the ability to learn incrementally, thus making them effective in online learning.

Stergiopoulou and Papamarkos [91] use GNG to model the topology of the hand itself (rather than more abstract features of the scene) in various finger-extended postures. Skin color is used as the dominant feature. From this, finger directions are extracted based on the centroid of the palm. Classification is accomplished using Gaussian probability of finger angles. Angelopoulou et al. [5] present a probabilistic growing neural gas (A-GNG) method for tracking the topology of the human hand as it progresses through various gestures. A-GNG offers improved topology mapping

to the basic GNG algorithm. However, the approach is chiefly video based and forms the GNG codebook vectors based on the appearance of the hand rather than on any of the movement characteristics of the action. In this way, the method is mainly that of a static analysis of hand shape.

The GNG algorithm [39] is a variant of the self-organizing feature map. Because it is capable of tracking a moving distribution [53], adding new reference nodes, and operating from static input parameters, it is well suited to the task of gesture recognition where no labelled data is available. Indeed, since the acquisition of gesture data is often expensive in terms of the effort and time required of both the user and the researcher, such a technique which learns online is particularly desirable. Further, its ability to grow and alter its topology over time suggests that it may be effective in learning new gestures as they are observed. For these reasons, GNG is the clustering method explored in this paper.

1.3.3.4 Machine Learning

Although techniques described in subsection 1.3.3.3 may be broadly categorized as machine learning methods, the term as it is used in this thesis refers to a mechanism by which some manner of feedback is used to improve future outcomes of a robot's assistive behavior. Typically, such a mechanism implies the use of training data to refine a classifier of choice off line as with conventional neural networks. However, a goal of this research is to create an online learning modality that utilizes direct interaction with the user so that a robot agent converges upon a desirable configuration. Hence, our goal is to iteratively create a direct mapping between sensed gestures and inferred goals.

Such *sensorimotor* mappings of sensor input to robot motor commands have been successfully used in several applications. Ritter et al. [85] and Martinetz et

al. [72] showed that Self-Organizing Feature Maps (SOFM) [66] could be used to discretize input space into receptive fields associated with individual neurons. Each node in the network then uses an error correction rule to learn an output composed of a vector of joint angles and a Jacobian to effect a desired robot configuration. In this way, the SOFM is capable of a nonlinear mapping between input and output spaces. The topology preserving nature of the SOFM allows for faster learning than conventional neural networks by taking advantage of the idea that similar inputs should yield similar outputs. Hence, topological neighbors will encode similar sensor inputs and thus, they can be made to learn desired outputs as a group. Walter and Schulten [106] use a Neural Gas (NG) mapping [71] and apply a Gaussian neighborhood function to soften learning across the discretized input space of nodes to produce smoother output control. Gross et al. [43] use NG to map neighborhoods of sensory input (locations in a maze) to motor outputs of *forward*, *backward*, *left* and *right* commands to a mobile robot. The authors use Q-Learning (described below) to develop an optimal command policy for moving straight and forward for the longest intervals possible. A good survey of these and related applications can be found in [7].

Reinforcement Learning approaches (RL) are frequently applied to the control of robots. Unlike supervised learning approaches which require a set of training data with desired output values, an agent (robot) in an RL framework senses its environment and operates under some policy so as to maximize the expected future returns (evaluations) it will receive though a scalar reward signal. RL techniques use Markov Decision Processes (MDPs) to refine a mapping between an agent's state and its future actions. Over successive iterations of input, action, and evaluation, a policy for maximizing the sum of future (discounted) reward is learned which, in the limit, can be seen to approach optimality [92]. Arguably, the most popular RL technique

is that of Q-Learning [107] for its simplicity and for its lack of need to model the environment. We apply Q-Learning in this work as described in section 3.4.1.

Within an RL framework, an agent in a particular state s of the environment, an action a is selected based on the highest available expected return (or Q *value*). The policy may be periodically modified to allow for exploration of the action space. Following each episode of state-action sequences toward a known goal, the policy is evaluated and a table of state-action pairs is updated to reflect the actual realized returns (which may be expected in future episodes under a given policy). Typically, convergence to an optimal policy requires a large number of iterations during a training phase. In the field of robotics, this is generally impractical to achieve given the potentially large number of state-action pairs coupled with the mechanical limitations of execution speed, reliability and energy consumption of a robotic agent. Hence, generalization of actions across similar states is critical [99].

Touzet [100] presents a method for generalization among state-action pairs in a Q-Learning framework using Kohonen's self-organizing map (*Q-Kohon*). As previously mentioned, the SOFM's topology preserving structure allows for *neighborhood learning*. Hence it applies well to the Q-Learning approach which underlies Touzet's method. Q-Kohon uses the SOFM as an associative memory. Each node stores a tuple consisting of its state label (or *situation* in Touzet's terminology), an action, and a Q value. The input situation probes the map for the nearest state label having a positive Q value. The neighborhood actions are updated according to the reward received from taking the action associated with that node.

The approach used in the experiment of chapter 3 is an adaption of Q-Kohon. As previously stated, the GNG algorithm is employed so as to avoid extensive parameter tuning. Also, the capability of the GNG topology to add nodes in the presence of new gesture forms or significant distribution error is seen as key. However, the

strengths of the SOFM paradigm remain available.

Usually, reinforcement learning utilizes an automated, internally generated reward function. As previously mentioned, the number of trials required to learn an optimal policy in this case is typically large. Further, the reward function is typically sparse in nature. For example, Tesauro's implementation of an automated backgammon player taught by reinforcement learning assigned a reward of 1 (one) for a winning game and a reward of 0 (zero) otherwise. Given the huge state space of the backgammon game, the player required hundreds of thousands of games to become proficient [92].

For the application of RL to assistive robotics, and in particular, to robotic agents which learn gestured human commands, such lengthy training phases are not feasible. As such, several variants of knowledge transfer between human teachers and robots have been devised. A summary of these approaches is described by Knox and Stone [65] which covers advice-taking agents, learning by example, and human-generated reward signals. The authors note that the advising of agents in a meaningful way may involve expertise beyond that of a typical user. Learning by example in which the user demonstrates a desired response may place a burden on the user to observe the outcome, or require that they possess expertise to generate an adequate example (as with simulated aircraft operation).

One straightforward way for a human teacher to influence the learning of a robotic agent is by allowing them to control a simple *good/bad* reward indicator. Kaplan et al. [58] proposed the use of the animal training technique known as *clicker training* to teach an AIBO dog robot to learn complex actions. Blumberg et al. [14] extend this idea to use reinforcement learning to instruct virtual characters. Breazeal and Thomaz [94] use RL in a simple virtual kitchen environment called *Sophie's Kitchen*. The environment uses a relatively small state-action space to show that a

task (baking a cake) within this space can be learned through clicker-like guidance from a human teacher.

Fiebrink et al. [34, 35] attempt to improve machine learning through user feedback and thereby, to generate a model of human gesture recognition that learns from the user. The goal of the research is to take user feedback on the correctness of a gesture recognition model to improve the model. In general, the user exercises *direct interaction* to alter a training data set. That is, user advice is used to essentially *relabel* newer and more correct training data. In a similar way, Förster et al. [37] utilize a *teacher* signal to train an activity recognition system where no ground truth training data is available. The user provides *correct/error* feedback which is used to judge the system's recognition of an activity according to the correctness of classification (essentially a relabeling of classifier output). The authors show that the teacher signal allows the system's modified kNN classifier to learn more quickly and with equal accuracy than when in the presence of ground truth data.

Kartoun et al. [60] create an extension to Q(λ) (Q-Learning with mulitple step eligibility tracing) to switch between fully autonomous and semi-autonomous operation (in which human guidance is accepted) in a bag emptying task. This approach, called CQ(λ) allows for levels of collaboration with a human observer. It is shown that the influence of human guidance speeds the learning process.

Similarly, Kuno et al. [67] use face identification and hand gesture recognition to control an intelligent wheelchair. The system makes an initial assumption of an appropriate direction and speed response for the wheelchair based on a best guess at the user's gesture. If the user approves of the response, it is assumed that they will repeat the gesture. In this way, the chair's response is reinforced and the gesture is deemed registered for future use.

Since this research assumes an unskilled human user who is attempting to

train an assistive robot, the clicker training style is chosen as a viable means for the user to express approval or disapproval of a robotic response. And, as noted by Knox et al. [64], a human trainer has a broader view of the benefit of a specific individual action than is considered by MDPs. Rather, the trainer may give reward based on a qualitative view of how a task *should* be performed by an agent. These authors suggest that this observation indicates that using a human teacher is more akin to a supervised learning approach. However, for simplicity and to facilitate the incorporation of other reward modalities in the future, this work makes use of the Q-Learning method.

It is assumed that a goal configuration is known by the user and that through gesture and a simple reward/punishment evaluation of the robot's response, the robot will eventually achieve this configuration. Certainly, higher dimensional configurations will be more challenging to attain with this simple binary feedback mechanism. Further, some trajectories toward the final configuration may allow the robot to pass through undesirable configurations. The assessment of trajectory is left to future work. Thus, our learning algorithm essentially undertakes the problem of developing the user's goal rather than modelling the environment to obtain ever higher rewards. The shortest path to the goal is understood to yield the greatest benefit.

1.4 Summary

A significant body of work exists in the area of automated recognition of human activity and gesture. The diversity and persistence of efforts to sense, classify and respond to human motion is evidence of the importance of this research especially as it relates to assistive robotics. This review of the applicable literature presented here has attempted to clarify major problem segments associated with such recognition and

to identify specific aspects of the existing solution set which inform the experimental approaches described in the following chapters.

In particular, this review has allowed us to focus on possible strategies which best cater to the goals of aging in place. Thus, the use of ambient, non-vision based sensing is indicated. In contrast to an abundance of approaches using wearable devices and conventional video cameras, the use of ambient, privacy preserving depth sensing in this work presents a key innovation. The sensor platform becomes more a part of the DNA of the environment as opposed to an unwelcome agent with which the user must contend. Recognition of gesture according to the acceleration features frequently used by humans to characterize motion (e.g. Dynamic Instants [84]) presents the opportunity for a robust classifier. Machine learning of gestured commands on line with the human user as teacher facilitates *come as you are* operation [12] that may accommodate unskilled or impaired users.

Chapter 2

Activity Recognition

A significant body of work exists in the area of automated recognition of human activity. Detection of user activity and inference of user context and intention are central to action planning by system software in order to control assistive robotic components. Despite the development of many promising techniques, the goal of robust generalized recognition of human activity remains elusive. Due to such factors as changes in lighting and camera position, and variations in anthropometry and speed of execution, the problem remains largely unsolved [10, 56, 84].

The loss of dexterity in the hands is a key factor affecting performance of certain ADLs including precision and grip tasks. Further, decreased manual dexterity is often coupled with pathological conditions such as osteoporosis and Parkinson's disease. Hence work to characterize a user's manual dexterity is of interest and importance to the broader field of research that considers how impaired users perform ADLs [21].

In this chapter, an experiment is presented which focuses on the automated classification of representative motions related to manual dexterity. In keeping with the research goals stated in section 1.2, the use of ambient, non-intrusive sensing

techniques is explored. It is shown that the comparatively sparse sensor data stream available from such devices can be used to construct a robust classifier for simple actions.

2.1 Method

This section describes the laboratory fixture used to collect both video and IR motion sensor data sets as well as the analysis technique used to generate descriptors and to classify motions. Motion samples were collected for three activities which were chosen for their fundamental importance to a person lying in bed as in a healthcare setting. These activities included:

1. (*Reach*) Bringing a cup to the mouth.
2. (*Press*) Pressing a nurse call button.
3. (*Grab*) Grabbing the bed rail.

These candidate motions are all functions involving the hand which have been used in studies [47, 79] exploring hand and finger mobility in aging adults.

2.1.1 Data Collection

Motion data samples were collected at 17 Hz over seven second intervals using both a Logitech video web camera and a Panasonic AMN23112 analog IR motion sensor. For repeatability, the motions were performed by a PUMA robot serving as a proxy for the human arm. Samples recorded from an array of points over the surface of a virtual sphere surrounding the workspace of the robot. To provide a visual context to the reader, a mock up of the scenario as it might exist a hospital patient

room is shown in Figure 2.1. At the time of this experiment, the sensor platform was envisioned to be of the form shown. A continuum surface [57, 74] would be used to position its embedded IR sensors at an optimal vantage point for characterization of the user's activity. The more recent work described in chapters 3 and 4 moved beyond this vision to a more versatile sensor paradigm.

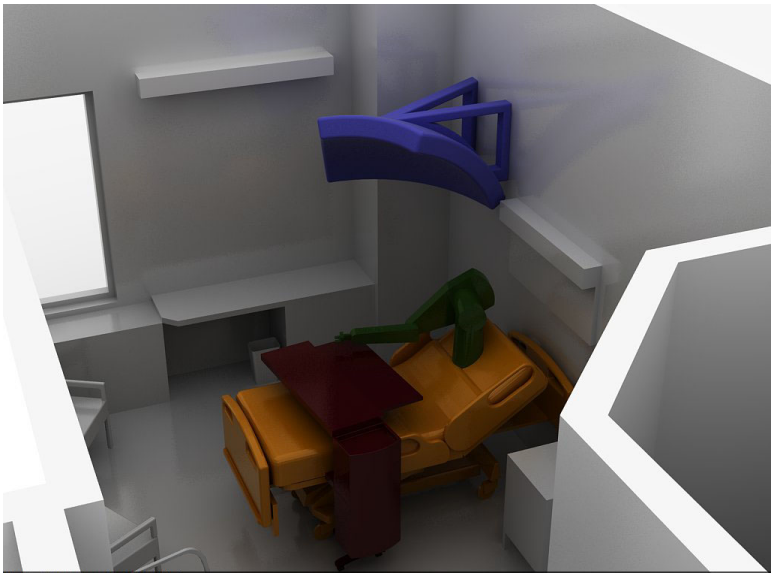
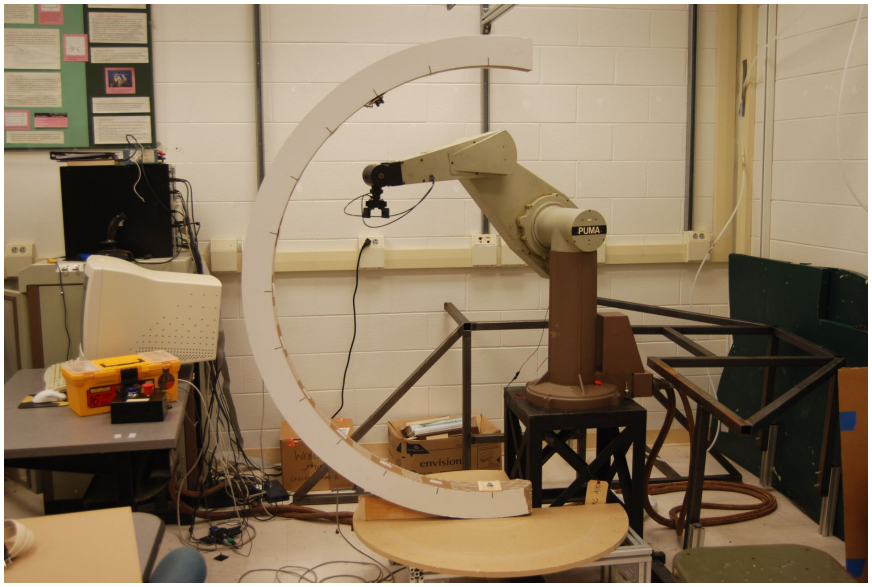


Figure 2.1: Hospital room scenario with continuum sensor surface.

The rotating arc fixture shown in Figure 2.2 was constructed to sweep the surface of the virtual sphere excluding that portion of the surface through which the PUMA was inserted. Sensors were placed on the interior of the arc and were trained toward the center of the sphere. This arrangement facilitated precise positioning of sensors at uniform vantage points.

Sensor vantage points (r, θ, ϕ) were selected uniformly over the surface of the sphere with $r = 30$", $\theta \in \{0°, 30°, 60°, \ldots, 180°\}$ and $\phi \in \{0°, 30°, 60°, \ldots, 240°\}$. For this study, the angle θ was measured downward from 0° at the vertical axis. This arrangement comprised an array of 63 sensor positions as depicted in Figure 2.3. For



(a)

(b)

Figure 2.2: (a) The rotating arc positioning fixture for spherical sensor placement. (b) The IR sensor board attached to the interior of the fixture.

the remainder of section 2.1 descriptions of the method employed in this experiment are the same for video as for IR sensor data. Individual video images are analogous to individual motion sensor readings. The ease of comparison, the sampling rate for the IR motion sensor was set equal to the frame rate of the camera.

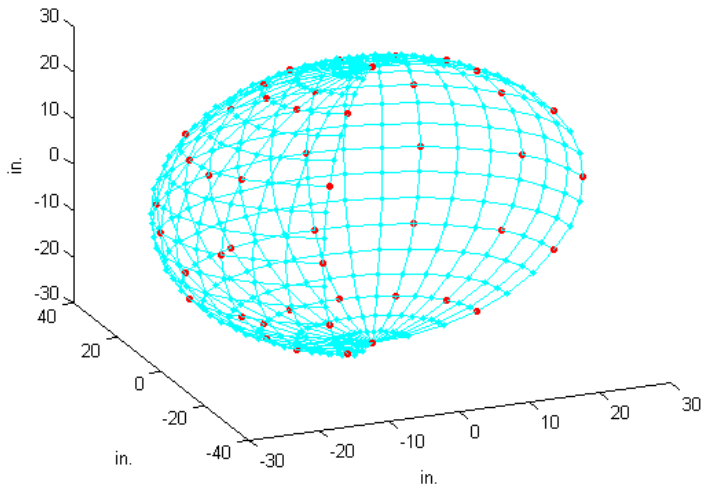


Figure 2.3: Sensor vantage points at 30° increments.

2.1.2 Descriptor Calculation

The formation of an image sequence descriptor for classification consisted of a two part process. First a Self-Similarity Matrix (SSM) was computed. A Histogram of Gradients (HOG) generated from the SSM then served as the descriptor. The Self Similarity Matrix $S(I)$ for each image sequence $I = \{I_1, I_2, \ldots, I_N\}$ was calculated

using (2.1).

$$S(I) = \begin{bmatrix} 0 & d_{12} & d_{13} & \dots & d_{1N} \\ d_{21} & 0 & d_{23} & \dots & d_{2N} \\ \vdots & \vdots & \vdots & & \vdots \\ d_{N1} & d_{N2} & d_{N3} & \dots & 0 \end{bmatrix} \tag{2.1}$$

where elements of $S(I)$ represent the Euclidean distance measure d_{ij} between image pairs $\{I_i, I_j\} \in I$ such that

$$d_{ij} = ||I_i - I_j||_2. \tag{2.2}$$

Assumptions implicit in the distance calculation of (2.2) are that, for a given sequence, the sensor does not move and that the background does not change. Hence, any change in the intensity of a given image pixel denotes movement of a feature point. In this way, the total movement of all features can be represented as the summed differences between the pixel intensities of an image pair.

A local (overlapping) HOG descriptor is calculated for each point $i = 1 \dots N$ on the main diagonal of $S(I)$ where $N = 116$ for both video and motion data. The descriptor consists of a histogram of $m = 8$ gradient direction bins for each of $j = 11$ *log*-polar cells as shown in Figure 2.4 [56]. Gradients are computed using the Prewitt operator as suggested in [28]. Bin entries are weighted by the associated gradient magnitudes. Since $S(I)$ is symmetric, only the entries above the diagonal are included in the descriptor computation. Descriptors for all points are concatenated to form a composite descriptor H for the action sequence. Hence, for our data set, H is an $(8 \times 11) \times 116 = 8 \times 1276$ matrix.

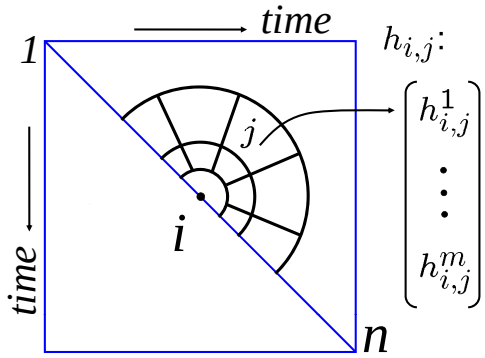


Figure 2.4: HOG descriptor format [56].

2.1.3 Action Classification

Class exemplars for each of the three candidate actions were calculated as the mean HOGs for a specified percentage of the available data. These HOGs were selected randomly and constituted the training data. The remainder of the data points were used as test data. Each test data HOG was compared with each of the exemplars and classified by the exemplar to which it was nearest according to (2.3)

$$i = \arg\min_{j} D_E(H_{test}, H^j_{train}) \tag{2.3}$$

where H_{test} is a test data point, H^j_{train} is one of $j = 3$ candidate action classes, D_E is the distance to the exemplar using the Frobenius norm, and i is the classification. The percentages of data points used as training data were varied from a single vantage point up to 50%, at 10% increments. In this way, it was possible to determine whether a descriptor from any given vantage point resembled that of its class exemplars so as to validate/invalidate the claim that the stability of the SSM across the range of sensor vantage points allowed for robust view invariance.

2.2 Results

Figure 2.5 shows SSM plots for the three candidate motions taken from orthogonal vantage points. Rows of the figure correspond to specific motion classes. Columns correspond to the vantage points from which the samples were taken. The relative similarity between subfigures in each row is indicative of the stability of the SSM as the basis for a view-invariant classifier. Use of orthogonal vantage points for this comparison provides support for the assertion of view-invariance. Visual comparison of all 63 collected samples further bears this out. Quantitative results are given in section 2.2.1.

2.2.1 Video Data Classification

Classification results for the video sequences are given by Table 2.1. Because exemplars were calculated using a percentage of the available data points selected at random, any individual execution of the classifier could be expected to yield wide ranging results. To mitigate this effect, all statistics shown in the table have been averaged over 20 classification runs. Results were favorable ($> 90\%$ accuracy) when multiple data points (10% and higher) were used to calculate the exemplars. Further, they show continued improvement as more data points are used to compute exemplars. It is notable that, when a single data point was used as a class exemplar, over 77% classification accuracy was still achieved. This result lends further credibility to the assertion by [56] that SSMs provide a stable representation across the range of sensor vantage points and that the method does support view invariant activity recognition. Also, since the *grab* motion is kinematically distinct from either *reach* or *press*, classification accuracy is generally highest for this class.

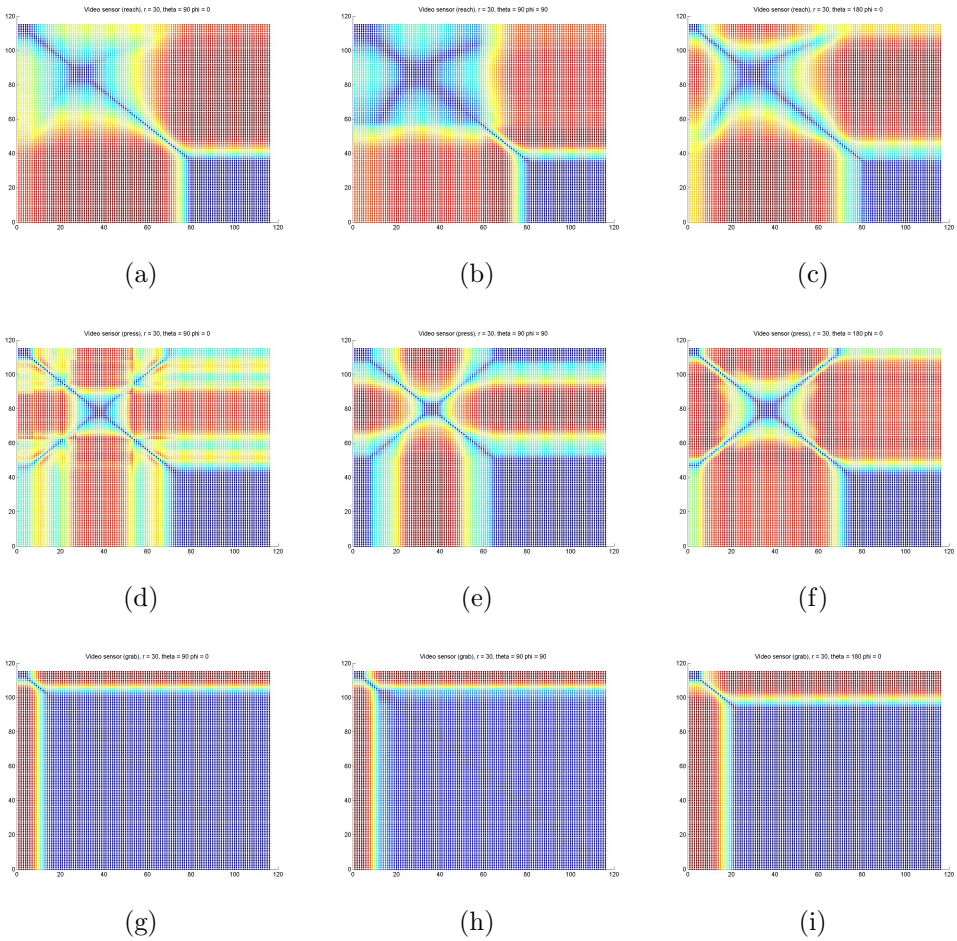


Figure 2.5: SSMs for video sequences taken from orthogonal views. Rows represent specific motion classes: (a,b,c) reach, (d,e,f) press, and (g,h,i) grab. Columns represent specific sampling vantage points (r, θ, ϕ): (a,d,g) sensor placed at $(30", 90°, 0°)$, (b,e,h) sensor placed at $(30", 90°, 90°)$, (c,f,i) sensor placed at $(30", 180°, 0°)$.

Table 2.1: Video classification results.

	Classification Accuracy (%)		
Training Points	Reach	Press	Grab
1	84.11	77.82	90.24
10%	95.70	90.96	100.00
20%	96.18	93.43	99.90
30%	96.00	93.11	100.00
40%	96.05	94.21	100.00
50%	96.41	92.50	100.00

2.2.2 Motion Sensor Data Classification

SSMs for IR motion sensor data readings taken from the same vantage points used above (see Figure 2.3) are given in Figure 2.6. It can be seen that, although there is a nominal resemblance between SSMs for a given class, the similarity is clearly less than that for video SSMs.

Classification results for the motion sensor readings are given by Table 2.2. Results are poor when only a single view is used to generate exemplars - no better than random guess. Again, the *grab* motion shows greatest accuracy, owing to its inherent dissimilarity from the other motion classes. Results improve as the percentage of data used to train the classifier is increased (reaching 65% - 70%), though, not to a level that could be considered reliable.

Clearly, motion sensor data does not carry the richness of information found in video data. Single video frames consist of large number of pixels versus only a single numerical reading for motion sensor data points. However, results with motion sensor data are promising nonetheless. To increase the amount of information available for activity classification through motion sensing, the following two approaches were attempted.

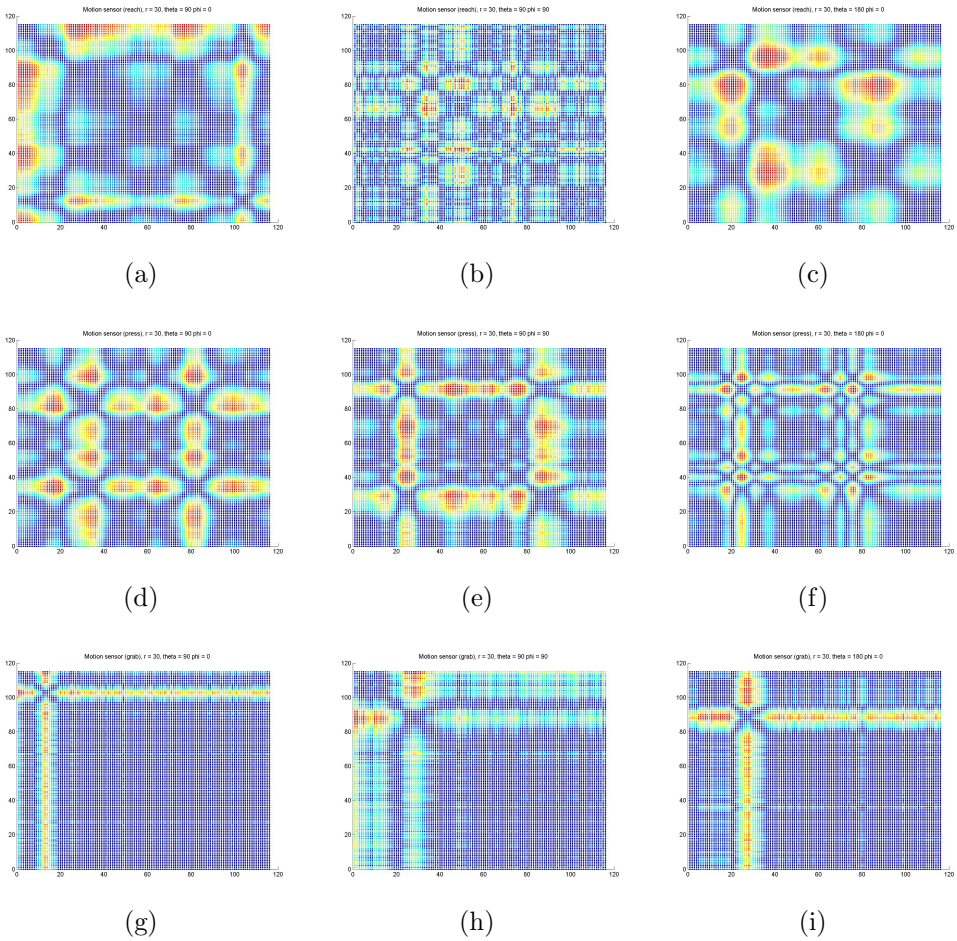


Figure 2.6: SSMs for motion sensor data taken from orthogonal views. Rows represent specific motion classes: (a,b,c) reach, (d,e,f) press, and (g,h,i) grab. Columns represent specific sampling vantage points (r, θ, ϕ): (a,d,g) sensor placed at $(30", 90°, 0°)$, (b,e,h) sensor placed at $(30", 90°, 90°)$, (c,f,i) sensor placed at $(30", 180°, 0°)$.

Table 2.2: Motion sensor classification results for sensors at 30° increments.

	Classification Accuracy (%)		
Training Points	Reach	Press	Grab
1	25.08	24.68	86.45
10%	50.79	56.32	88.60
20%	56.27	63.04	93.73
30%	61.89	61.78	94.78
40%	60.39	73.68	96.97
50%	65.63	70.16	96.56

Table 2.3: Motion sensor classification results for sensors at 15° increments.

	Classification Accuracy (%)		
Training Points	Reach	Press	Grab
1	37.50	35.82	88.60
10%	65.73	65.90	97.16
20%	72.39	73.30	98.99
30%	75.00	75.40	98.84
40%	75.04	74.18	99.08
50%	74.87	76.37	98.93

First, as suggested in [111] increasing the number of sensors offers an intuitive method for increasing available information. To this end, the density of vantage points for motion sensing was increased to 15° increments over the sphere such that points (r, θ, ϕ) were $r = 30$", $\theta \in \{0°, 15°, 30°, \ldots, 180°\}$ and $\phi \in \{0°, 15°, 30°, \ldots, 255°\}$. This constellation of sensors effectively quadruples the original number of motion sensor vantage points to 234. Classification accuracy for this scenario improved by, typically, 5%-15% as can be seen in Table 2.3. Still, however, such results do not practically approach the results available through video sensing.

Second, a surface contour encompassing an array of sensor vantage points

Table 2.4: Motion sensor array classification results.

	Classification Accuracy (%)		
Array Size	Reach	Press	Grab
1×1	56.27	63.04	93.73
1×2	76.18	75.78	98.63
2×2	86.76	86.18	100.00
3×3	94.90	95.39	100.00

was envisioned. Such a contour was emulated by fusing sensor inputs by averaging readings over regional subsets of the virtual sphere. Table 2.4 shows several scenarios for such arrays. The table assumes 20% of data points were used to calculate class exemplars. Using this scheme, motion sensor data approaches the accuracy found using video data for arrays of 2×2 and larger.

2.3 Summary

In this chapter, the use of Self-Similarity Matrices (SSM) to generate Histogram Of Gradient (HOG) classifiers for activity recognition has been explored. It has been shown that video recordings of basic motions can be classified by this method with a high degree of accuracy.

Further, and most interestingly, we have used *non-video* motion data to evaluate whether a holistic activity representation might be useful in privacy sensitive applications. It has been shown that motion sensor readings of basic actions can be classified by this method with a promising accuracy. Where single sensor inputs are used as class exemplars, classification accuracy is sensitive to vantage point and thus performs poorly. Where multiple descriptors are averaged to produce exemplars,

classification improves though it is still subject to the choice of vantage point for best outcomes. Coupled with our robust classification for video, we interpret these findings as supportive of sensor view invariance in that the *appearance* of SSMs for a given class is stable enough over the range of vantage points to collectively form a useful discriminant. It is noted, however, the large 8×1276 HOG descriptor presents a potentially prohibitive burden in terms of computation and storage.

Experimentation with single motion sensor inputs also yielded poor results. However, when multiple sensor views are combined into a single average reading for a small contour surrounding a vantage point, results improve significantly. Hence, the use of motion sensor data for the purpose of activity recognition appears to be a viable area for continued exploration.

Given this finding, it is intuitive that the fusing of larger numbers of sensors might enable further improvements in classification accuracy. At the logical extreme of this idea, one can easily envision a dense array or *point cloud* of sensor readings which effectively map the topography of the participant to produce a moving depth image. Such an array is produced by the near-IR technology that is the basis for the Microsoft Kinect RGB-D sensor as discussed in section 1.3.3.1. The experiments described in chapters 3 and 4 utilize the Kinect platform.

The detection and characterization of generalized human activity is a decidedly immense prospect. Paraphrasing Bobick [15], recognition of basic actions will never be generally applicable to identification of higher level activities such as shop-lifting unless some portion of the act can be ascribed to a specific movement. Indeed, no collection of atomic movements can identify higher level activities without benefit of the experiential knowledge which allows inference. As such, the key findings of this chapter are of less impact to the problem of activity recognition than to the sub-problems of sensor selection and motion representation. However, these results

are seen as integral to the more focused problem of gesture recognition, and thus, to the goal of improved modalities for HRI. Chapter 3 applies these findings to the use of gesture as the basis for an human-robot command interface. The use of human guidance is employed so that the machine learning component of our approach makes use of experiential knowledge during the training phase.

Chapter 3

Gesture Recognition

In chapter 2, the problem of activity recognition considered utilizes a formulation that is common in much current research. Regardless of the method employed, the success of activity recognition (and thus, of gesture recognition) is measured according to classification accuracy. However, framing the problem in these terms implies that the researcher has specified an appropriate choreography of the motion and that the actor will perform according the the researcher's expectation.

Given that the target population of the ART project includes potentially unskilled or impaired users, the common approach of matching a performed gesture to some element in a stored gallery of templates is inappropriate. Rather, the problem of gesture recognition is formulated here as the mapping of gestures to a user's desired environmental configuration. Strict classification is bypassed in service of the user's goal which may be initially unknown. An analogy of this formulation may be drawn to animal training. The animal has little purpose for labeling (classifying) its trainer's actions. Moreover, it associates its own actions in response to the trainer's command with some type of reinforcement (reward or punishment) from the trainer. A reward indicates that a particular action is to be repeated in a given context; punishments

indicate actions to be avoided. By affording the user some mechanism (such as a push button) of indicating their relative satisfaction with a robotic agent's response to gesture, the gestures themselves become part of a command vocabulary to the agent. As noted by Kaplan et al. [58], this form of training, termed *shaping*, has been used successfully by animal trainers to break down complex tasks into manageable segments in which simple reinforcement signals may be effective. Unlike Kaplan's application to the Sony AIBO robotic dog, research work presumes no working action *primitives* such as AIBO's ability to walk, kick a ball or dance. Rather, static final configurations of the robot agent are the defined goals.

In this chapter, an approach is introduced which explores a method for generating such a mapping between gesture and robotic configuration when the preferences of the user are considered. Two experiments are presented in sections 3.3 and 3.4 which examine the effectiveness of simulated human feedback in 1D and 3D configuration spaces, respectively. The 1D case (originally described in [116]) demonstrates the general efficacy of the sensor platform, data representation (DIs), and the Growing Neural Gas (GNG) clustering algorithm. The 3D case (introduced in [119]) shows the applicability of GNG in tandem with Q-Learning to provide a superior learning platform for the mapping of sensed gesture to the k-Nearest Neighbor algorithm commonly used in this problem space. Leveraging the topology of the GNG cloud, alternative distance metrics are also considered as introduced in [120].

3.1 Institutional Review Board Approval

The experimentation described in this chapter makes use of data collected from human participants. For the purposes of this research, only fellow researchers who were well-acquainted with the experiment's goals and procedures were consid-

ered for participation. Research using these data involved development of a software platform and machine learning algorithm for gesture recognition. Clemson University IRB-approved protocol 2011-266 was used in the collection of these data. Supporting documents for this protocol including the approved consent forms are shown in Appendix B. This development was termed *phase 1* of the protocol and is the subject of the experimentation described in this chapter. An envisioned *phase 2* of this research would involve work with unacquainted participants using the apparatus developed during phase 1. This is left to future work.

3.2 Method

This section describes the method and laboratory fixture used to collect gesture data. Included in this fixture are the sensor platform and software modules which generate data representation, perform clustering, generate robotic response action and issue user feedback (reward). Source code in C++ and Matlab [73] as well as data processing and control scripts are given in Appendix A.

An operational flow diagram of the system is shown in Figure 3.1. In the figure, the gestures performed by the user were drawn from samples collected from fellow researchers. The data collection fixture is described in section 3.2.1. Because this experimentation is intended as *proof of concept* work, the generation of user feedback was implemented as a simulated user to expedite training as described in section 3.2.6.

Data samples were collected for three arm-scale gestures which were deemed an essential baseline command set for the eventual operation of an assistive robotic agent. Although the overall approach implemented in the system places no expectation on the user to perform gestures in a particular manner, motion models for these

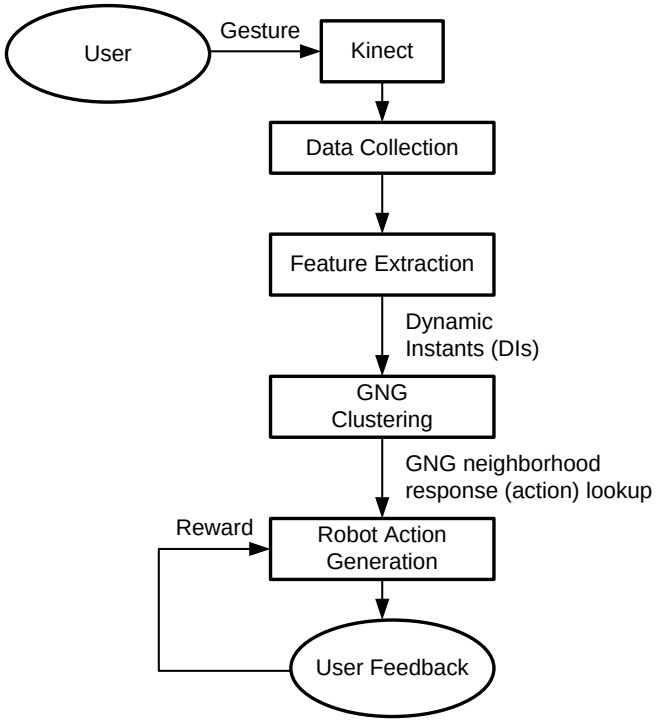


Figure 3.1: System block diagram. User feedback is automated for the experiments described here. The human user would generate the reward in the eventual implementation.

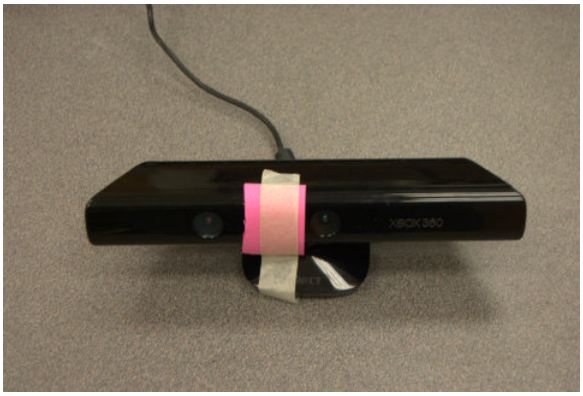
gestures were taken from the American Sign Language Dictionary (as demonstrated at [6]) to facilitate repeatability across the participant pool during the data collection phase of this experiment. The candidate gestures included *come closer*, *go away* and *stop*. Although the gesture command vocabulary was increased in later experiments, these were considered sufficient to show the viability of the approach at this stage of development. The *stop* gesture requires special treatment since it intuitively suggests that the robot is presently executing an earlier command. In order to properly handle such a scenario, segmentation of gestures from continuous free motion of the arm would be required. However, since segmentation is not the focus of the research, gestures are captured in isolated time intervals as described in section 3.2.1. The

problem of segmentation of free motion is left to future work. Instead, *stop* will not be interpreted in its literal sense, but rather as having a specific goal configuration similar to that of *come closer* and *go away*.

3.2.1 Data Collection Fixture

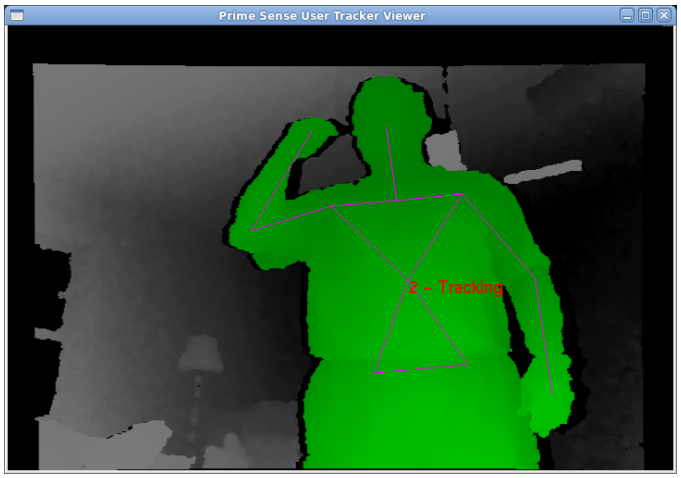
Data samples were collected using the depth sensing feature of the Microsoft Kinect RGB-D system [75] shown in Figure 3.2. The Kinect produces depth maps of the user at approximately thirty frames per second. Samples were collected over five second intervals for a total of 150 data points per motion sample. Although RGB samples are also generated by the Kinect, these were not stored in order to preserve user anonymity. The RGB camera was covered (Figure 3.2a) in accordance with the IRB protocol and to assure participants that no identifying images were being collected. The Kinect was set at desk height (75 cm) with the participant standing at a distance of 1.3 m. The Kinect was angled so that the eleven upper-body joints (Figure 3.2c) were visible in the depth image. Participants were invited to occasionally shift their weight or angle of approach slightly so as to introduce nominal variation in the collected data. Five participants each performed fifty repetitions for each of the three candidate gestures. This yielded 250 samples of each gesture type for a total of 750 samples.

The data collection program was developed using the Robot Operating System (ROS) [88]. ROS was selected for its open source and for its active community of research-oriented users. Further, it supports a variety of simulated and real world robotic platforms through a message based publisher/subscriber environment. Thus, direct migration of this research to the proposed hardware platform (Figure 1.2b) is expected to be a viable path. Within ROS, the Kinect data stream was accessed



(a)

(b)



(c)

Figure 3.2: Kinect sensor data collection setup: (a) The Kinect sensor with RGB camera covered. (b) A participant performing the *come closer* gesture. (c) The PrimeSense OpenNI depth image showing skeletal tracking during the *come closer* gesture. Note that the PrimeSense OpenNI viewer displays the participant's mirror image.

using the PrimeSense OpenNI Kinect package [87] to track the skeletal joints of the participant by ROS messages. An example of the Kinect depth image showing skeletal tracking is shown in Figure 3.2c. Depth data for eleven joints were available over the sampling interval. However only a participant's left hand is considered for gesture characterization in these experiments. Data points consist of (x, y, z) coordinates of the location of the left hand.

3.2.2 Feature Extraction

Using an approach similar to [84], Dynamic Instants (DI) were extracted from each 150-point data sample for motion of the left hand joint. Position data for each of the three dimensions were first smoothed by convolution with the discrete Gaussian kernel given by (3.1) with $\sigma^2 = 1.0$ [50].

$$G = [1, 4, 6, 4, 1]/16 \tag{3.1}$$

As a further smoothing step, a moving average of seven time steps was applied to the position data so that short term *jitter* of the actor could be filtered and longer term trends could be captured.

Velocity and acceleration data were then computed from position data for each dimension. The five highest occurrences of peak acceleration were selected as the dynamic instants. As discussed in [84], such peaks occur at sharp changes of direction or speed, and starts/stops. For the DIs used in this work, the (x, y, z) coordinates and the frame number were recorded. Given the Kinect's capability to represent the positions of these peaks in 3D space and with the frame number accounting for discrete time, the spatial trajectory of gesture execution is grossly replicated. Hence, DIs did not require the extra dimensions of velocity and acceleration to be stored for

effective discrimination between gesture types.

Feature vectors for each sample were constructed by the concatenation of the five DIs to yield a 20×1 descriptor as shown in Figure 3.3. Both frame numbers and coordinate values were scaled to $[0, 1]$ based on the range of values of their respective types so as to prevent any given field from dominating the feature vector. Feature vectors were then clustered using the Growing Neural Gas algorithm (Algorithm 1, details in the next section).

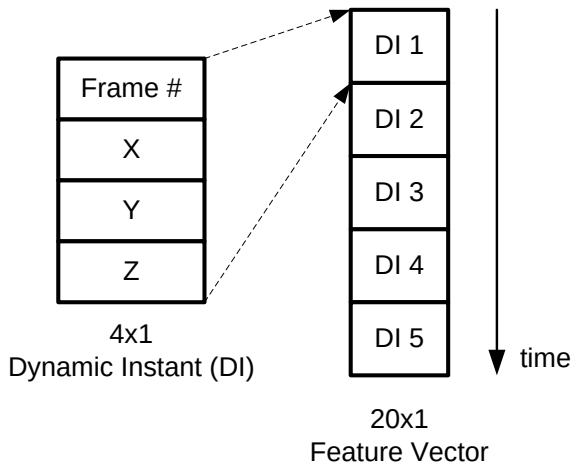


Figure 3.3: Feature vector format for a depth-sampled gesture. DIs are concatentated in chronological order by frame number.

3.2.3 The Growing Neural Gas Algorithm

The Growing Neural Gas (GNG) algorithm proposed by Fritzke [39] is a vector quantization technique in which neurons (*nodes*) represent *codebook* vectors that each encode a submanifold of input data space. In this regard, GNG is similar to the Neural Gas (NG) algorithm proposed by Martinetz and Schulten [71]. GNG differs from NG in its ability to form connections between nodes and thus preserves a topological representation of input space in a manner functionally similar to the Self Organizing

Feature Map (SOFM) [66]. Further, GNG is capable of adding new nodes over time so as to effectively map the topology of a non-stationary input data distribution. The basic GNG algorithm is given by Algorithm 1 [39]. For the implementation of GNG used in this work, operating parameters were: $\epsilon_b = 0.05$, $\epsilon_n = 0.0006$, $\lambda = 100$, $\alpha = 0.5$, $\beta = 0.0005$ and $a_{max} = 88$. Also, a maximum limit of 100 nodes was imposed on the network. Data structures associated with the implementation of GNG and of the larger system are discussed in section 3.2.4.

Algorithm 1 The Growing Neural Gas (GNG) algorithm

1: Begin with a set A of two nodes at positions w_a and w_b in R^n: $A = \{a, b\}$.
2: Initialize a set of connections to the empty set: $C = \emptyset$.
3: **repeat**
4: Apply an input signal ξ according to $P(\xi)$.
5: Find nodes s_1 and s_2 in A closest to ξ.
6: Establish a connection between s_1 and s_2 if one does not exist: $C = C \cup \{(s_1, s_2)\}$.
7: Set the age of the connection (s_1, s_2) to zero.
8: Increment the ages of all edges connected to s_1.
9: Adjust the local error of s_1 by the square of its distance to the input: $\Delta E_{s1} = ||\xi - w_{s1}||^2$.
10: Move s_1 toward ξ by fraction ϵ_b: $\Delta w_{s1} = \epsilon_b(\xi - w_{s1})$.
11: Move the topological neighbors of s_1 toward ξ by fraction ϵ_n: $\Delta w_n = \epsilon_n(\xi - w_n)$.
12: Remove all edges having an age greater than a_{max}. If this leaves any nodes with no connecting edges, remove them also.
13: **if** $(numInputs\ mod\ \lambda = 0)$ **then**
14: Determine the node q with maximum error.
15: Insert a new node r halfway between q and its neighbor f with the largest error: $A = A \cup \{r\}$ such that $w_r = 0.5(w_q + w_f)$.
16: Decrease the error of q and f by fraction α: $\Delta E_q = -\alpha E_q$ and $\Delta E_f = -\alpha E_f$.
17: Initialize the error of the new node to the interpolated error of its neighbors: $E_r = (E_q + E_f)/2$.
18: Decrease all node error variables by fraction β: $\Delta E_c = -\beta E_c \ \ (\forall c \in A)$.
19: **end if**
20: **until** Stopping criteria is met.

3.2.4 Data Structures

3.2.4.1 The A Data Structure

The A data structure contains the list of reference nodes (codebook vectors) generated during execution of the GNG algorithm (Algorithm 1). Each node A_i carries the associated fields which support mapping of input gestures to 3D robotic response configurations. These fields include the node's feature vector, node label, and of key importance, the response configuration $(x, y, \theta)_i$ (or *action vector*) for a 3-DOF robot, and its most recent user-generated reward. Table 3.1 includes a complete listing and descriptions of the fields for the A data structure.

As the GNG algorithm updates the cloud of reference nodes with each input vector, the nearest reference node in the cloud already holds a learned robotic response based on the history of the system. In this way, the system avoids the task of correctly labelling the input in favor of generating a desirable response to it. Rather, the action vector associated with the node serves as its label. Using a reward signal from the user to gauge the quality of response, the algorithm attempts to improve the desirability of the action as it quantizes the input space.

3.2.4.2 The C Data Structure

The C data structure consists of a list of undirected connections (or *edges*) between reference nodes in the GNG cloud. This structure also includes the *age* of the connection. The fields of C are described in Table 3.2.

3.2.5 Simulation Environment

As a simulated proxy for a 3-DOF mobile robot, the ROS Turtlesim environment was used. Turtlesim is a basic ROS tutorial construct capable of accepting and

Table 3.1: Fields for nodes in the A data structure (node list).

Field Name	Description
`numObs`	Number of observations. The total number of input gesture patterns which have been previously observed.
`nodeLabel`	Node label. Each node carries a unique integer label. Initially, these will be consecutive but may become non-consecutive as nodes expire due to lack of use.
`numConx`	The number of connections to other nodes within the GNG cloud.
`featureVec`	The feature vector (or weight vector) of the node. This is the node's mapping in input space. Each gesture input is compared to the feature vector in order to locate its nearest neighbor among the GNG cloud of reference nodes.
`action`	The current action vector. This is the node's mapping to output configuration space.
`last`	The action vector from the last time step during which this node was activated. A node may revert to this as their action vector when negative reward is issued for the current action vector.
`reward`	The most recent user-generated reward $\in \{-1, 1, 0\}$. A reward of -1 indicates that the action vector moved the robot agent away from the user's desired goal configuration. A reward of 1 indicates that the action vector moved the agent toward the goal. A reward of 0 indicates that the desired configuration has been reached. In this case, the node is fully trained and the learning policy is henceforth frozen for this node.
`ancestor`	The node label of the neighboring node (or itself) from whom the current action was learned.
`Q`	Accumulated past reward. This is equivalent to the length of the action vector from a global origin at $(x, y, \theta) = (0, 0, 0)$
`E`	Accumulated local error of the node.

Table 3.2: Fields of the C data structure (connection list).

Field Name	Description
`v1`	Vertex 1. The node (label) at which a connection is joined at one end.
`v2`	Vertex 2. The node (label) at which a connection is joined at the end opposite vertex 1.
`age`	The age of the connection.
`length`	The length of the connection.

attaining successive (x, y, θ) configuration goals. Movement with higher degrees of freedom is untested, though it is expected to be feasible using this approach.

3.2.6 Automated Reward Generation

A key aspect of this approach is the use of a user-generated reward which indicates the relative success of a robotic response to gesture. Reward is utilized to effectively guide online system learning in real time and with no initial training data that reflects any specific desired response. However, as previously stated, obtaining gesture data and the associated rewards may be expensive in terms of the burden placed on the participant. For this work, generation of a reward signal was automated in software according to predefined goal configurations. The manner in which rewards are generated and the configurations to be attained are specific to the individual experiment scenarios described in sections 3.3 and 3.4.

3.3 Gesture Learning in 1D

This section describes initial experimentation using the GNG recognition paradigm described above to generate learned robotic responses in one dimension. In this experiment, robot configurations were limited to points along the diagonal line $y = x$ by simulated agents in the ROS Turtlesim environment. Although both the x and y dimensions are changing for the agent, they are doing so in unison and with equal magnitude so as to simplify the response to generated rewards. Positive rewards indicate that the agent should continue forward in its current direction; negative rewards indicate that the agent should move in the opposite direction.

3.3.1 Reward Generation

For this early work, user-generated reward was automated programmatically according to predefined goals. These goals represent relative translations (x, y, θ) from the starting position of the simulated robotic agent $(0, 0, 0)$ and were selected to be easily distinguishable. The goal configurations are given by Table 3.3.

Table 3.3: 1D goal configurations for a simulated mobile robot.

Gesture Type	(x, y, θ)
Come closer	$(3.95,\ 3.95,\ 315^o)$
Go away	$(-3.95,\ -3.95,\ 315^o)$
Stop	$(-2.00,\ -2.00,\ 315^o)$

Rewards were generated as an integer value in $\{0, \ldots, 10\}$ as shown in Figure 3.4. Reward values less than 5 indicate a response that moved farther away from the desired configuration than where it began. Values greater than 5 indicate movement

toward a desired goal. For example, a response which caused the robot to move 20% closer to the goal would cause a reward of 6 to be generated.

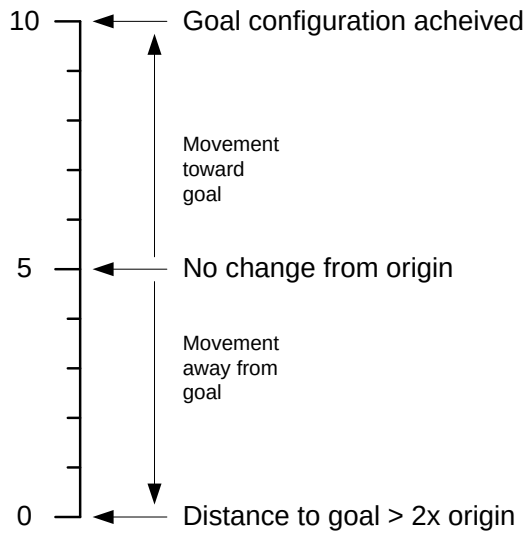


Figure 3.4: User-generated reward scale for 1D goal configurations.

3.3.2 Response Refinement

The system receives a reward value from simulated user and uses it to refine and update the generated response. The portion of the system responsible for the update is isolated from that which generates the reward. This is to emulate a future scenario in which an actual human user is providing the reward signal. In this experiment, the update is performed according to a simple rule-based approach given by Algorithm 2.

Algorithm 2 1D Response Update Rule

```
if reward < 5 then
    Move in the opposite direction by a fraction of the distance indicated by the feedback.
else if reward > 5 then
    Move in the current direction by a fraction of the distance indicated by the feedback.
else
    Move in the direction indicated by signs of (x, y) in the present response (i.e. make a guess).
end if
```

3.3.3 Experimentation

The 750 collected samples (see section 3.2.1) were randomized and presented to the system as input. One application of all 750 samples constituted an input *epoch*. For each sample, feature vectors (Figure 3.3) were computed and passed to the GNG algorithm, a response was issued, reward was automatically generated and the response was updated accordingly. The per-sample error was calculated between the updated goal configuration and the known goal for that sample's gesture type. Following each epoch, the average error per gesture type was also computed. In this manner, sixty epochs were executed. Results are shown in Figure 3.5a. Average error can be seen to trend downward with typical error less than 1 m within approximately 15 epochs. Goal seeking results (in Turtlesim) using the mature GNG cloud can be seen in Figure 3.6.

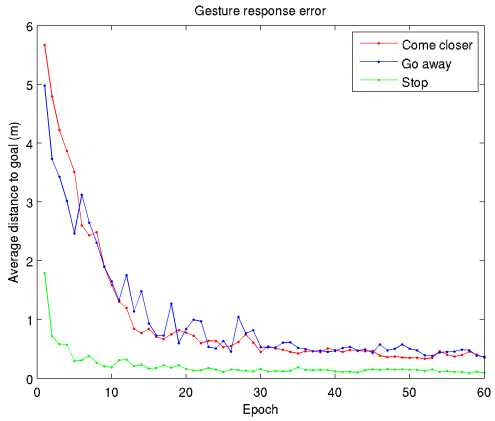
Dissimilarity among computed DIs for a given gesture was seen to effect the smoothness of convergence: some samples of a given gesture differed significantly from the majority. For comparison with the original dataset, a subset of samples was also generated by filtering out samples with a significant number of outlying data points. Those samples having fewer than twenty data points farther than 1.5 standard deviations from the mean for the gesture type were retained. Those samples not

retained were deemed to be poorly separable from other gesture types. This filtering process reduced the data set to an average 191 samples per gesture type for a total of 573 samples. These results are shown in Figure 3.5b. Although the downward trend is smoother for the subset, the rate of convergence is similar. In a real world setting, users would be expected to exhibit natural variation in the performance of gestures. These results suggest that this system would be robust to such variation.

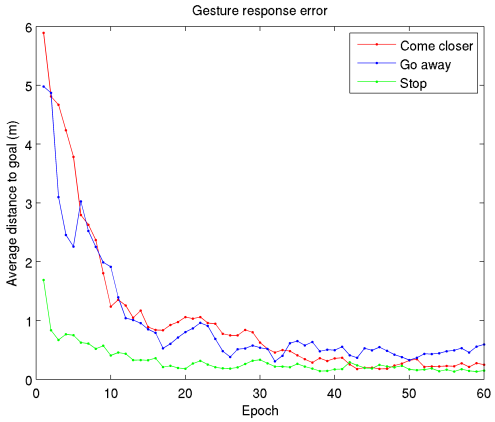
Perturbations within the GNG cloud can also be seen as the error curves do not descend smoothly. This may be explained again by samples within the data set which remain poorly separable despite filtering. Samples implicitly mistaken for the wrong gesture type would find their generated response to be far from desirable. However, despite such cases, the algorithm reliably re-converges toward goal configurations and average error continues to trend downward.

3.3.4 Summary

In this section, early work toward development of a gesture based human-machine interface has been presented. It has been shown that 3D data from the Kinect depth camera can be used to generate a useful descriptor of gesture in the form of prominent dynamic instants. Further, the GNG algorithm is capable of differentiating between these descriptors. Most interestingly, the goal of gauging the success of our learning algorithm based on the desirability of response rather than on a classifier label is shown to be practical. Clearly, the policy based update method we employ in this initial experiment is a simplistic approach to reinforcement learning. Further, the use of an integer-based reward signal provides unrealistically *rich* information to the response update process that allows it to converge relatively quickly. It is foreseen the generation of such information-rich rewards will place an undue burden on the



(a)

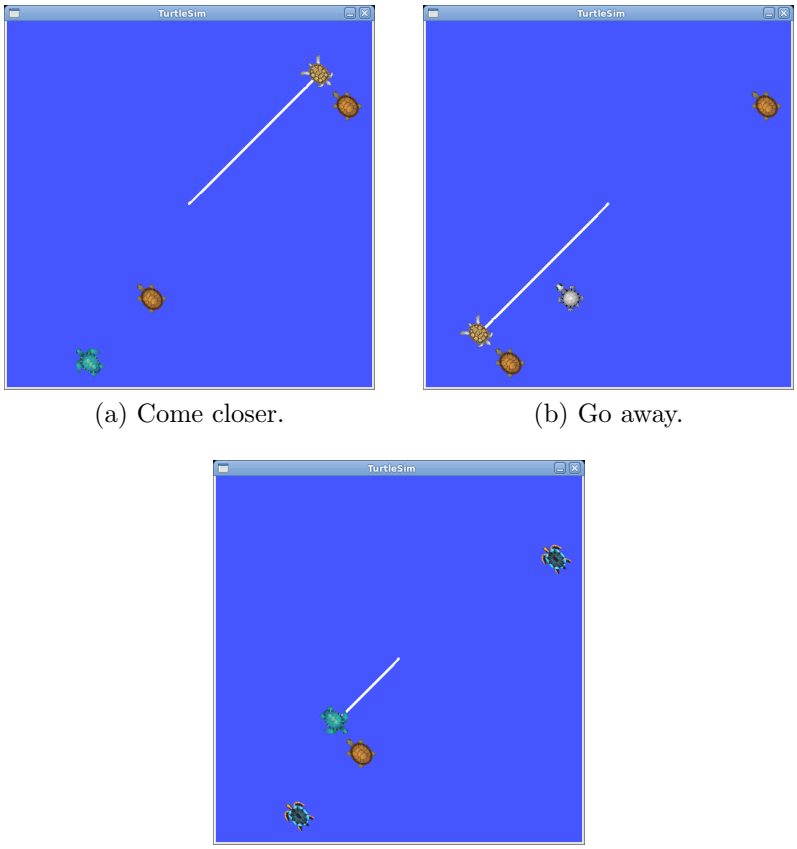


(b)

Figure 3.5: Average 1D gesture response error per epoch using (a) the full 750-sample data set, and (b) the 573-sample filtered data set. The response for the *stop* gesture can be seen to converge most rapidly since the desired configuration is nearest the origin as shown in Figure 3.6c.

participant to provide it. Development of a longer term value function to maximize user satisfaction while minimizing reward complexity will be of central focus of section 3.4 which follows.

It is noted that due to the need for improved separability in the data set, DIs present concerns regarding both spatial scale and speed of execution of the performed gesture. Future progress in this area could be expected to increase the speed of



(a) Come closer.

(b) Go away.

(c) Stop.

Figure 3.6: Motion paths for Turtlesim agent in 1D with a mature GNG cloud. The goal seeking turtle begins at the center of the frame and traces its trajectory (white line) as it moves along the diagonal line $y = x$ toward its 1D goal configuration. Markers are shown at the goal positions for each candidate gesture. Trajectories for each response are given in their respective subfigure as noted. The turtle agent can be seen to align with the appropriate marker for each gesture type. In all cases, the error is less than 0.1 m

convergence by the GNG algorithm, thereby reducing the expense of data collection.

3.4 Gesture Learning in 3D

This section presents experimentation which proceeds from the 1D scenario of section 3.3 to a more practical 3D scenario. Here, user-generated reward is reduced

from an integer to a binary signal so as to cater to the cognitive loading limitations of an impaired or unskilled user population. Instead of the richer integer-based rewards used in the experiment of the previous section, a *good/bad* indication which could be provided by a simple button push is used. The confluence of this more sparse reward signal and the higher-dimensioned configuration space in which the robotic agent operates will inevitably slow the rate at which the system is able to converge on a set of desired outcomes.

Toward overcoming this limitation, this experimentation demonstrates the available benefits which result from using the topology of the GNG cloud to conduct *neighborhood* learning wherein connected nodes may emulate one another's past success. A comparison is presented between the efficacy of the commonly used k-Nearest Neighbors (kNN) classifier and the proposed GNG/Q-Learning combination. The impact of data separability on neighborhood learning is also shown.

3.4.1 Q-Learning

In order to place this work within established terminology, the reinforcement learning paradigm of Q-Learning is adopted. This section describes the implementation of Q-Learning as it is used in these experiments in tandem with the GNG state-action data structure.

Within a reinforcement learning framework, an agent attempts to learn an optimal policy for mapping its set of possible states to future actions that are likely to be encouraged (or *reinforced*) through a reward signal from the environment. In this way, the total reward received throughout a sequence of state-action pairs may be maximized. Typically, a table of the state-action values (*Q values*) is maintained. As the agent encounters a state, the highest-valued action for that state is selected

and performed. The reward signal is observed and the table is updated according to (3.2):

$$Q_{t+1}(s,a) \leftarrow Q_t(s,a) + \alpha[r + \gamma(\max_a Q_t(s',a)) - Q_t(s,a)] \tag{3.2}$$

where (s,a) is a state-action pair, (s',a) is a particular next state-action pair which may be chosen from the current state, α is a positive learning rate, γ is the discount factor which allows near term rewards to be valued more highly than future rewards, and $r \in [-1,0,1]$ is the reward value. Reward values of -1 and 1 reflect user feedback of *bad* and *good* respectively. A reward of 0 reflects an outcome that requires no future adjustment (i.e. the human user is satisfied and training has been completed for a given gesture) and the policy is frozen for that state. For this implementation, each gesture sample is followed by a training episode of a single time step. Discounting is unnecessary since each reward from the human user is equally important as evidence of movement toward or away from the goal configuration. Hence, $\gamma = 1$. With $\alpha = 1$ and multiplying the reward by a step length (*stepLen*) of linear forward progress, the update rule is reduced to the form of (3.3). By viewing only the next action as a complete episode, the relation of (3.3) constitutes a degenerate case of Q-Learning. Nonetheless, the prevalence of this in machine learning research [43, 60, 63, 94, 99] warrants its adoption as a paradigm for future work.

$$Q_{t+1}(s,a) \leftarrow r(stepLen) + \max_a Q_t(s',a)) \tag{3.3}$$

For this research, $stepLen = 0.1$. This quantity is the same as the final error tolerance for the robot to achieve the goal configuration.

As previously mentioned, topological neighbors in the network may be expected to represent similar vectors in input space (sensed gestures) and should, there-

fore, produce similar output actions. Thus, the topology of the GNG cloud is utilized to accelerate learning by taking into account the rewards obtained by neighboring nodes (Figure 3.7). A network-structural interpretation of (3.3) can be stated as selection of the highest-valued action vector from the neighborhood of a reference node since that vector has the richest history of positive reward. The selection and update process is given by Algorithm 3 (adapted from Touzet [100]).

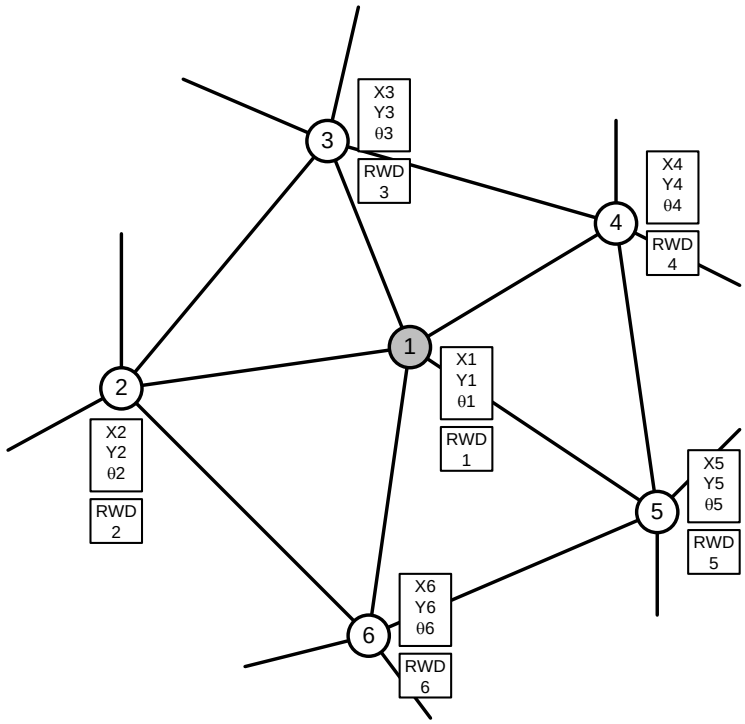


Figure 3.7: An example (2D) GNG neighborhood with associated action vectors and most recent rewards. Gestures which fall closest to node 1 will solicit action possibilities from nodes $1-6$ and then select the highest-valued of these action. The GNG *cloud* created in this research has a dimension of 20 using features vectors of the form shown in Figure 3.3.

Each node in the GNG network represents a state-action tuple consisting of an input (gesture) feature vector (which is the state label), an output action vector, and a Q value. As each gesture is sensed, the GNG network is scanned for the node with the closest input vector by Euclidean distance. The set of available actions is taken from

Algorithm 3 Q-Learning

```
Initialize Q values of all states (nodes): Q(s, a) = 0.
Initialize action vectors for all nodes: {x, y, θ} = {0, 0, 0}.
repeat
    Apply an input signal gesture vector ξ.
    Find node s closest to ξ.
    Find the set of nodes N which includes s and it's neighbors.
    if (r = 0) then
        The node is fully trained. Select the associated action.
    else
        Examine past rewards r ∀s ∈ N.
        if (r = 1) for any a ∈ N then
            Select and extend an action a to be performed according to (3.3).
        else
            (r = −1)
            Select the action (with angular correction).
        end if
    end if
    Perform the action.
    Observe and record the reward.
until Training complete: r = 0 ∀s ∈ S
```

those of its topological neighbors. For this implementation, the Q value is the length of the action vector. Since action vectors pointing to locations in configuration space farthest from the origin must have experienced the greatest number of positively rewarded episodes, they represent actions which promise the greatest likelihood of future reward. Given this intuition, (3.2) becomes a simple search for the longest vector in a node's immediate neighborhood. For positive reinforcements ($r = 1$), the node's action vector is updated with that of its highest Q-valued neighbor and is increased by a uniform step length (*stepLen*) along its current trajectory.

An action vector whose reinforcement value is negative ($r = -1$) indicates an action which would move the agent farther from the user's goal. If no neighboring node possesses a higher-valued, positively-rewarded action, exploration is required and the node's action vector is updated with a small randomized angular correction.

Repeated application of randomized correction will eventually yield an action that will be positively rewarded. This scenario is depicted in Figure 3.8.

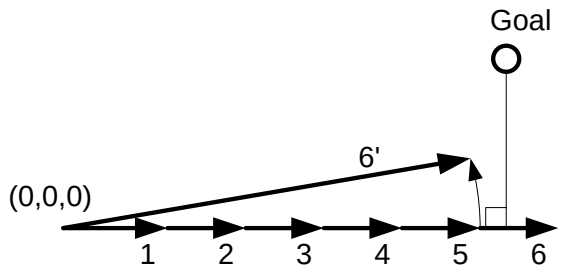


Figure 3.8: Q-Learning exploration for successive approximation of actions toward a goal. Steps $1 - 5$ receive positive reward and proceed in a consistent direction moving closer to the goal. Continuing this policy at step 6 would cause the robot to move farther from the goal than it had been at step 5 and would receive a negative reward. Random angular adjustments are attempted until the accumulated action vector comes closer to the goal as in step $6'$.

If the reinforcement is 0 (zero), the action vector is deemed *trained* and the policy for the associated node is frozen. In this implementation, such a node's action vector is removed from consideration by its neighbors in subsequent queries. This is to avoid the possibility of assigning action vectors repeatedly which may be incorrect for similar, but different gestures. In this way, the fully trained network will form an associative memory mapping between gestures and actions as shown in Figure 3.9.

3.4.2 *k*-Nearest Neighbors

The k-Nearest Neighbors algorithm (kNN) [13] seeks to classify an unlabeled data point using the known classes of neighboring data points. Typically, the nearest k points by Euclidean distance constitutes this *neighborhood*. The classification of the unlabeled point is determined as the majority class among its neighbors.

This algorithm is frequently used in reinforcement learning as a means of determining the state of an agent from sensor data. Knox recently employed this technique

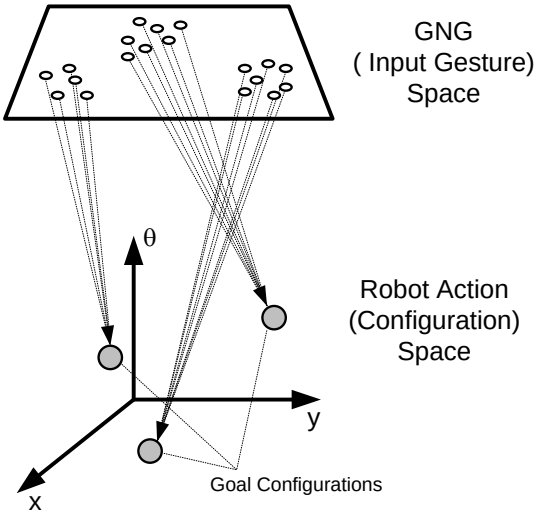


Figure 3.9: Mapping of input gesture to robot action. Input gestures are clustered by GNG and mapped through successive reinforcement to desired robot action vectors.

[63] to conduct user-guided training of machine agents in Q-Learning environments (see section 1.3.3.3). However, determination of state in the cases presented is typically a straight forward matter. In training a mobile robot to simple behaviors, the author uses the placement of a white card on the floor as a visual benchmark for the robot's sensor system. Depending on the behavior being learned by the robot, the relative position of the card in its perceptual field could be correlated with positive or negative rewards being issued by the trainer. This results in a relatively small state space for classification and action selection. Further, the actions themselves are fairly discrete in nature, including such options as *go forward*, *go backward*, *turn right* and *turn left*. In such a scenario, a kNN framework for state/action determination proved effective.

However, in the context of the experimentation described in this section, the ultimate preference of the user is initially unknown and may require a potentially large number of learning steps in order to be attained. Further, the sensor inputs are

complex gestures rather than a simple orientation with respect to a visual benchmark. Thus, a small set of discrete state labels for sensor input patterns is insufficient. In order to show the effectiveness of the GNG algorithm when combined with a Q-Learning framework in this thesis, a comparison is drawn between the performance of kNN-based action selection and the proposed experimental system. The GNG algorithm provides an efficient method for encoding manifolds of the input data space into single reference nodes. This feature of the approach is shown to allow more rapid learning of complex gesture representations and does so using only sparse rewards. The ability of GNG to perform in this manner makes it a good candidate for learning under the physical and cognitive loading constraints of potentially impaired human users.

In the kNN paradigm, a classification system must first have a set of training data for comparison. Such training data are typically labeled according to class. However, data sets used in this research are, pursuant to our system learning objectives, unlabeled. That is, no gesture inputs are accompanied by any sort of guidance for the preferred response by the robot agent. Working around this limitation, two implementations for kNN were defined and constructed as described below.

1. **kNN Type 1**. A set of 300 input samples (selected at random) are designated as *training* data. Test data patterns are then compared with the entire training set by Euclidean distance between their respective feature vectors. Actions from the k nearest neighbors among the training set are considered for execution (and extension) in the next time step. The training data pattern whose action was selected is updated with the reward received from the simulated user and the performed action for future consideration. In general, this is not a realistic implementation since unlabeled data cannot be characterized for uniformity

across classes. The sample data, although meaningful in this case, would not be relevant in any way to a real world learning scenario. The system would have no reason to base future outcomes to gesture on past data that is not correlated with the user's present actions.

2. k**NN Type 2**. As each test pattern is applied as input, it is added to a 100-entry history buffer. This emulates the method employed by Knox [63]. The entire contents of the history buffer is made to serve as training data and compared with new input patterns by Euclidean distance between feature vectors. Actions from among the k nearest neighbors are considered for execution (and extension) in the next time step. The current input pattern is updated with the reward received from the simulated user and the action performed. It is then placed at the top of the history buffer. After the history buffer is filled, the oldest entry is discarded. In this way, the history buffer is expected to contain the *latest and greatest* set of neighbors from which to select future actions.

It is shown in the experimentation which follows that, both kNN implementations are significantly outperformed in all analogous cases which utilize GNG. These outcomes are discussed in detail in section 3.4.9.

3.4.3 Floyd's Shortest Distance Algorithm

The topology of the GNG network affords the opportunity to employ network distance metrics in the formation of local network neighborhoods. Floyd's algorithm [101] determines the shortest path distances between node pairs in an undirected graph using edge lengths between individual nodes (see Algorithm 4). Following execution, the algorithm returns a matrix $D_{n\times n}$ for a network with n nodes in which all entries d_{ij} represent the length of the shortest path between nodes x_i and x_j.

Algorithm 4 Floyd's shortest distance algorithm

```
1: Initialize D_{n×n} with all d_ij = ∞.
2: for k = 1 to n do
3:    for i = 1 to n do
4:       for j = 1 to n do
5:          d_ij ← min{d_ik + d_kj, d_ij}
6:       end for
7:    end for
8: end for
```

The Matlab source code implementation of Floyd's algorithm is given in Appendix A.2.3.5. Here, the `age` field of the C (connection list) data structure is used to represent edge length. In this research, the distances between node pairs are set to ∞ (following execution of Floyd's algorithm in each time step) in cases where the action vector has yielded negative reward. This situation is indicative that the `ancestor` node for the current action vector should not have been emulated. Setting the distance to ∞ excludes that ancestor from consideration during the next time step.

3.4.4 Network Clumpiness

Although the method employed in learning a gesture-based command vocabulary described in this thesis makes no effort to label or otherwise classify input patterns, the use of the GNG network topology allows the learning process to benefit from the reward history of nodes within a neighborhood region. This is based on the intuition that gestures performed in a similar manner (and are thus close to one another in the GNG network) might be expected to elicit the same robotic response.

Frequently in classification problems, data of a given class may cluster nearer to the mean for that class than to the mean of an different class. Within a GNG topology, such clustering may be reflected in the degree (number of connections) of

a reference node. Given the unlabeled nature of the data used in this approach, the number of gesture commands represented in a network is unknown. Hence, a metric involving the degree of nodes and their relative distances from each other might be expected to serve as a measure of cluster centrality within nodes of a given gesture type. With this motivation, Estrada's *clumpiness* metric [33] is considered as a metric for use in the formation of node neighborhoods. A node's clumpiness characteristic relates the respective degrees of a pair of nodes within a network to their distance from one another. A clumpiness coefficient Ξ_{ij} for a given pair of nodes x_i and x_j may be computed according to (3.4):

$$\Xi_{ij} = \begin{cases} \dfrac{k_i k_j}{(d_{ij})^2} & \text{for } i \neq j \\ 0 & \text{for } i = j \end{cases} \tag{3.4}$$

where k_i is the degree of node x_i and d_{ij} is the network distance between nodes x_i and x_j as computed using Floyd's algorithm (section 3.4.3). It is shown in section 3.4.9.4, that although computationally intensive, clumpiness is highly effective as a means of selecting neighborhood nodes with action vectors likely to yield positive rewards.

3.4.5 Network Resistance Distance

The GNG algorithm provides an aging function for connections between nodes. If the age value (or other length metric) of a connection is interpreted as the electrical resistance of a conducting path between the pair of connected nodes, the *resistance distance* [62] between any pair of nodes may be determined. Total resistance between two points in a resistor network is a function of both the values of resistors involved and of the number of paths between the two points. The resistance between two points in such a network is always less than the shortest path distance if more than

one path between the two points exists.

Since the topology of the GNG network and the relative ages of connections is reflective of the frequency with which input gesture patterns fall into the receptive fields of a given reference node, the resistance distance between a pair of nodes may, similarly, be shorter than the shortest path length. The shortest path by resistance distance may follow a different route than Floyd's algorithm (section 3.4.3) would select. This metric is also considered in the formation of neighborhoods when selecting action vectors likely to yield positive rewards. The resistance distance Ω_{ij} between two nodes x_i and x_j in a connected network of n nodes is:

$$\Omega_{ij} = \begin{cases} L^+_{ii} + L^+_{jj} - 2L^+_{ij} & \text{for } i \neq j \\ \infty & \text{for } i = j \end{cases} \tag{3.5}$$

where L^+ is the Moore-Penrose generalized inverse of the graph Laplacian L. Normally, $\Omega_{ii} = 0$. However, since the goal in using the resistance distance matrix in this research is to determine potential neighbors, a node's distance to itself is set to ∞. The graph Laplacian is computed as:

$$L = K - Av \tag{3.6}$$

where Av is the admittance matrix of the network:

$$Av_{ij} = \begin{cases} 1/r_{ij} & \text{for } i \neq j \\ 0 & \text{for } i = j \end{cases} \tag{3.7}$$

and K is the degree matrix:

$$K = diag\left(\sum_{z=1}^{n} \frac{1}{r_{xz}}\right). \tag{3.8}$$

The `age` field of the C data structure is interpreted as the edge resistance r_{ij} between nodes x_i and x_j for this research. The source code implementation of the resistance distance calculation is given in Appendix A.2.3.9. Also, because computation of resistance distance requires that the GNG network remain connected, the maximum age of connections in the network must be set to a large value when this computation is used to prevent components of the network from becoming disconnected as older connections expire.

3.4.6 Simulation Environment

As with the experimentation in section 3.3, ROS Turtlesim is once again employed as a simulated proxy for a mobile robot to fully *close the loop* between a gestured command from a human user and a final, learned robotic actuation. Further, the dimensionality of the Turtlesim utility matches that of the envisioned ART environment of an assistive robot in a hospital patient room or home setting (see Figures 1.1 and 1.2). Hence, the scale and accuracy of the generated actions might aid the reader in visualizing the effectiveness of this approach.

3.4.7 Reward Generation

Generation of a reward signal may be expensive in terms of effort by the human participant. For this experiment, reward generation is automated in software according to the predefined goal configurations shown in Table 3.4. Once again, these configurations represent relative translations (x, y, θ) from the starting position of the

robotic agent $(0, 0, 0)$ and were chosen to be easily distinguishable.

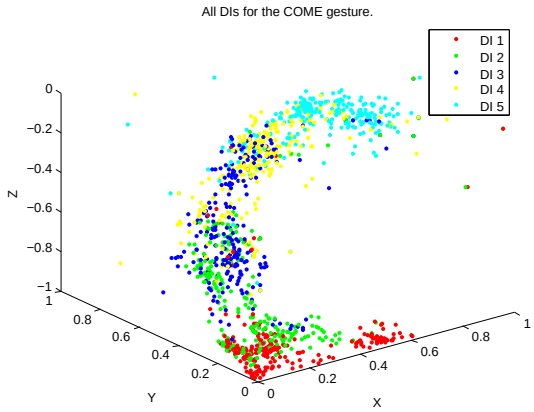
Table 3.4: 3D goal configurations for a simulated mobile robot.

Gesture Type	(x, y, θ)
Come closer	$(3.95,\ 3.95,\ 45^o)$
Go away	$(3.95,\ -3.95,\ 315^o)$
Stop	$(-3.95,\ -3.95,\ 225^o)$

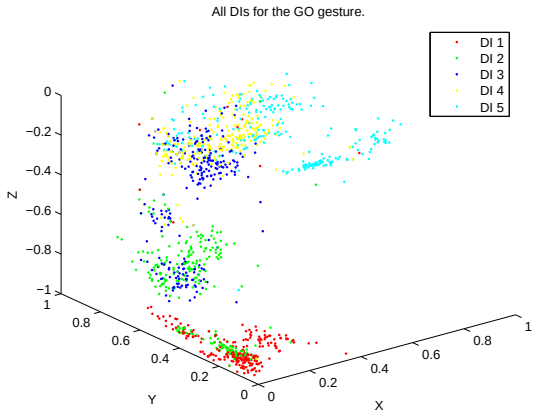
3.4.8 Experimentation

Using the collected data samples described in section 3.2.1, Dynamic Instants (DI) were computed for each gesture type. These are shown in Figure 3.10. Feature vectors based on these DIs will be termed the *real* data set so as to distinguish them from an artificial, idealized data set discussed below. It can be seen that the *real* data are not well separated and may not be expected to yield GNG neighborhoods which can be readily clustered by gesture type. The implications of the separability are discussed in section 3.4.9.

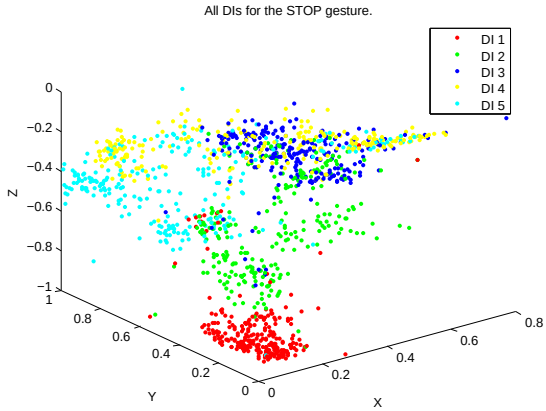
For the purposes of comparison and other experimentation, a second *ideal* data set was created based on a single exemplar gesture of each type. A collection of 750 samples was generated by the application of uniformly distributed noise within a margin of 5% of each of the DIs in each exemplar. DIs for the ideal data set are shown in Figure 3.11.



(a)

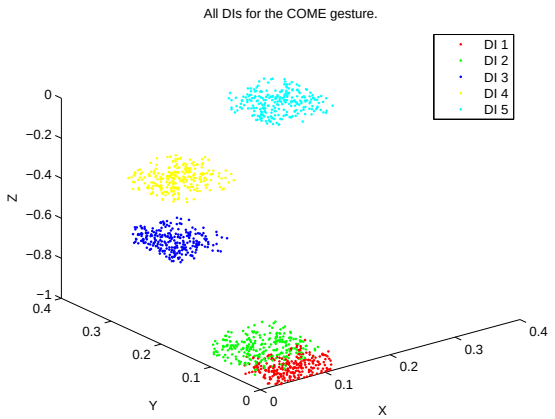


(b)

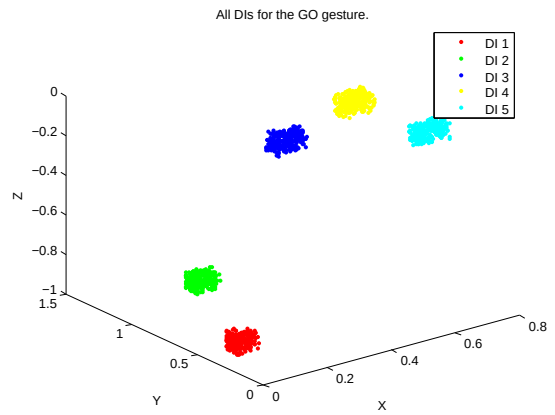


(c)

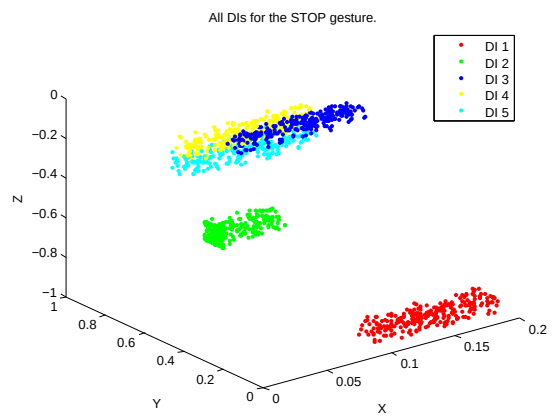
Figure 3.10: Dynamic Instants for *real* data samples (a) come closer, (b) go away, and (c) stop. Each color represents a specific DI (one of five) for the gesture type. Five DIs constitute a feature vector representation of a gesture motion as shown in Figure 3.3.



(a)



(b)



(c)

Figure 3.11: Dynamic Instants for *ideal* data samples (a) come closer, (b) go away, and (c) stop. Five DIs constitute a feature vector representation of a gesture motion as shown in Figure 3.3. Note: apparent differences in data spread for ideal data are due to scaling within the plots.

3.4.8.1 Neighborhood Formation

Six neighborhood scenarios were defined in order to explore various possibilities for exploiting the topology of the GNG cloud and to demonstrate the possible benefits to a neighborhood learning strategy afforded by GNG. Each scenario reflects a different manner for selecting neighborhood reference nodes whose actions vectors are candidates for the agent's next actions based on past rewards. Three additional scenarios were defined to demonstrate the effectiveness of GNG over kNN. These nine neighborhood formations are described in Table 3.5 below. In the table, the *winner* refers the GNG reference node closest to the input feature vector by Euclidean distance.

3.4.8.2 Data Processing

For each data set, (*real* and *ideal*), the 750 samples were randomly divided into two groups. One group consisted of 300 samples (100 of each gesture type) and was applied one sample at a time in order to train the GNG network to the topology of the input space with a low degree of error. A full application of all 300 samples was termed an *epoch*. Action vectors were not updated during this training phase and retained their initial values: $(x, y, \theta) = (0, 0, 0)$. Application of this first group was not a necessary step, though it facilitated smoother convergence during the learning of actions in the subsequent phase described next.

A second group of 450 samples (150 of each gesture type) was then applied in epochs and the learning of actions was allowed to proceed. For each of the 450 samples passed to the GNG algorithm, an action was selected and performed from among a neighborhood of nodes, reward was automatically generated and the reference node was updated accordingly. In this manner, 250 epochs were executed for each of the

Table 3.5: Neighborhood formation scenarios

Scenario Name	Description
Lone	Only the winner is considered.
Mean	Adjacent nodes within a mean connected distance of the winner are considered.
Large	All adjacent nodes are considered.
Floyd	The node with minimum network distance and Q value greater than the winner is considered.
Clumpiness	The node with the maximum clumpiness coefficient and Q value greater than winner is considered.
Resistance	The node with the minimum resistance distance and Q value greater than the winner is considered.
$k = 1$	The nearest feature vector (from a training set) to the input vector is considered. This scenario is analogous to the *Lone* scenario above.
$k = 3$	The three nearest feature vectors (from a training set) to the input vector are considered. This scenario is analogous to the *Mean* scenario above.
$k = 5$	The five nearest feature vectors (from a training set) to the input vector are considered. This is analogous to the *Large* scenario above.

neighborhood formation scenarios described in Table 3.5. For each applied gesture sample, the per sample error was calculated between the updated goal configuration and the known goal for that sample's gesture type. Following each epoch, the average error per gesture type was computed. This process was repeated for each of the nine neighborhood formation scenarios.

3.4.9 Results and Discussion

This section presents results in which the system was asked to learn desired outcomes for each of three candidate gestures: *come*, *go* and *stop*. Convergence

plots for each 250-epoch learning session are shown for each of the nine neighborhood formation scenarios (Table 3.5). Comparisons between results categories and their implications are discussed.

3.4.9.1 Results with Real vs. Ideal Data Using GNG

Plots of Dynamic Instants in Figures 3.10 and 3.11 show that the performance of gesture varies widely across the pool of participants. As in any pattern recognition problem, such variation decreases the separability of data and with it, the ease of classification. Hence, the need to generalize outcomes in the presence of such variation becomes vital to the construction of a robust recognition system. Figures 3.13 and 3.12 show average error plots using real and ideal data respectively for neighborhood scenarios *Lone*, *Mean*, and *Large*.

As may be expected, results for the ideal data set are shown to be more favorable in general, converging more quickly in all cases. Further, it is noted that increasing neighborhood size improves the speed of convergence when data are more separable, confirming the usefulness of neighborhood learning in a GNG context. This is indicative of well-defined clusters in the ideal data set and implies that the uniformity of performance of the actors will the strongly influence learning rate of the system.

However, given that the target user community may consist of unskilled or impaired users, such variation in performance is inherent. Given this quality of the real data set, larger neighborhoods scenarios are seen to cause slower convergence as neighborhood size increases (Figure 3.13). Note that this is the inverse relationship than was observed with ideal data. The boundary between gesture classes is not smoothly defined and the class regions overlap. Nevertheless, GNG performs robustly, converging in approximately similar timescales to as it did with unrealistic ideal data.

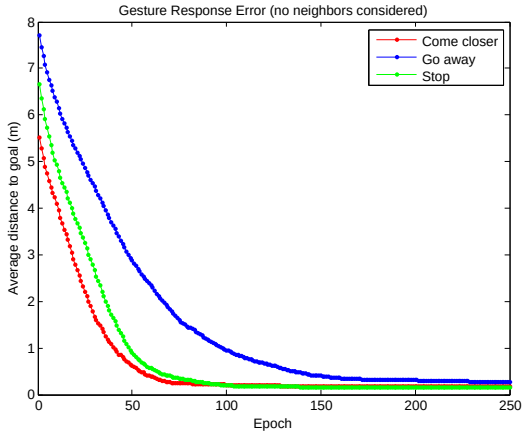
The capability of the GNG cloud to construct a topology which minimizes global error in the presence of noisy input is central to this outcome. This fact becomes especially apparent when comparing outcomes for GNG with those of kNN using real data in section 3.4.9.2.

3.4.9.2 Results with GNG vs. kNN

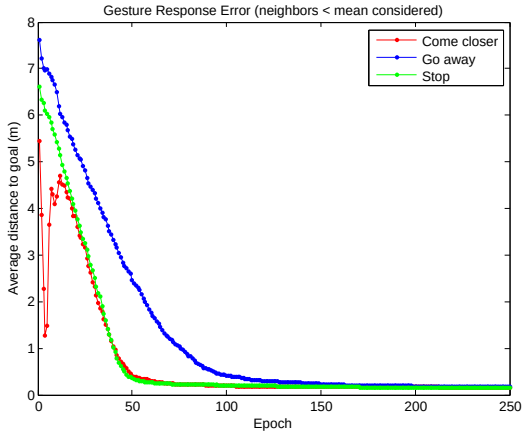
This section compares outcomes obtained using GNG with those obtained using kNN for three neighborhood scenarios using real data: *Lone*, *Mean* and *Large*. These neighborhood sizes are analogous to kNN neighborhoods of 1, 3, and 5 respectively. Also, among simulation results for kNN are those for two separate implementations of the kNN paradigm (Types 1 and 2 as described in section 3.4.2). It can be seen from the plots of Figure 3.14 that GNG outperforms kNN for all cases and implementations.

For Type 1 kNN, 300 training gesture samples were selected at random. As previously mentioned in section 3.4.2, this is not a practical scenario since randomly selected input can have no particular correlation with a human user's present actions. It is included here strictly for reasons of performance comparison only. Smaller numbers of training samples (e.g. 100 to emulate the maximum node count in GNG) were seen to converge very poorly or not at all. This is due to *competition* caused by neighboring nodes representing gestures of different classes. This problem is mitigated in GNG as reference nodes move in order to reduce global input error. Since this is not the case with kNN, global error remains high even as individual sample actions sometimes attempt (in futility) to learn multiple action responses simultaneously. Again, even for larger numbers of training data points, the performance falls far short of GNG. These results can be seen in Figures 3.14a-(c).

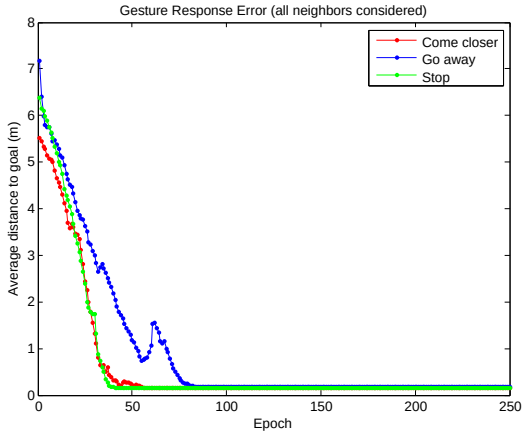
Type 2 kNN employs a history buffer of past inputs in order to influence



(a)

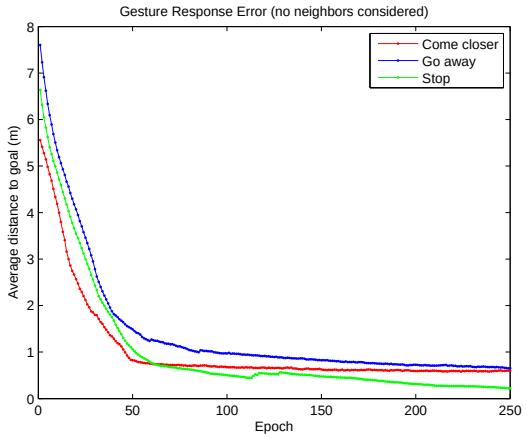


(b)

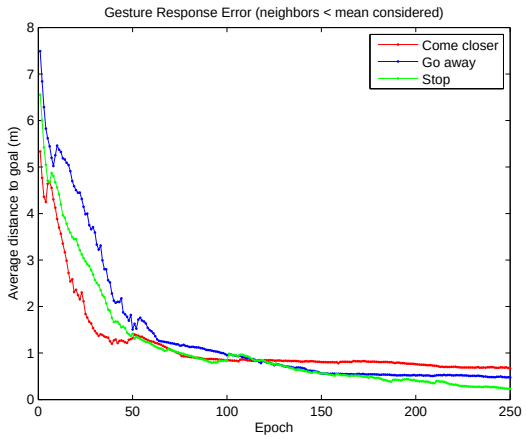


(c)

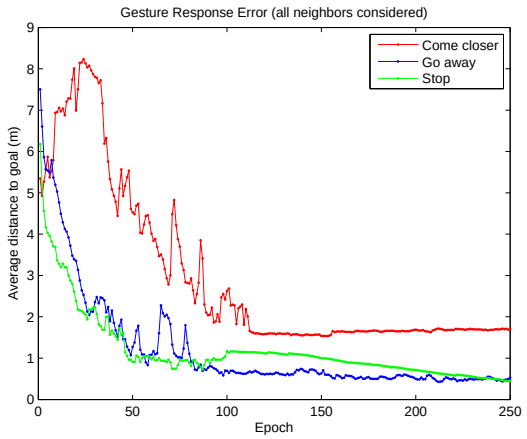
Figure 3.12: Average error curves for GNG with ideal data for neighborhood scenarios: (a) *Lone*, (b) *Mean*, and (c) *Large*.



(a)



(b)



(c)

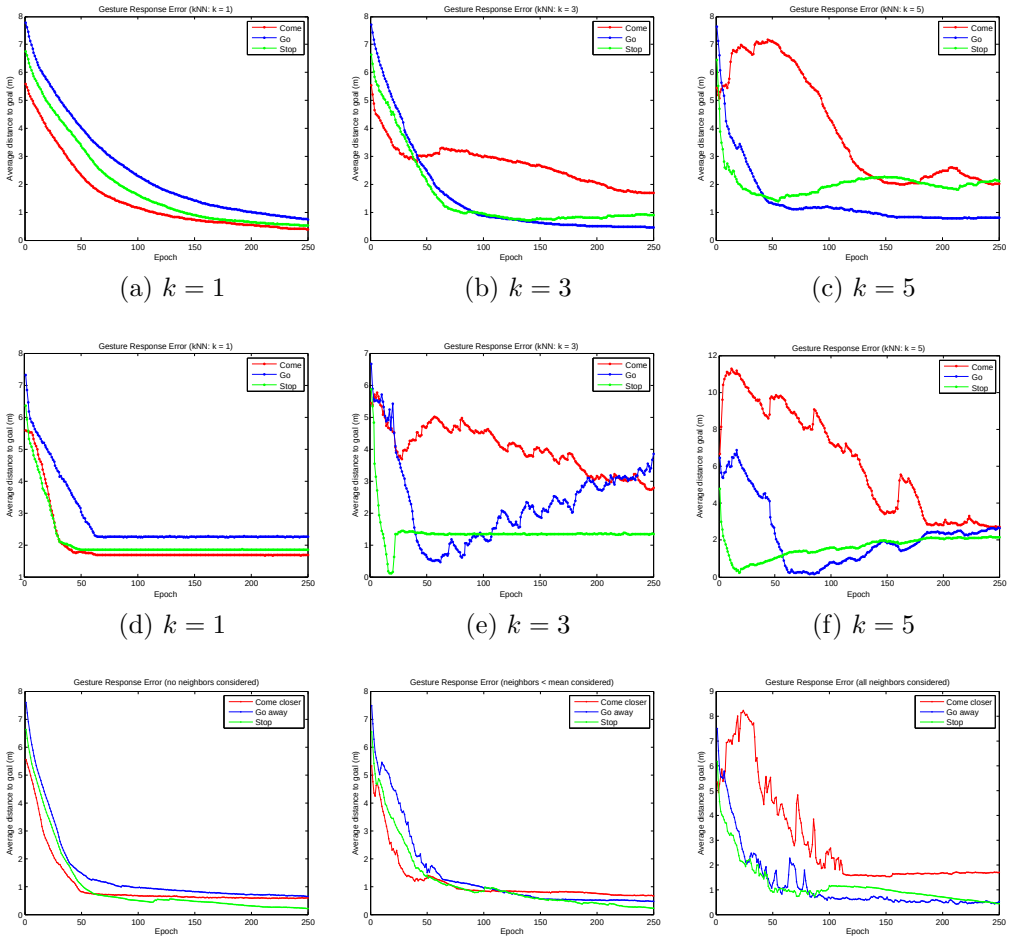
Figure 3.13: Average error curves for GNG with real data for neighborhood scenarios: (a) *Lone*, (b) *Mean*, and (c) *Large*.

future actions. This emulates the method used by Knox [63]. Neighbors are selected from this buffer. The size of the buffer influences the learning rate of the system. Larger buffers allow a finer-grained comparison for the purposes of neighbor selection. However, excessively large buffer sizes were seen to slow processing time prohibitively. A 100-element buffer was used in this experimentation to emulate the maximum node count imposed on the GNG algorithm. Although the contents of the buffer as a whole may steadily improve as new entries extend the positively rewarded actions of older elements, any isolated case or new gesture type is ultimately lost. Thus, kNN may be expected to learn single gestures well, but the prospect of augmenting its command vocabulary is impractical unless only a small discrete input state space is defined and which may be stored entirely. It can be seen in Figures 3.14d-(f) that the performance of kNN degrades severely as neighborhood size increases.

For reference, and to provide visual contrast between GNG and kNN, the previously discussed results for GNG across comparable neighborhood scenarios (*Lone*, *Mean*, and *Large* are shown in Figures 3.14g-(i).

3.4.9.3 Results with Floyd's Algorithm

Typical results obtained using Floyd's shortest distance algorithm yielded the results shown in Figure 3.15. These results qualitatively resemble those obtained using a neighborhood of *Mean* size. These results suggest that there is sufficient motivation to use the graph topology of the GNG network to seek action vectors from non-adjacent nodes. Also, as mentioned in section 3.4.4, Floyd's algorithm underpins the calculation of clumpiness discussed below.



(a) $k = 1$ (b) $k = 3$ (c) $k = 5$

(d) $k = 1$ (e) $k = 3$ (f) $k = 5$

(g) *Lone* (h) *Mean* (i) *Large*

Figure 3.14: Average error curves for kNN with real data: (a)-(c) Type 1, 300 training samples, (d)-(f) Type 2, 100-element history buffer, (g)-(i) results for GNG shown for comparison to kNN.

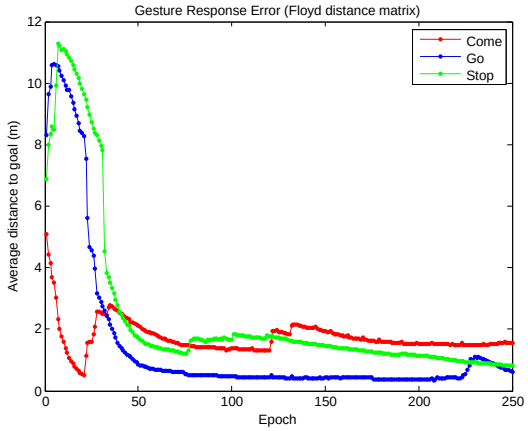


Figure 3.15: Average error curves for Floyd's shortest distance algorithm.

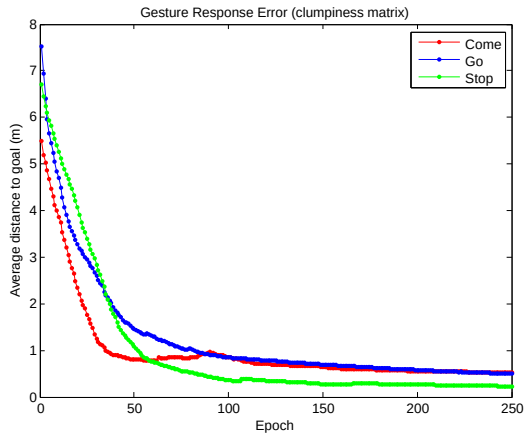


Figure 3.16: Average error curves for the clumpiness metric.

3.4.9.4 Results with Clumpiness

Typical results obtained using the clumpiness matrix yielded the results shown in Figure 3.16. These results qualitatively resemble those obtained using a neighborhood of *Lone* size. In general, the clumpiness metric yielded the best results for all neighborhood strategies tested. This is interpreted as evidence of the assertion that the clumpiness metric is successful in locating cluster centers in unlabeled data. Use of clumpiness in this manner is novel in the fields of pattern recognition and machine learning and particularly for the purpose of gesture learning as employed here.

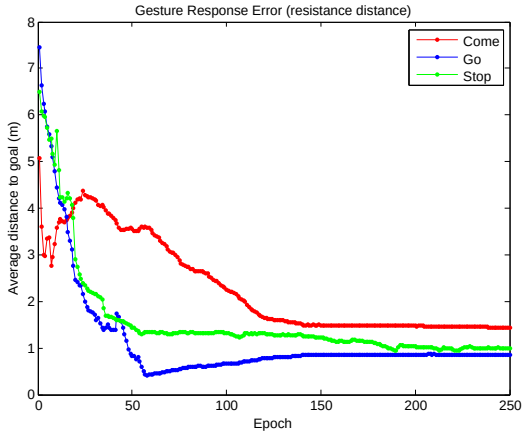


Figure 3.17: Average error curves for resistance distance.

3.4.9.5 Results with Resistance Distance

Typical results obtained using resistance distance yielded the results shown in Figure 3.17. These results qualitatively resemble those obtained using a neighborhood of *Large* size. As implemented here, the resistance distance metric is not an effective method for locating non-adjacent neighbors with positively rewarding action vectors. An improved future implementation would include a means of assigning connection reistances based on the success of neighbor-to-neighbor interactions. The requirement for the GNG topology to remain connected is seen as both computationally and functionally inefficient in that excessively old and essentially meaningless connections must be maintained. A means of moving connections while allowing the network to remain connected would be of benefit to this approach.

3.5 Summary

In this chapter, an innovative approach toward development of a gesture based human-machine interface has been presented. It has been shown that the Growing Neural Gas (GNG) algorithm is capable of differentiating between gesture descriptors

far more effectively than a conventional kNN approach. The GNG network topology coupled with Q-Learning supports neighborhood learning which effectively reduces the number of observations (and may thus, reduce the size of the data set) required for convergence. This finding is of key importance to the implementation of ART given the cognitive loading and physical limitations of the target user population.

Various strategies have been investigated for determining reference nodes from adjacent and non-adjacent neighbors. As a simple quantitative metric for the relative success of these strategies, the total accumulated error (for all gesture types combined) in each training session is shown in Table 3.6. These data were calcululated using (3.9).

$$E_{Session} = \sum_{epochs} \sum_{\substack{gesture \\ types}} E_{avg} \tag{3.9}$$

Because the exploratory aspect of Q-Learning utilizes randomization to choose possible future outcomes, the specific quantities shown may vary from run to run, though the general relationships are expected to hold. The clumpiness metric is shown to outperform all other methods. Use of clumpiness as a neighborhood search metric is novel in this problem space. Both Floyd's shortest path algorithm and the resistance distance approach yielded promissing reults. However, future work would be required to strategically *weight* connections in order to take advantage of these metrics.

Again, the TurtleSim environment is used to fully *close the loop* between a gestured command from a human user and a final learned robotic actuation. The dimensionality of the TurtleSim utility closely matches that of the envisioned ART environment. Hence, the scale and accuracy of the generated actions may help the reader to better visualize the effectiveness of our approach. Typical learned action

Table 3.6: Total accumulated error summary.

Scenario Name	Total Accumulated Error
Lone	864.24
Mean	934.64
Large	1444.73
Floyd	1423.32
Clumpiness	797.03
Resistance	1253.63
$k = 1$	1467.39 (Type 1), 1717.93 (Type 2)
$k = 3$	1456.47 (Type 1), 1935.56 (Type 2)
$k = 5$	1844.12 (Type 1), 2524.86 (Type 2)

trajectories can be seen in Figure 3.18.

An eventual use model for the gesture-base interface described here is envisioned in which the user performs a single gesture and provides repeated reward input so as to allow a GNG reference node to train fully. Investigations in chapter 4 will examine the possibility of using this approach to further reduce the number of gesture motion samples that are required. Also, online learning of new gestures will be explored. Certainly, for the envisioned system to effectively assist the user, the vocabulary of known commands must be open to amendment as needed.

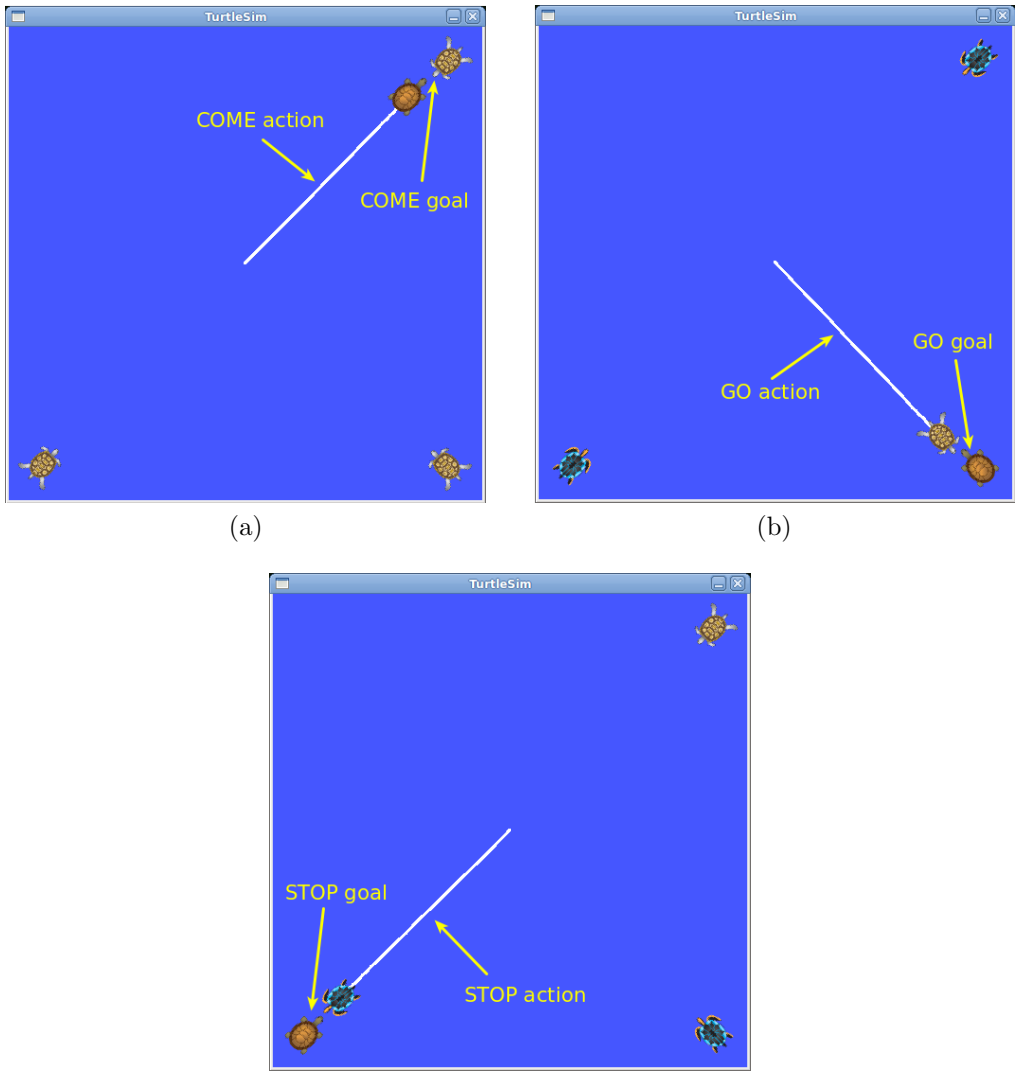


Figure 3.18: Learned action trajectories in TurtleSim for gestures: (a) Come closer, (b) Go away, and (c) Stop. Markers placed in upper right and bottom corners represent goal positions as shown. With a mature GNG-to-action mapping data structure, learned trajectories and final angles of approach accurately attain goal configurations.

Chapter 4

Gesture Vocabulary Augmentation

In order for the Assistive Robotic Table (ART) to be adaptive to the needs of its user, it must be capable of acquiring and learning new gestures during operation with the user acting as trainer, providing guidance as to their otherwise unknown goals and preferences. In chapter 3, refinement of the Growing Neural Gas (GNG) network's response output is achieved by applying input gesture samples randomly across the range of available gesture types and assigning rewards to the generated responses for each one. Although this procedure is useful as a means of demonstrating the efficacy of GNG to the gesture learning task, it is acknowledged that a human user would not likely undertake to train ART in this manner. Requiring the user to perform an assortment of gestures while also assigning reward to robotic responses at each time step would represent an undue physical and cognitive burden.

In this chapter, an alternative use/training model for ART is proposed which aims at reducing both the number of observations of a new gesture required to train ART to desired responses and the effort borne by the user in doing so. Experimentation is described which investigates the applicability of the GNG-based system to the learning of new gestures and to the retention of past learning.

4.1 Method

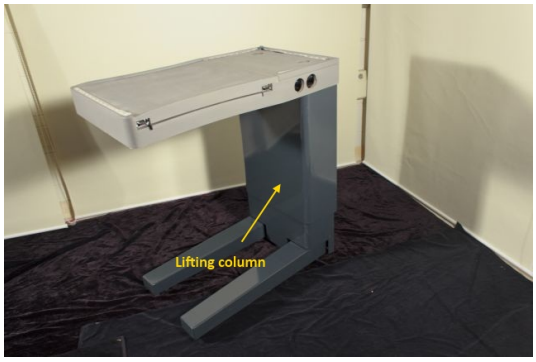
This section describes the gesture set used for experimentation and their relationship to the ART device. Toward the goals of reducing user effort and size requirements of the input data set, a new use model and training paradigm is detailed. Also, a new method for node insertion which preserves network stability while promoting the rapid learning of new gestures is proposed.

4.1.1 User-Centered Gestures

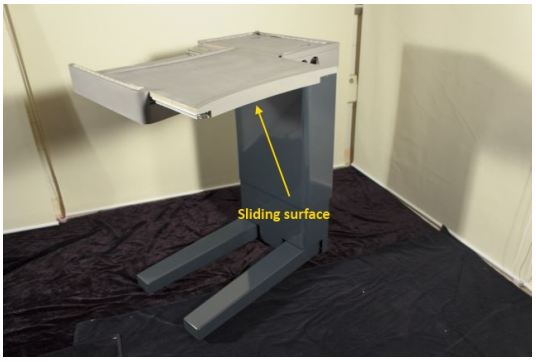
For the experimentation discussed in this chapter, six new gesture types are considered. These are selected with the user's intention in mind. While the gestures used in chapter 3 are direct commands to a robot agent (*come closer*, *go away*, *stop*), these new gestures are reflective of a more user-centered mindset and are broadly indicative of activities in which the user wishes to engage or to have the robot support. These include *eat*, *read*, *rest*, *take* (take an item away), *give* (bring an item closer) and *therapy*. Envisioned goal configurations for these these gestures are understood to exercise the three Degrees of Freedom (DOF) within ART shown in Figure 4.1. Their numerical values are mappings to points (x, y, θ) within the Turtlesim environment (see chapter 3) for simulation purposes. The qualitative labels and their respective mappings are given in table 4.1. Figure 4.2 depicts each configuration in a clinical setting.

4.1.2 A Use Model

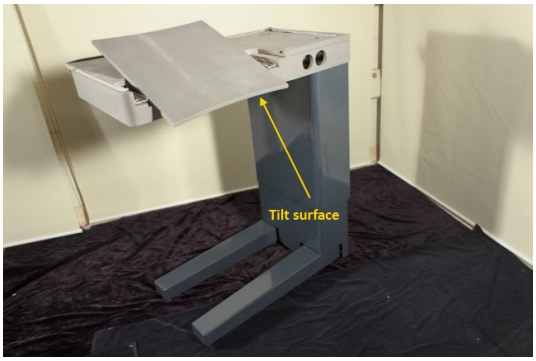
In this section, a use model is proposed which aims at reducing the physical burden to the user in terms of the number of training iterations required for the system to fully develop the desired actuation. In this model, the user demonstrates a



(a)

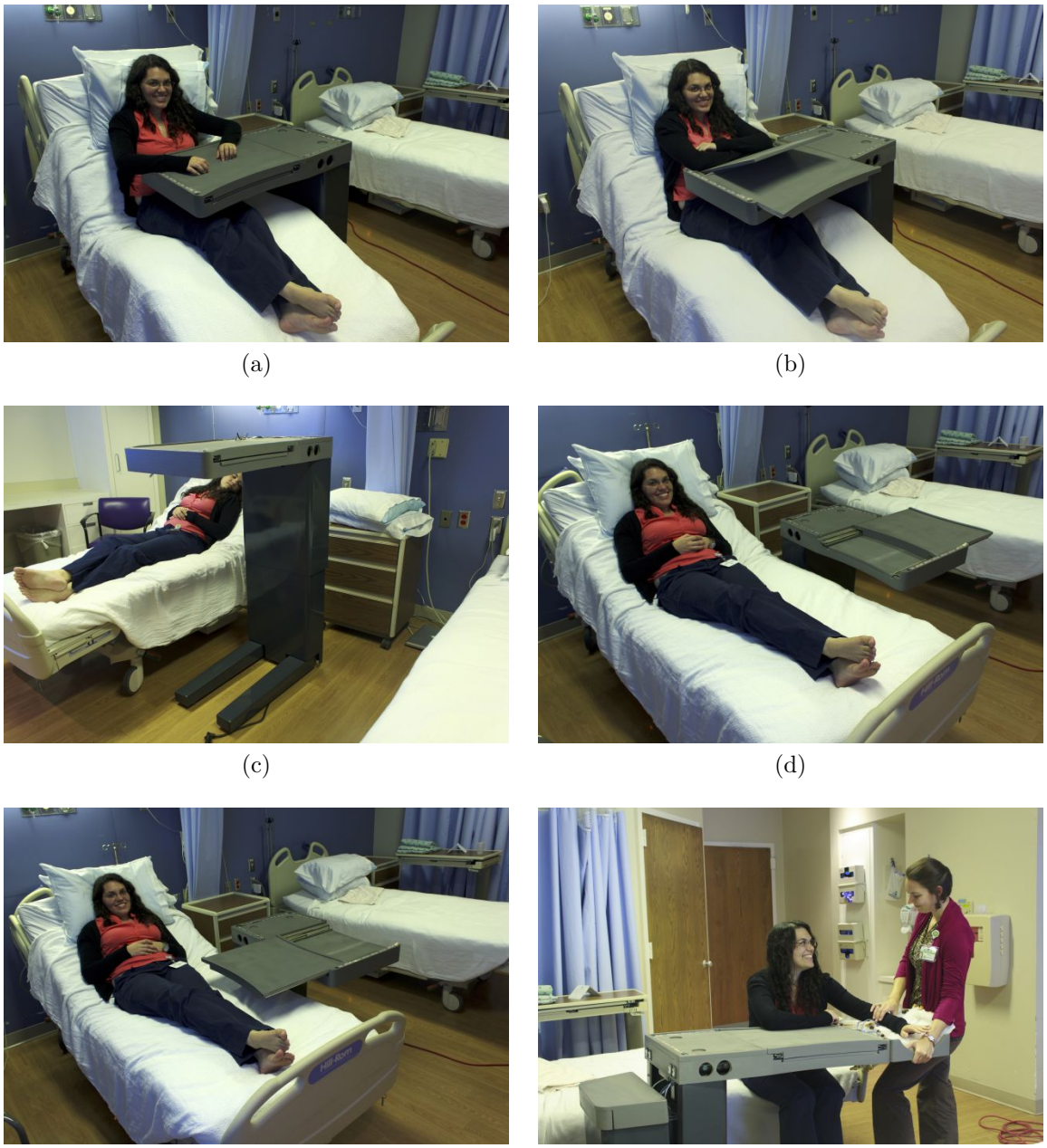


(b)



(c)

Figure 4.1: The three DOFs of ART: (a) the vertical lifting column, (b) the horizontal sliding table top and (c) the tilting work surface.



(a) (b) (c) (d) (e) (f)

Figure 4.2: 3D Configurations for ART in a clinical setting: (a) *eat* - the table surface is lowered to a comfortable dining height (b) *read* - the work surface is inclined, (c) *rest* - the table surface is raised (and moved aside), (d) *take* - the sliding surface is extended and away from the user, (e) *give* - the sliding surface is extended toward the user, (f) *therapy* - the table surface is at medium height to accommodate a therapy session.

Table 4.1: 3D goal configurations for ART.

Gesture Type	Lift	Slide	Tilt	Mapping in (x, y, θ)
Eat	Low	Center	Down	$(-3.95, 3.95, 135^o)$
Read	High	Center	Up	$(3.95, 0, 0^o)$
Rest	High	Center	Down	$(0, 3.95, 90^o)$
Take	High	Away from user	Down	$(-3.95, 0, 180^o)$
Give	High	Toward user	Down	$(0, 3.95, 270^o)$
Therapy	Middle	Center	Down	$(3.95, 1.98, 22.5^o)$

single sample of a new gesture to a system already trained to respond to some number of other gestures. The user then provides a series of consecutive rewards until the system is fully trained for that sample.

As in chapter 3, training (or, *path shaping*) consists of simple *good/bad* rewards assigned to incremental movements of the robot agent in response to the gesture. Movements toward a user-defined goal are assigned positive rewards. Movements away from the goal are assigned negative rewards. Gestures which, through training, elicit the full and complete action toward the user's goal are deemed *trained*. Upon completion of training for a given gesture, the Q-Learning policy for the state-action pair (the associated GNG node and its action vector) is frozen. Thus, any subsequent similar gestures whose feature vector fall into the receptive field for the same node require no further training. Across a given data set, this approach is shown to require a relatively small average number of training iterations.

4.1.3 Life-long Learning

Often, the operational life of a learning system is divided into the distinct phases of learning versus recognition. This paradigm neglects the possibility that the

system may need to acquire new recognition capabilities in the face of a changing input distribution from its environment. Conventionally, systems forced to consider new forms of input must reiterate the training phase. In so doing, they may suffer degradation in their ability to preserve knowledge acquired in the past. Thus, by extending their recognition capability, the stability of the system is compromised. This problem is termed the *Stability-Plasticity Dilemma* [45]. Toward the development of a system which can acquire new gestures as the user requires, the need for *life-long learning* is considered [48].

The plasticity of the GNG network lies in its ability to add and delete nodes during normal operation. The feature vectors of new nodes represent input patterns which differ from those seen in the past and the topology of the network is altered accordingly. Indeed, this feature of GNG is one of the primary motivations for its selection in this research. Fritzke [39] proposed the incremental augmentation of GNG based on the periodic assessment of local error at each node. The node with the largest accumulated local error is the node whose receptive field (or cell) is too large to adequately represent the distribution of inputs within the region and most in need of a new node to reduce the global error of the network. However, in this simple form, incremental learning may be overcome by the addition of a large number of nodes over time. This may result in both overfitting at overlapping cluster boundaries and excessive computing time. Alternatively, a maximum node count may be set which potentially limits network plasticity [48].

Fritzke [40] also proposed a utility-based approach (GNG-U) for the resource-conserving deletion of nodes in order to allow GNG to track non-stationary input distributions. However, in terms of life-long learning, this approach may remove nodes which represent past learning thus leading to instability. Hamker [48] proposed a method for stategic insertion of nodes using local error thresholds developed from

quality measures based on both long-term and short-term local error. The method was effective but focussed on supervised learning scenarios. Furao and Hasegawa [42] extend this work to focus on the insertion of nodes in unsupervised tasks. Their method attempts to assign unlabeled data to clusters autonomously before applying an adaptive similarity threshold based on cluster size. Input to an existing node is compared to the threshold to determine if it represents a new pattern class and is thus a candidate site for node insertion. The method also performs assessment to determine whether a particular insertion effectively reduced the network error in the long-term. Nodes which do not reduce the error are deemed ineffective and removed. This method, however, presupposes separable input distributions in order to place nodes in distinct clusters.

4.1.4 Learning with a Human Trainer

The presence of a human trainer poses a key difference between the methods in section 4.1.3 and that presented in this research. Here, input gesture samples are unlabeled and may not be well separated. However, using the proposed use model, the user-generated reward may be considered a binary *in-cluster/out-of-cluster* indicator. In the case of fully trained nodes, an input pattern which receives negative rewards when executing the action vector defined by that node must be of a different class. The location indicated by the input feature vector is interpreted as a likely good candidate site for node insertion. In the proposed approach, the local accumulated error of the winner in this case (the node nearest the input feature vector) is artificially inflated to the network maximum. At the same time, any nodes in the network whose most recent reward is negative (*cold* nodes) are considered for deletion. The `age` field for connections within the network may loosely be thought of as being indicative of a

node's nearness to a cluster center. A node with older-aged connections has previously been matched with fewer incoming patterns in those regions where its connections are oldest. When the network has reached a defined maximum number of nodes, the node with the highest sum of connection ages is targeted for deletion by the artificial *aging* of its connections to the maximum age limit. If the network is not at the maximum node count, then a new node may be added without deletion elsewhere in the network. In cases where all nodes in the network are either fully trained or are receiving positive rewards, new nodes may be added above the predefined maximum. This effectively relaxes the predefined maximum to afford plasticity when needed. This scheme for node insertion/deletion is summarized in Algorithm 5.

Algorithm 5 Node insertion/deletion algorithm

```
Apply a gesture input sample.
Determine winner reference node.
Observe reward.
if winner is trained and reward is cold or warm then
    Inflate local error: winner.E = max(refNode_i.E) + 1.
    if numNodes < maxNodeCnt then
        A node will be inserted near winner.
    else
        Locate a cold node for deletion.
        if A cold node exists then
            Inflate connection ages at the targeted node: C_i.age = ageMax + 1.
        else
            numNodes is allowed to increase beyond maxNodeCnt.
        end if
        A node will be inserted near winner.
    end if
end if
GNG will perform node insertion and deletion in the next time step.
```

4.2 Experimentation

This section describes the experimental data sets and the neighborhood scenarios in which they were tested. The experimental procedure sequence and key outcome metrics are also given.

4.2.1 Data Collection

Using the Kinect data collection fixture shown in Figure 3.2, five fellow-researcher participants each performed 50 repetitions for the six candidate gestures. This yielded 250 samples of each gesture for a total of 1500 samples. For repeatability, as in chapter 3, the gestures were performed according to their choreography in American Sign Language [6]. In consideration of the importance of data separability to the convergence of the GNG/Q-Learning algorithm, participants were encouraged to perform gestures as consistently as possible. Dynamic instants (DIs) were computed for each sample. Feature vectors were constructed from the DIs and presented to the system as described in section 4.1.2.

4.2.2 Data Sets

The 1500 gesture samples for the six candidate gestures were divided into two data sets. The *training data* set consisted of the gestures *eat*, *read* and *rest*. From these, a set of 450 samples (150 samples of each type) were selected and randomized. The second *test data* set consisted of the 750 samples of the gestures *take*, *give* and *therapy* sequenced randomly.

4.2.3 Procedure

The system was initially pre-trained using the training data set. The network was constrained to include 100 nodes. This was done in the manner presented in section 3.4.8 using the *Lone* neighborhood formation scheme (although, the choice of neighborhood formation method is not material to this experiment). This step yields A and C data structures. Once trained, these fully define a mature GNG network for the *eat*, *read* and *rest* gestures contained in the training set.

With the system pretrained, a single epoch either the test or training data sets (depending on which phase of the procedure was being conducted) was applied one sample at a time according to the use model described in section 4.1.2. Upon each presentation of a sample to the system, a simulation sequence was performed which included execution of GNG, generation of robot action, and assignment of reward. This sequence was repeated for the sample until one of three terminating conditions was reached:

1. The reference node closest to the input gesture sample became fully trained.

2. The input gesture sample received a negative reward in the receptive field of a fully trained node. In this case, the sample was immediately ignored and an additional node was inserted near the trained node according to Algorithm 5.

3. The number of training iterations exceeds 1000 (the *confusion threshold*). This indicates that the formed neighborhood is issuing conflicting action advice and the input sample is near a cluster boundary. In this case also, the sample was also ignored. However, the number of attempted learning iterations was considered in calculation of outcome metrics.

In this way, a 3-epoch sequence was conducted as described below. Following each

epoch, performance metrics were recorded. These included the total number of nodes in the GNG network, the number of fully trained nodes, the percentage of samples ignored, and the average number of training iterations per sample. These are chosen for their relationship to level of effort required by the user in training the system.

The sequence was conducted for five of the six neighborhood formation methods described in Table 3.5. The resistance distance metric, however, was not considered in this experiment. Computation of this metric requires that the GNG network remain connected. In instances where the insertion/deletion procedure of Algorithm 5 renders the network disconnected, the calculation would become unreliable. A method for artificially *reconnecting* network fragments during the node deletion step so as to preserve the usefulness of the resistance computation is left to future research.

4.2.3.1 Demonstration of Plasticity

With the system initially trained using the training data set, a single epoch of the test data set was applied. Execution of this epoch is intended to demonstrate the plasticity of the GNG network to learn the *take*, *give* and *therapy* gestures.

4.2.3.2 Demonstration of Stability of Past Learning

The training data set was reapplied in a single epoch. Execution of this epoch is intended to demonstrate the stability of the system learning implementation. If the implementation is indeed stable, the outcome would be expected to reflect an already-trained network. That is, the performance metrics would be expected to show iteration counts which remain tolerable to a human trainer.

4.2.3.3 Demonstration of Stability of New Learning

A final epoch of the test data was also executed. This step effectively reinspects the network for the stability of the additional *take*, *give* and *therapy* gestures introduced by the test data set in the first epoch. Results for this procedure are given in section 4.3. The source code implementation which executes each phase of this procedure is Appendix A.2.1.9.

4.3 Results and Discussion

Typical results for execution of the first epoch in which test data was applied is given in Table 4.2. Given that the GNG network was initially trained to the *eat*, *read* and *rest* gestures, application of the test data shows the plasticity of the network in learning new gesture types under the proposed use model.

Two metrics in particular are seen as key to evaluation of the proposed use model: (1) the percentage of samples ignored and (2) the average number of training iterations. As previously stated, samples may be ignored by either falling into the receptive field of a node that is already trained, or by simply taking too long to train (exceeding the confusion threshold of 1000 iterations). The rationale to ignore such *problem* samples is based on the assertion that non-action on the part of the robot is preferred to persisting with training and ultimately performing an undesirable action. Further, alteration of a previously trained action would negatively affect the stability of the system. Of course, this assertion is predicated on the user making a fairly accurate first attempt at performing the gesture. Thus, the priority for alteration of the network is set in favor of stability over attempting to adapt to a rapidly changing input distribution. Excluding the *Floyd* case, it can be seen from Table 4.2 that the percentage of samples ignored is small, averaging 10.0%.

The *Clumpiness* scenario ignores the fewest samples. This is interpreted as being coupled to the improved separability of the data set as participants were guided to perform gestures in a uniform manner. With well-defined clusters in the GNG network, the proximity of any given gesture input to the cluster center for its class is likely to have improved, while the distance between cluster centers will have increased. Thus, the clumpiness computation would be more apt to form its neighborhood from with its own class.

The average numbers of iterations is manageable in general, if still somewhat burdensome to the user. Also, those gesture samples which are ignored for having exceeded the confusion threshold will negatively impact this metric. The attempted iterations are not deducted from the total iteration count over the epoch and thus contribute to the average. Presently, no mechanism has been considered for detecting these situations before simulation proceeds to the threshold. Such a method is likely the subject of future work. After several gesture exemplar nodes of each class are fully trained within the network, the overwhelming majority of subsequent samples requires no training at all. Further, the average number of training iterations is seen to decrease further in subsequent epochs. In summary, these results demonstrate that the fully trained network which existed before the test data was first applied is capable of learning new gestures in a human-tolerable number of time steps. A method for detecting and abandoning training efforts which are not converging would contribute significantly to lowering the average number of iterations in a typical training cycle.

Table 4.3 shows results from an epoch in which training data was reapplied to the network having been newly trained with the test data gesture set. These results reflect the stability of the GNG implementation. Results which show less than tolerable average numbers of iterations or a large percentage of samples ignored are indicative of a network which has lost previous learning. The table shows, however,

Table 4.2: Results for application of new gesture types to a trained network.

Neighborhood Formation	# Nodes	# Trained Nodes	Samples Ignored (%)	Average Iterations
Lone	100	89	10.3	9.74
Mean	100	85	9.1	7.32
Large	101	85	12.4	31.07
Floyd	118	96	21.9	78.77
Clumpiness	100	93	7.3	8.89

that both the average number of iterations and the percentage of samples ignored from the training set are small for all neighborhood formation schemes. Also, the increase in the number of trained nodes reflects the ability of GNG to continuously adapt to changing input. Having learned many of the gestures in the test data set, the topology of the network has been altered. Still, some samples for which learning may have failed to converge during the initial training phase have now been learned.

Table 4.3: Results for re-application of training data following new gestures.

Neighborhood Formation	# Nodes	# Trained Nodes	Samples Ignored (%)	Average Iterations
Lone	100	91	6.8	4.20
Mean	99	91	4.9	5.62
Large	105	96	6.0	19.67
Floyd	111	102	9.3	2.46
Clumpiness	100	98	5.6	2.89

The final epoch underscores the stability of the system to remain stable through the reapplication of test data. Table 4.4 shows results from this scenario. Both the average numbers of iterations and the number of samples ignored have decreased

from the first application of this data set under all neighborhood formation schemes. However, multiple executions of this experimental procedure do not show a clear best strategy for selecting nodes under the applied learning paradigm and use model. Although, the *Clumpiness* method is frequently seen to ignore the fewest samples. Although not reported quantitatively here, subsequent epochs for either the training data or the test data frequently resulted in convergence at zero iterations per sample: the entire network had become fully trained.

The fact that the network may become, effectively, an associative memory with perfect recall of all input samples is problematic. The node insertion/deletion scheme of Algorithm 5 may insert nodes between clusters in regions where class decision boundaries overlap. The method, in its present form, will ultimately be guilty of *overfitting* the problem. It will have placed too fine-grained a generalization on the input space, causing it to behave poorly in situations where gesture choreographies closely resemble one another between classes. A more discriminating method for node insertion would need to be considered in order to better temper the system when handling poorly separated data.

Table 4.4: Results for re-application of new gestures.

Neighborhood Formation	# Nodes	# Trained Nodes	Samples Ignored (%)	Average Iterations
Lone	100	92	0.7	0.73
Mean	100	93	3.1	0.79
Large	105	99	1.9	1.36
Floyd	108	102	2.7	2.90
Clumpiness	101	100	1.2	0.97

4.4 Summary

In this chapter a use model has been proposed which considers the effort required by a human user to train ART to a desired behavior. The model aims at reducing the number of training iterations which must be performed by the user in order for ART to robustly learn a collection of gestured commands. As such, the effectiveness of learning is judged over single epochs which expose the system to single observations of each gesture samples. The primary metrics used to evaluate the learning process under the proposed model are the average number of reward iterations per sample and the number of samples ignored as unrecognizable by the existing GNG network topology.

For learning to proceed, the network topology must remain mutable in the presence of new input patterns. At the same time, however, experience gained from past learning must not be discarded. Hence, the Stability-Plasticity Dilemma is also addressed as part of this effort. An algorithm for altering the GNG network topology (by node insertion/deletion) in service of these learning and retention requirements is proposed. The algorithm exploits the fact that reward from the human trainer may be interpreted as indicator to the presence of a new gesture class.

Both the proposed use model and the node insertion/deletion algorithm are exercised in a simulated learning sequence which attempts to evaluate their respective efficacy. A collection of user-centered gesture types is employed to test the use model across multiple neighborhood formation schemes of the GNG network. The system is shown to learn well and to afford a tolerably small number of training/reward iterations per sample. Further, the percentage of samples ignored is small - reflecting an effective recognition rate by the network. With the exception of the *Floyd* method, all neighborhood formation strategies are shown to be feasibly applied to the learning

problem. The *Clumpiness* method exhibits a superior initial learning capability. This is attributed to the insertion of nodes for the newest gesture classes forming small clusters which are removed from existing clusters. Hence the relative clumpiness of the included nodes is greatest with respect to themselves. As these clusters expand, their nearness to other clusters may diminish the effectiveness of the metric. In these cases, best results will be seen for the *Mean* and *Lone* strategies.

Chapter 5

Conclusions and Future Work

In this thesis, the broad goal of creating assistive environmental components for individuals seeking to maintain independence as they age has been advanced. In pursuit of this goal, contributions in the area of Human-Robot Interaction have been made which seek to surmount common weaknesses in many existing interface design forms. Targeting its implementation as the command interface to the Assistive Robotic Table (ART), a novel method for learning arm-scale gesture with simple supervisory feedback from the user has been proposed.

The problem space which exists between the detection of physical gesture motions as input and the actuation of robotics which meet a specific user preference is quite large and is composed of several major areas of continued inquiry. This work has dissected the problem space into its constituent subproblems and presents a thorough consideration of each one: sensing, data representation, pattern recognition, and machine learning. The rationale behind the selections made in these areas for the proposed interface have been discussed and compared with other prevailing techniques. In the aggregate, this set of candidate solutions and the innovations by which the current states of the art have been extended herein present significant contribu-

tions to the development of assistive devices capable of learning from human teachers. This chapter discusses the implications and strengths of the approach and identifies opportunities for future work which would bring the interface design to fruition and further advance specific areas of underlying research.

5.1 Conclusions

In chapter 2, the issue of sensing gross motion patterns while preserving user privacy is addressed. Common camera-based or wearable sensing strategies present drawbacks in the areas of privacy and usability which preclude wide adoption by the target user population. It was shown that sparse signals available from simple privacy-preserving proximity or IR sensors are capable of providing a useful motion descriptor in the form of a Histogram of Gradients (HOG) computed over the Self-Similarity Matrix of the motion data stream. Visible commonalities of descriptor plots taken from multiple vantage points were shown to support Junejo's assertion [56] that prominent features of motion are view-invariant. This finding is further exploited in the experimentation of chapter 3. The findings of chapter 2 also reveal that recognition of activity approaches a practical level of accuracy when inputs from multiple vantage points are fused over a local region. This realization initiated the incorporation of the Kinect depth sensing camera into this research. The depth point cloud produced by the Kinect proved sufficiently rich to afford high recognition accuracy while adding a third image dimension without extra processing.

Using the joint tracking capability of the OpenNI package and the PrimeSense viewer in the experimentation of chapter 3, the task of capturing a sufficiently rich feature set from an anonymity-preserving sensor was readily achieved. With the available data, the concept of Dynamic Instants (DI) [84] of motion (extrema of

acceleration) were applied using 3D data for what is believed to be the first time. This technique is intuitively appealing since, as past research suggests [55], such extrema are the features by which human beings visually discern the activities of one another. The DI-based representation would be expected to allow arbitrary placement of the Kinect sensor. Results showed that the use of DI-based feature vectors as formulated cluster well for well-separated data. For poorly-separated data, the performance of the proposed system remains robust.

Also within chapter 3, the realms of pattern recognition and machine learning are unified to the direct (progressive) mapping of input (gesture) space to output (action) space. Typically, these are treated as separate problems. In the methodology used, however, unchoreographed gesture can have no label on which to base its classification. Rather, a positive or negative reward indication from the user as they observe the robotic response to gesture constitutes the only metric by which the mapping accuracy may be judged. Thus, accuracy (and speed) of mapping is key to the success of the system implementation. With this requirement in mind, the use of Growing Neural Gas in a Q-Learning framework is introduced.

Use of the Growing Neural Gas (GNG) clustering technique allows an efficient method for generalizing across the input space with a relatively small number of samples. Rather than requiring an extensive training phase, the unsupervised nature of GNG topologically maps the input space to discrete receptive fields surrounding a predefined number of reference nodes. Simultaneous application of simulated reward from the user permits the refinement of *output weights* directly applicable to the 3-DOF actions of the ART device. Further, the ability of GNG to quantize input space in a manner superior to the commonly used kNN algorithm is demonstrated. The use of a Q-Learning framework places the selection of action in the context of a state-action value function, $Q(s, a)$. Selection of state-action pairs by exploiting

the GNG network topology is shown to be effective as a means of foreshortening the learning process as described below.

Given that the paths taken by the envisioned robotic agent are shaped through human guidance, the number of learning iterations (gesture/reward cycles) must be sufficiently small as to be tolerated by the human trainer. Hence, opportunities to accelerate learning using the GNG network topology are exploited. Neighborhoods of nodes are formed in the region of a *winner* reference node. Action vectors of nodes within the neighborhood are considered so as to maximize future rewards. Nodes whose action vectors have the richest history of reward are considered. It is shown that the choice of neighborhood formation strategy can have a significant effect on the speed of learning convergence.

Various methods for composing node neighborhoods are investigated. These include groupings of adjacent nodes at varying connection lengths from the *winner* reference node. Other methods considered include the novel application of graph distance metrics found using Floyd's Shortest Path algorithm, the network clumpiness metric, and resistance distance. This work is the first application of these metrics to the task of machine learning for gesture recognition of which we are aware. It was shown that the clumpiness metric outperforms other methods investigated for poorly-separated data, although it is more computationally intensive. The `age` characteristic of GNG connections was used to emulate *length* for each of these metrics on the assertion that it is directly analogous to the frequency of node activation.

Although the resistance distance metric was seen as comparable to other tested metrics, its computation requires the GNG network to remain connected. Since the GNG network does not naturally remain connected for well-separated input distributions, maximum age limits on connections were disabled. Hence, rather than removing connections between clusters, the metric simply treated them as being connected by

a high-resistance connection. This measure introduced excess computation and potentially compromised results. Further, the edge weights used to represent resistance were assigned in a simplistic manner based on age. It is suspected that a more strategic assignment of connection weights based on past rewards could allow the resistance distance metric perform to strongly in this problem space.

In chapter 4 the problem of learning new gestures on line is examined. A use model which envisions the training process that an actual user might find tolerable in terms of physical demands and duration is presented. The ability of the GNG network to learn new gestures while retaining past knowledge (the Stability-Plasticity Dilemma) is addressed. A novel algorithm which exploits the availability of human-generated reward to strategically insert and delete nodes in the GNG network is proposed. The algorithm is shown to facilitate the learning of new gestures given a previously trained network (plasticity). It was also shown to continually reduce its rate of non-recognition on subsequent presentations of familiar gestures (stability). Further, the number of trained nodes in the network is increased such that the gesture set is recognized (as evidenced by positively rewarded action) with very low average numbers of iterations on the part of the human trainer. Typically, the algorithm can be seen to *fully train* so that it represents an associative memory of all gesture types to which it has been trained. This outcome may be desirable for an individual user who is content with a small, uniformly executed gesture command set. Indeed, it is seen as likely to be effective for most applications. It is noted, however, that the learning of erratically performed or larger numbers of gestures may be compromised by a network topology that is overfitted to a particular command set. The proposed algorithm may insert nodes where clusters overlap, thus creating a ragged decision boundary where learning may be slow or unable to converge. Extended use of clustering-oriented recognition techniques, especially where plasticity of learning is desired, will require

future work toward strategic node insertion.

5.2 Future Work

Although the overarching goal of creating a command interface for an assistive robot is rather clear cut as a concept, the depth of the problem as it was considered in this thesis proved a great deal more formidable than initially perceived. When the work described in this thesis was first undertaken, the sheer volume of challenges which would be encountered was unforeseen. Thus, this thesis has attempted to draw clear boundaries around the major facets of the problem which will allow for continued innovation with a more compartmentalized view of the problem. This section discusses several areas of inquiry which were inspired by the work of this thesis and which are integral to the advancement of the ART vision.

- **Baseline Capabilities.** As mentioned in section 1.3.2, creation of a system in which the user plays a role in forming an agent's behavior would benefit from a set of baseline capabilites on which new learning could be based. The system proposed here makes no assumption of existing knowledge prior to instruction. However, in the process of learning the robotic responses as they have been defined, various social aspects of trajectory assessment (speed of movement, angle of approach, path selection and proximity with respect to the user, etc.) and consideration of the user's personal sensibilities would certainly be required for any practical implementation of ART.

- **Evaluation with Human Participants.** The use model presented in chapter 4 represents a supposition of how the user might prefer to conduct a gesture-based training session. This is also true of the larger gesture-based interface

concept. Presently, details of how the user perceives and interprets the action of a machine which reacts to gesture is unknown. Extensive interviews and testing with human participants are called for to judge the efficacy of the ART implementation. Of course, the approach described in this research may effectively learn from a simulated user. However, if the human user is somehow unwilling to use the device based on visceral reaction, illegibility of the implementation, complexity, or undue physical burden, the interface may not be considered successful. The human factors involved must be considered.

- **Gesture Segmentation.** Detection of gestures in continuous time (gesture *spotting*) is seen as essential to a practical implementation of a gesture-base command language. Various methods have been advanced [3, 41, 108], chiefly focused on 2D motion video of gestures performed under highly controlled circumstances. Spotting of gestures in unconstrained free motion is largely an unsolved problem requiring further research.

- **Variable Autonomy.** One aspect of the approach described in this thesis which presents a notable drawback is the need for the user to assign reward to small uniform increments of motion by the robot agent. Clearly, this process would require extreme patience from the user to complete a training session. Enabling the agent to make intermediate decisions on its own for extended periods during training (*variable autonomy*, e.g. [60]) would reduce the required vigilance of the user. Moreover, the agent's motion trajectories could proceed without interuption until it received a corrective action (negative reward) from user. This possibility represents a direct extrapolation from the approach set forth in this thesis.

- **Qualitative gestures.** Gestures considered in this research have been essentially *absolute* in nature, having been defined as a combination of translation and rotation with respect to a global origin. For a gesture-based system to acquire a *feel* that is comfortable to a human user, qualitative gestures must be defined. Examples might include: *more*, *less*, *stop*, *faster*, *slower*, etc. It has been shown that when humans are asked to teach robots, they are susceptible to treating the agent as though it would benefit from encouragement or motivation [95] such as might be inferred from application of these examples. Mastery of such a class of gestures could thus be considered a type of reward in a reinforcement learning context which supplements or replaces the binary reward structure used here. Handling of such *relative* gestures would represent an interesting departure from the proposed action mapping paradigm.

- **Training of Response Sequences.** In this thesis, user goals have been defined as static final configurations of ART. However, such responses could, by the extension of the learning paradigm discussed here, develop a library of response primitives. Such primitives could be connected sequentially to perform higher level tasks.

- **Detection of User Performance Variation.** A primary element of the ART vision is the ability to assistively support individuals whose needs are changing (and possibly degrading) over time. In terms of the gesture interface implementation, this ability implies periodic analysis of the *shapes* of GNG network clusters. Slow moving changes in the distribution of network nodes may be indicative of declining health of the user. An ability to spot gradual changes of this nature would help to elevate ART from assistant to companion and potentially prevent catastrophic outcomes for the user.

- **Multimodal Interaction.** Introducing and integrating other modes of interaction (voice, gaze, affect, etc.) with gesture is an obvious long term direction for HRI. These, in their own right, represent distinct areas of research. However, the framework presented in this thesis is expected to provide a mechanism for their inclusion as additional sensed environmental events which can be clustered according to their relative salience with the user.

- **Dealing with Imperfect Rewards.** Simulations in this thesis have assumed the presence of a perfect human trainer. All rewards have been issued with a distinct goal in mind and were always faithful to that goal. In simulation, the determination of whether an agent's action resulted in motion toward or away from the goal is a simple matter. In practice, however, the human trainer will most likely not be able to visualize the present state of ART's configuration as a Cartesian space. Even if one were capable of such a leap, simple human error would eventually enter in. Higher degrees of freedom would certainly exacerbate the problem. Research which brings statistical or fuzzy methods to bear in predicting the veracity of rewards would likely increase the speed of learning convergence in practice.

5.3 Summary

The key contributions of this research include both a broad formulation of a gesture learning framework as well as necessary advances in specific problem areas. This thesis has examined the essential functional areas of the ART interface concept in depth. The definition of the problem space in the areas of sensing, data representation, pattern recognition and machine learning has addressed the salient concerns and limitations in each one. Robust candidate solutions in each area have been chosen

which advance the goals of an intuitive interface based on gesture, and of the broader ART vision. Innovations in the areas of accelerated learning through network topology analysis and plasticity have facilitated an approach capable of learning from a human trainer while minimizing physical and cognitive burden. It is foreseen that the research presented in this thesis will constitute a foundation on which future work in assistive robotics and the design of adaptive interfaces may be based.

Appendices

Appendix A

Source Code

A.1 C++ Code

This appendix includes a C++ implementation of the system described in chapter 3. Readers wishing to execute this code must first install ROS [88]. The *Electric* release of ROS was used for this research. This code does not implement the node insertion/deletion algorithm, policy freezing for trained nodes or resistance distance (see the Matlab implementation in Appendix A.2 for these). Top level programs include:

- `gestureLrnList.cpp`: This is the main program used in the experimentation of chapter 3. It trains the GNG network based on gesture samples randomly applied in epochs.
- `gngTrain.cpp`: This program is used to learn the topology of the input space without training action vectors for each node.
- `getSkelData.cpp`: This program is used to collect data using the Kinect sensor.
- `genDI.cpp`: This program generates dynamic instants (DI) from ROS `.bag` files.
- `turtleControl_server.cpp`: This program executes commands to Turtlesim. It is not necessary to use this program unless Turtlesim is specifically desired for visualization. The `moveTurtle` function must be uncommented in `gestureLrnList.cpp` in order to use it.
- `turtleControl_client.cpp`: This program is used to generate commands to Turtlesim. It is not necessary to use this program unless Turtlesim is specifically

desired for visualization. The `moveTurtle` function must be uncommented in `gestureLrnList.cpp` in order to use it.

A.1.1 gestureLrnList.h

```
#include "kinect_includes.h"

// This file contains functions which support the top level
// calling routine for the gestureLrn(List) program.

// ----------------------------------------------------------------
// Function name:  findNearHood
// Description:  This function computes a near neibhorhood of
// reference nodes based on the goal vectors of the nodes.  Nodes
// outside a given distance are removed from consideration since
// they might be part of a differenct cluster in the GNG cloud.
//
// 6/30/2012 - The distance threshold is 2 standard deviations.
// Consider adding a distance function later.
// ----------------------------------------------------------------
std::vector<refNode> findNearHood(std::vector<refNode> &N, refNode &NN)
{
  std::vector<double> distances(0);
  std::vector<refNode> nearHood(0);
  int numNodes = N.size();
  double dist = 0;
  // double sigma = 0;
  double mean_distance = 0;


  // Estimate Q values
  // Consider adjusting theta values on [0,2pi] here.
  NN.Q = calc_Q(NN);

  // Compute a vector of distances from the NN.
  for (int i=0; i<numNodes; i++)
  {

    // Estimate Q values.
    N[i].Q = calc_Q(N[i]);
    dist = vecNorm2(NN.featureVec, N[i].featureVec);
    distances.push_back(dist);
  }

  mean_distance = calc_mean(distances);
  // sigma = calc_stdDev(distances);


  // Build the near neighborhood.
  for(int i=0; i<numNodes; i++)
  {
    // if (distances[i] < mean_distance)
    if (distances[i] <= mean_distance)
    {
      nearHood.push_back(N[i]);
    }
  }


  // Everything below is for debug purposes.

  /*
  printf("\n");
  printf("numNodes = %d, NN = %d, mean = %8.5f.\n",
    numNodes, NN.nodeLabel, mean_distance);
  for (int i=0; i<numNodes; i++)
  {
    printf("Node %d, distance = %6.3f\n",
      N[i].nodeLabel, distances[i]);
  }
  // printf("\n");
  printf("Pushing to nearN: \n");
  for (int i=0; i<numNodes; i++)
  {
    if (distances[i] <= mean_distance)
    {
    printf("%d, ", N[i].nodeLabel);
    }
  }
  printf("\n");
  */
```

```
  return(nearHood);

}

// ------------------------------------------------------------------
// Function name:  findClosestAngle
// Description:  This function selects an action from the nearest
// neighbors of a node which has higher Q and smallest angle w.r.t.
// the current action angle.  The node is returned.
// ------------------------------------------------------------------
refNode findClosestAngle(std::vector<refNode> &N, refNode &NN)
{
  int numNodes = N.size();
  double a_dot_b;
  std::vector<double> angles(0);
  double angle;
  refNode bestN;
  double mag_product;

  // Calculate angles between NN and all elements of N.
  NN.Q = calc_Q(NN);
  for (int i=0; i<numNodes; i++)
  {
    a_dot_b = (NN.action.x * N[i].action.x) +
              (NN.action.y * N[i].action.y) +
              (NN.action.theta * N[i].action.theta);

    N[i].Q = calc_Q(N[i]);
    mag_product = NN.Q * N[i].Q;
    angle = (mag_product == 0) ? 0 : acos( a_dot_b / mag_product );

    angles.push_back(angle);
    // printf("Angle between nodes %3d and %3d is %6.3f\n",
    //  NN.nodeLabel, N[i].nodeLabel, angle);
  }

  // Find min angle
  double minAngle = 9999;
  int minAngleLabel = 9999;
  bool foundOne = false;
  for (int i=0; i<numNodes; i++)
  {
    if ( (N[i].reward==1) && (N[i].Q > NN.Q) )
    {
      if (angles[i] < minAngle)
      {
        minAngle = angles[i];
        minAngleLabel = i;
        foundOne = true;
        // printf("FOUND ONE:  node %d\n", N[i].nodeLabel);
      }
    }
  }

  if (foundOne == false)
  {
    bestN = NN;
  }
  else
  {
    bestN = N[minAngleLabel];
    // printf("Choosing %d\n", bestN.nodeLabel);
  }

  return(bestN);

}

// ------------------------------------------------------------------
// Function name:  printN
// Description:  This function prints the set of neighbors (N) of the
// nearest neighbor (NN) for debug purposes.
// ------------------------------------------------------------------
// Print out N and NN if needed.
void printN(std::vector<refNode> &N, refNode &NN)
{
  int numNbrs = N.size();

  // printf("Printing the %d Neighbor(s) of NN (node %d):\n", numNbrs, NN.nodeLabel);
  printf("Node %3d: x = %6.3f, y = %6.3f, t = %6.3f, rwd = %2d, lastx = %6.3f, lasty = %6.3f, lastt = %6.3f (NN)\
    n",
    NN.nodeLabel, NN.action.x, NN.action.y, NN.action.theta, NN.reward, NN.last.x, NN.last.y, NN.last.theta);

  for (int i=0; i<numNbrs; i++)
  {
    printf("Node %3d: x = %6.3f, y = %6.3f, t = %6.3f, rwd = %2d, lastx = %6.3f, lasty = %6.3f, lastt = %6.3f\n",
```

```
      N[i].nodeLabel, N[i].action.x, N[i].action.y, N[i].action.theta, N[i].reward, N[i].last.x, N[i].last.y, N[i].last.theta);
  }

} // printN



// ------------------------------------------------------------------
// Function name:  genNeighborhoodResponse_xyt
// Description:  This function uses the set of neighbors (N) of the
// nearest neighbor (NN) to produce a response vector.   The generated
// response assumes a feedback vector with only a single dimension in
// {-1, 0, 1}.  The neighbor is selected based on a feedback value of 1
// and the longest length (Q value) of any such neighbor.
// ------------------------------------------------------------------
void genNeighborhoodAction_xyt(refNode &NN, std::vector<refNode> &N,
  std::vector<refNode> &A, std::vector<connection> &C,
  const double error_tol, int hoodRadius)
{

  std::vector<refNode> nearN(0); // the nearer neighborhood of NN.

  // Increase movement in direction of neighborhood response.
  double step_size = 0.1;

  // Reward values:
  int hot = 0; // on target (terminal state)
  int warm = 1; // closer than before.
  int cold = -1; // farther than before.
  double angle_delta = M_PI/18; // 10 degrees
  double huge_distance = 9999999; // inf

  // double r = 0, p = 0; t = 0; // Sherical coordinates (r,phi,theta)
  double x=0, y=0, theta=0; // Cartesian coordinates (x, y, z)

  // Print out NN and N for a sanity check.
  // printN(N, NN);

  // Choose neighborhood radius:
  N.push_back(NN);
  if (hoodRadius == 7)
  {
    // kNN - not populated yet.
    printf("kNN is not a valid hood radius\n");
  }
  else if (hoodRadius == 6)
  {
    printf("Resistance distance is not populated in C++.\n");
    /*
    // Resistance distance - may not work in C++
    // Use the MATLAB implementation for now.
    std::vector<std::vector<double> > Omega(0);
    printf("hoodRadius6 1\n");

    // Compute the resDist matrix.
    Omega = resDist(C);
    printf("hoodRadius6 2\n");

    int thisNode = NN.nodeLabel;
    int maxNodeLabel = getMaxNodeLabelC(C);

    // Find the lowest resistance.
    double minVal = 99999999;
    int minLabel = -1;
    for(int i=0; i<=maxNodeLabel; i++){
      if (Omega[thisNode][i] < minVal){
        minVal = Omega[thisNode][i];
        minLabel = i;
      }
    }

    // Assign nearN to be the node with lowest resistance.
    for (int z=0; z<=maxNodeLabel; z++){
      if (A[z].nodeLabel == minLabel){
        nearN.push_back(A[z]);
      }
    }

    // Add NN to the neighborhood if it is not already there.
    int nearN_size = nearN.size();
    if (nearN_size == 0) {
      nearN.push_back(NN);
    }
    else if (NN.nodeLabel != nearN[0].nodeLabel) {
      nearN.push_back(NN);
    }
```

```
249     */
250
251   }
252   else if (hoodRadius == 5) // Clumpiness
253   {
254     std::vector<std::vector<double> > ClumpMat(0);
255     ClumpMat = clumpiness(A,C);
256
257     int thisNode = NN.nodeLabel;
258     int numNodes = A.size();
259
260     // Find the node with max clumpiness for NN.
261     double maxVal = -99.9;
262     int maxNodeLabel = -99;
263     int thisLabel = -99;
264     for(int i=0; i<numNodes; i++) {
265       thisLabel = A[i].nodeLabel;
266       if (ClumpMat[thisNode][thisLabel] > maxVal) {
267         maxVal = ClumpMat[thisNode][thisLabel];
268         maxNodeLabel = thisLabel;
269       }
270     }
271
272     // Add the node with max clumpiness to nearN
273     for(int i=0; i<numNodes; i++) {
274       if (A[i].nodeLabel == maxNodeLabel) {
275         nearN.push_back(A[i]);
276       }
277     }
278
279     // Add NN to the neighborhood if it's not already there.
280     int nearN_size = nearN.size();
281     if (nearN_size == 0) {
282       nearN.push_back(NN);
283     }
284     else if (NN.nodeLabel != nearN[0].nodeLabel) {
285       nearN.push_back(NN);
286     }
287
288
289   } // end Clumpiness
290   else if (hoodRadius == 4) // Floyd
291   {
292     int thisNode = NN.nodeLabel;
293     std::vector<std::vector<double> >DistMat;
294     DistMat = floyd(A,C);
295
296     // Search all nodes with reward = 1.
297     // Choose the nearest one that is longer than
298     // the current Q.
299
300     int Arows = A.size();
301     int D_index = -1;
302     for (int z=0; z<Arows; z++) {
303       // Rule out nodes with reward = -1 by making their
304       // distance in D equal to infinity.
305       if (A[z].reward == -1) {
306         D_index = A[z].nodeLabel;
307
308         DistMat[D_index][thisNode] = huge_distance;
309         DistMat[thisNode][D_index] = huge_distance;
310       }
311     }
312
313     // Rule out nodes with shorter Q value than NN.
314     for (int z=0; z<Arows; z++) {
315       // Recalculate Q
316       A[z].Q = calc_Q(A[z]);
317
318       if (A[z].Q <= NN.Q) {
319         D_index = A[z].nodeLabel;
320         DistMat[D_index][thisNode] = huge_distance;
321         DistMat[thisNode][D_index] = huge_distance;
322       }
323     }
324
325     // Find the nearest remaining node.
326     double minDist = huge_distance+1;
327     int D_size = DistMat.size();
328     int minNodeLabel = -1;
329     for (int z=0; z<D_size; z++) {
330       if (DistMat[thisNode][z] < minDist) {
331         minDist = DistMat[thisNode][z];
332         minNodeLabel = z;
333       }
334     }
335
```

```
    // Assign the minNode to nearN.
    for (int i=0; i<Arows; i++) {
      if (A[i].nodeLabel == minNodeLabel) {
        nearN.push_back(A[i]);
      }
    }

    // Add NN to nearN if not already there.
    int nearN_size = nearN.size();
    if (nearN_size == 0) {
      nearN.push_back(NN);
    } else if (nearN[1].nodeLabel != NN.nodeLabel) {
      nearN.push_back(NN);
    }

  }
  else if (hoodRadius == 3)
  { // Use all neighbors
    nearN = N;
  }
  else if (hoodRadius == 2)
  { // Use neighbors within a radius.
    nearN = findNearHood(N,NN);
  }
  else if (hoodRadius == 1)
  { // Only use the node itself.
    // printf("Got Here.\n");
    nearN.push_back(NN);
  }
  else
  {
    printf("Bad Scenario.\n");
  }

  int numNeighbors = nearN.size();

  if (NN.reward == hot)
  {
    // printf("HOT=%3d ", NN.nodeLabel);
    // Do nothing.
  }
  else
  {

    // Find the near-neighborhood member with
    // max Q and reward = 1 (if it exists).
    int maxQ = -99;
    int maxQnode = -99;
    pt3sph spt;
    pt3 cpt;
    int this_reward = -99;
    config2D action;
    double thisQ = 0;

    for (int i=0; i<numNeighbors; i++)
    {
      if (nearN[i].reward == cold){
        action = nearN[i].last;
      } else {
        action = nearN[i].action;
      }

      thisQ = sqrt( pow(action.x,2)+pow(action.y,2)+pow(action.theta,2) );

      if (thisQ > maxQ) {
        maxQ = thisQ;
        x = action.x;
        y = action.y;
        theta = action.theta;
        this_reward = nearN[i].reward;
        maxQnode = i;
      }
    }

    // Convert to spherical coordinates to make adjustments.
    spt = cart2sph(x,y,theta);

    // Lengthen the action vector if needed.
    // if ((this_reward == warm) || (spt.r == 0)) {
    if (this_reward == warm) {
      spt.r = spt.r + step_size;
    }

    if (this_reward == cold) {
      if (spt.r == 0) {
        spt.p = spt.p + rand_in_range(-M_PI,M_PI);
        spt.t = spt.t + rand_in_range(-M_PI,M_PI);
```

```
423         spt.r = spt.r + step_size;
424       }
425       else
426       {
427         // To get off of the origin at t=1, give priority to angles below.
428         spt.p = spt.p + rand_in_range(-angle_delta/spt.r, angle_delta/spt.r);
429         spt.t = spt.t + rand_in_range(-angle_delta/spt.r, angle_delta/spt.r);
430       }
431     }
432
433     // Convert back to cartesian coordinates.
434     cpt = sph2cart(spt.r,spt.p,spt.t);
435
436     switch(this_reward)
437     {
438       case 0: // hot
439         NN.last = nearN[maxQnode].action;
440         NN.action = nearN[maxQnode].action;
441         NN.ancestor = nearN[maxQnode].nodeLabel;
442         // NN.reward = hot;
443         break;
444
445       case 1: // warm
446         // Set last to be the current action.
447         NN.last = nearN[maxQnode].action;
448         NN.action.x = cpt.x;
449         NN.action.y = cpt.y;
450         NN.action.theta = cpt.z;
451         NN.ancestor = nearN[maxQnode].nodeLabel;
452         break;
453
454       case -1: // cold
455         // No change to .last
456         NN.action.x = cpt.x;
457         NN.action.y = cpt.y;
458         NN.action.theta = cpt.z;
459         NN.ancestor = nearN[maxQnode].nodeLabel;
460         break;
461
462       default:
463         printf("Bad reward designation:  reward = %d\n",this_reward);
464         break;
465     } // switch
466
467   }
468
469   // This sets up NN.Q for the next iteration.
470   NN.Q = calc_Q(NN);
471
472 } // genNeighborhoodAction_xyt
473
474
475 // ----------------------------------------------------------------
476 // Function name:  getResponseFeedback_warmerColder
477 // Description:  This function gets the user feedback on the robot's
478 // response.  If autogen is enabled, the response will be autmatically
479 // generated.  Otherwise, the function will prompt the user for an
480 // integer response in {-1, 0, 1, 2}
481 // ----------------------------------------------------------------
482 void getResponseFeedback_warmerColder(std::vector<refNode> &A, refNode &NN,
483   int &autoGen, int &gestureType, double err_tol, int epoch)
484 {
485
486   int fb = -1;
487
488   // Indices into known actions data structure.
489   // static int origin = 0;
490   // static int come = 1;
491   // static int go = 2;
492   // static int stop = 3;
493
494   // Get response associated with the NN (it's current action).
495   double x = NN.action.x;
496   double y = NN.action.y;
497   double t = NN.action.theta;
498   // double v = NN.action.vel;
499   // int rwd = NN.reward;
500
501   double last_x = NN.last.x;
502   double last_y = NN.last.y;
503   double last_t = NN.last.theta;
504
505   double goal_x, goal_y, goal_t;
506   double dis2goal_x, dis2goal_y, dis2goal_t;
507   double last2goal_x, last2goal_y, last2goal_t;
508
509
```

```
510   // Initialize known goals if autoGen is selected.
511   knownGoals2 autoGoals;
512   autoGoals.init();
513
514   if (autoGen == 1)
515   {
516     goal_x = autoGoals.goalVec[gestureType].x;
517     goal_y = autoGoals.goalVec[gestureType].y;
518     goal_t = autoGoals.goalVec[gestureType].theta;
519
520     // goal_v = autoGoals.goalVec[gestureType].vel;
521
522     // Compute the distance to goal from the most recent move.
523     dis2goal_x = goal_x - x;
524     dis2goal_y = goal_y - y;
525     dis2goal_t = goal_t - t;
526
527     last2goal_x = goal_x - last_x;
528     last2goal_y = goal_y - last_y;
529     last2goal_t = goal_t - last_t;
530
531     // Compute magnitudes of distance to goal and last distance to goal.
532     // If the distance to goal has decreased, then we're getting warmer.
533     double mag_dis2goal =  sqrt( pow(dis2goal_x,2)  +
534       pow(dis2goal_y,2)  + pow(dis2goal_t,2) );
535     double mag_last2goal = sqrt( pow(last2goal_x,2) +
536       pow(last2goal_y,2) + pow(last2goal_t,2) );
537
538
539     if (mag_dis2goal < err_tol)             {fb =  0;} // on target
540     else if (mag_dis2goal < mag_last2goal)  {fb =  1;} // warmer
541     else if (mag_dis2goal >= mag_last2goal) {fb = -1;} // colder
542     else {
543       fb = -99;
544       printf("---------- FEEDBACK ERROR ---------- \n");
545       printf("goal = [%6.2f, %6.2f, %6.2f]\n", goal_x, goal_y, goal_t);
546       printf("act  = [%6.2f, %6.2f, %6.2f]\n", x, y, t);
547       printf("last = [%6.2f, %6.2f, %6.2f]\n", last_x, last_y, last_t);
548       printf("------------------------------------ \n");
549       } // error
550
551     // Compute average error per node (E), and report along with numNodes.
552     double avg_E;
553     avg_E = calc_avgE(A);
554
555     // Results reporting
556     printf("%d %d %.3f\n", gestureType, epoch, mag_dis2goal);
557
558
559 /*
560  printf("Obs#= %5d, NN= %3d, GstType= %d, r= %2d, ERR= %6.2f (Avg = %6.2f), Q = %6.2f, l2gXYT = %6.2f, fb=%2d, ",
561       NN.numObservations, NN.nodeLabel, gestureType, rwd, mag_dis2goal, avg_E, NN.Q, mag_last2goal, fb
562       );
563 */
564
565
566
567   }
568   else // augoGen=0 -> collect user reward
569   {
570     printf("Enter the X response quality: \n");
571     printf("-1 = colder \n");
572     printf(" 0 = no change \n");
573     printf(" 1 = warmer \n");
574     printf(" 2 = on target \n");
575     printf("> ");
576     fb = getchar(); // << Fix this; may not read two chars in "-1".
577
578     printf("Response:  reward = %d \n", fb);
579   }
580
581   NN.reward = fb;
582
583   // Put the updated refNode back into the A matrix.
584   int numNodes = A.size();
585   for (int i=0; i<numNodes; i++)
586   {
587     if (A[i].nodeLabel == NN.nodeLabel)
588     {
589       A[i] = NN;
590     }
591   }
592
593
594 } // getResponseFeedback_warmerCooler
595
596 // ------------------------------------------------------------------
```

```
597 // Function name:  genRep_HOGs
598 // Description:  This function reads a ROS bag file of skeleton
599 // messages and generates a gesture representation using Histograms
600 // of Gradients (HOGs) in 3D.  Bins are octants where each octant
601 // has an address consisting of 3 bits (000 through 111).  Each
602 // Bit represents either an increas (0) or decrease (1) in x, y, and z
603 // respectively.
604 // ------------------------------------------------------------------
605 std::vector<double> genRep_HOGs(const char* baseFileName)
606 {
607
608   static char bagFileName[40]; // ROS bag file previously collected
609   sprintf(bagFileName, "%s.bag", baseFileName);
610
611   // Initialize HOG to eight empty octants.
612   std::vector<double> HOG(8,0);
613
614   // Initilize position vector (x,y,z) for the Left Hand motion.
615   std::vector<pt3> posVec_LH(0);
616   pt3 posPt;
617
618   // Read in bag file data.
619   rosbag::Bag read_bag;
620   read_bag.open(bagFileName, rosbag::bagmode::Read);
621
622   rosbag::View view(read_bag, rosbag::TopicQuery("Skeletons"));
623   BOOST_FOREACH(rosbag::MessageInstance const m, view) {
624     body_msgs::Skeletons::ConstPtr s = m.instantiate<body_msgs::Skeletons>();
625     if (s != NULL) {
626       posPt.x = s->skeletons[0].left_hand.position.x;
627       posPt.y = s->skeletons[0].left_hand.position.y;
628       posPt.z = s->skeletons[0].left_hand.position.z;
629       posVec_LH.push_back(posPt);
630     }
631   } // BOOST_FOREACH
632   read_bag.close();
633
634   // Smooth position vectors (Gaussian).
635   smooth3Dpoints(posVec_LH);
636
637   int message_margin = 5;
638   int message_size = posVec_LH.size();
639   for (int i=message_margin; i<(message_size - message_margin); i++)
640   {
641     int index = 0;
642
643     // Find the HOG entry to increment.
644     if (posVec_LH[i+3].x <= posVec_LH[i-3].x) {index += 1.0;}
645     if (posVec_LH[i+3].y <= posVec_LH[i-3].y) {index += 2.0;}
646     if (posVec_LH[i+3].z <= posVec_LH[i-3].z) {index += 4.0;}
647
648     HOG[index] += 1;
649   }
650
651   return(HOG);
652
653 } // genRep_HOGs
654
655
656 // ------------------------------------------------------------------
657 // Function name:  genRep_dynamicInstants
658 // Description:  This function reads a ROS bag file of skeleton
659 // messages and generates a gesture representation using Rao's
660 // concept of dynamic instants.
661 // ------------------------------------------------------------------
662 std::vector<double> genRep_dynamicInstants(const char* baseFileName)
663 {
664
665   static char bagFileName[40]; // ROS bag file previously collected
666   static char DI_fname[40]; // dynamic instants file
667   static char PVA_fname[40]; // pos/vel/acc data file
668   sprintf(bagFileName, "%s.bag", baseFileName);
669   sprintf(DI_fname, "%s_DI.txt", baseFileName);
670   sprintf(PVA_fname, "%s_PVA.txt", baseFileName);
671
672   printf("Generating Dynamic Instants (DI) representation from %s. \n",
673     bagFileName);
674
675   int msgCnt = 0;
676   int numDIs = 5;
677   pt3 posPt, velPt, accPt;
678
679   // Left Hand (LH) osition/velocity/acceleration readings
680   std::vector<pt3> posVec_LH(0);
681   std::vector<pt3> velVec_LH(0);
682   std::vector<pt3> accVec_LH(0);
683   std::vector<pt3> tempVec(0);
```

```
684
685     // Dynamic Instants list
686    std::vector<dynamic_instant> DI_list(0);
687
688    // Vector form of DI list formatted for GNG algorithm.
689    std::vector<double> vec_in(0);
690
691    // Read in bag file data.
692    rosbag::Bag read_bag;
693    read_bag.open(bagFileName, rosbag::bagmode::Read);
694
695    rosbag::View view(read_bag, rosbag::TopicQuery("Skeletons"));
696    BOOST_FOREACH(rosbag::MessageInstance const m, view) {
697      body_msgs::Skeletons::ConstPtr s = m.instantiate<body_msgs::Skeletons>();
698      if (s != NULL) {
699        posPt.x = s->skeletons[0].left_hand.position.x;
700        posPt.y = s->skeletons[0].left_hand.position.y;
701        posPt.z = s->skeletons[0].left_hand.position.z;
702        posVec_LH.push_back(posPt);
703      }
704    } // BOOST_FOREACH
705    read_bag.close();
706    msgCnt = posVec_LH.size();
707
708    // Cut off ends of POS data.
709    posVec_LH.erase(posVec_LH.begin(), posVec_LH.begin()+10);
710    posVec_LH.erase(posVec_LH.end()-10, posVec_LH.end());
711
712    // Smooth position vectors (Gaussian).
713    smooth3Dpoints(posVec_LH);
714
715    // Scale the position vectors on [0,1]
716    scale01(posVec_LH);
717
718    // Calculate velocity vectors.
719    velVec_LH = deriv3D(posVec_LH, 7);
720
721    // Calculate acceleration vectors.
722    accVec_LH = deriv3D(velVec_LH, 7);
723
724    // Write out pos/vel/acc data (optional).
725    // write_PVA_data(PVA_fname, posVec_LH, velVec_LH, accVec_LH);
726
727    // Generate Dynamic Instants (DI) list.
728    vec_in = find_DIs(DI_fname, posVec_LH, velVec_LH, accVec_LH, numDIs);
729
730    return(vec_in);
731
732  } // genRep_dynamicInstants
733
734  // ------------------------------------------------------------------
735  // Function name:  genRep_dynamicInstants
736  // Description:  This function reads a ROS bag file of skeleton
737  // messages and generates a gesture representation using Rao's
738  // concept of dynamic instants.
739  // ------------------------------------------------------------------
740  std::vector<pt3> read_POS_data(const char* baseFileName)
741  {
742
743    static char bagFileName[40]; // ROS bag file previously collected
744    sprintf(bagFileName, "%s.bag", baseFileName);
745
746    // int msgCnt = 0;
747    pt3 posPt;
748
749    // Left Hand (LH) osition/velocity/acceleration readings
750    std::vector<pt3> posVec_LH(0);
751    // std::vector<pt3> velVec_LH(0);
752    // std::vector<pt3> accVec_LH(0);
753
754    // Read in bag file data.
755    rosbag::Bag read_bag;
756    read_bag.open(bagFileName, rosbag::bagmode::Read);
757
758    rosbag::View view(read_bag, rosbag::TopicQuery("Skeletons"));
759    BOOST_FOREACH(rosbag::MessageInstance const m, view) {
760      body_msgs::Skeletons::ConstPtr s = m.instantiate<body_msgs::Skeletons>();
761      if (s != NULL) {
762        posPt.x = s->skeletons[0].left_hand.position.x;
763        posPt.y = s->skeletons[0].left_hand.position.y;
764        posPt.z = s->skeletons[0].left_hand.position.z;
765        posVec_LH.push_back(posPt);
766      }
767    } // BOOST_FOREACH
768    read_bag.close();
769    // printf("Pos vector has %d elements\n", posVec_LH.size() );
770
```

```
771   // Cut off ends of POS data.
772   posVec_LH.erase(posVec_LH.begin(), posVec_LH.begin()+10);
773   posVec_LH.erase(posVec_LH.end()-10, posVec_LH.end());
774
775   // Smooth position vectors (Gaussian).
776   smooth3Dpoints(posVec_LH);
777
778   // Scale the position vectors on [0,1]
779   scale01(posVec_LH);
780
781
782   return(posVec_LH);
783
784
785 } // read_POS_data
786
787 // ----------------------------------------------------------------
788 // Function name:  moveTurtle
789 // Description:  This function uses the goalConfig fields of a refNode
790 // to generate a robotic response (x, y, theta, velocity).  This function
791 // assumes that the robot (TurtleSim) is always beginning its trajectory from
792 // (x,y,theta) = (5.5, 5.5, 0).
793 //
794 // Motion of the turtle is composed of:
795 // 1. an initial rotation toward (x,y) ("angle1"),
796 // 2. foward motion to (x,y) ("distance"),
797 // 3. a final rotation to the desired angle of approach ("angle2").
798 //
799 // 01/03/2012:  Current thinking is that a non-zero velocity represents
800 // constant speed.  A zero velocity represents a stopped robot.  This
801 // is to allow for the stop gesture.
802 // ----------------------------------------------------------------
803 void moveTurtle(refNode &NN)
804 {
805
806   // Create the action client. "true" causes the client to spin it's own thread.
807   actionlib::SimpleActionClient<turtleControl::moveTurtleAction> ac("turtle_motion", true);
808
809   // ros::Rate poll_rate(20);
810
811   ROS_INFO("Waiting for action server to start.");
812   // wait for the action server to start
813   ac.waitForServer(); //will wait for infinite time
814
815   ROS_INFO("Action server started, sending goal.");
816   // send a goal to the action
817   turtleControl::moveTurtleGoal goal;
818
819   goal.x = NN.action.x;
820   goal.y = NN.action.y;
821   goal.theta = NN.action.theta; // or 14*M_PI/8;
822
823   printf("Goal:  x = %8.5f, y = %8.5f, theta = %8.5f. \n", goal.x, goal.y, goal.theta);
824
825   ac.sendGoal(goal);
826
827   //wait for the action to return
828   bool finished_before_timeout = ac.waitForResult(ros::Duration(40.0));
829
830   if (finished_before_timeout)
831   {
832     actionlib::SimpleClientGoalState state = ac.getState();
833     ROS_INFO("Action finished: %s",state.toString().c_str());
834   }
835   else
836   {
837     ROS_INFO("Action did not finish before the time out.");
838   }
839
840 } // moveTurtle
841
842 // ----------------------------------------------------------------
843 // Function name:  getFileType
844 // Description:  This function reads the last few characters of
845 // the baseFileName and decomposes it to see what type of gesture it
846 // contains.
847 // ----------------------------------------------------------------
848 int getBagType (char* baseFileName)
849 {
850   int gestureType = 99;
851   int come = 1;
852   int go = 2;
853   int stop = 3;
854   int eat = 4;
855   int read = 5;
856   int sleep = 6;
857   int get = 7;
```

```
858   int give = 8;
859   int therapy = 9;
860   int nameLen = strlen(baseFileName);
861
862   // Cast baseFileName to string type.
863   std::string fileString = std::string(baseFileName);
864
865   std::string last4 = fileString.substr(nameLen-4,nameLen-1);
866   std::string last2 = fileString.substr(nameLen-2,nameLen-1);
867
868   if (last4 == "come")
869   {
870     gestureType = come;
871   }
872   else if (last4 == "stop")
873   {
874     gestureType = stop;
875   }
876   else if (last4 == "read")
877   {
878     gestureType = read;
879   }
880   else if (last4 == "leep")
881   {
882     gestureType = sleep;
883   }
884   else if (last4 == "give")
885   {
886     gestureType = give;
887   }
888   else if (last4 == "rapy")
889   {
890     gestureType = therapy;
891   }
892   else if (last2 == "go")
893   {
894     gestureType = go;
895   }
896   else if (last2 == "at")
897   {
898     gestureType = eat;
899   }
900   else if (last2 == "et")
901   {
902     gestureType = get;
903   }
904   else
905   {
906     std::cout << "Unrecognized gesture type: " << last4 << "\n";
907     // printf("Unrecognized gesture type: %c\n",last4);
908     // Do nothing, gestureType = 99;
909   }
910
911   // std::cout << last4 << "\n";
912
913   return(gestureType);
914 }
```

A.1.2 gestureLrnList.cpp

```
1  #include "kinect_includes.h"
2  #include "points.h"
3  #include "utilities.h"
4  #include "gng.h"
5  // #include "matrixOps.h"
6  #include "graphs.h"
7  #include "assertions.h"
8  #include "gestureLrn.h"
9  #include "lists.h"
10
11 // #include "cmatrix"
12
13 using namespace sensor_msgs;
14 using namespace ros;
15 using namespace std;
16
17 char A_fname[40] = "A.txt";
18 char C_fname[40] = "C.txt";
19 char* baseFileName;
20 int autoGen = 1;
21 std::vector<refNode> A(0);
22 std::vector<refNode> N(0);
23 std::vector<connection> C(0);
24 knownGoals autoGoals;
```

```
25 refNode NN;
26 int observationNum = 0;
27 int gestureClass = 99;
28 double err_tol = 0.2;
29 V_string args;
30
31 // Data representation parameters
32 std::vector<descriptor> descriptor_list(0);
33 // char descriptor_fname[40] = "DIs.txt";
34 int featureVecSize = 20;
35 // char descriptor_fname[40] = "HOGs.txt";
36 // int featureVecSize = 8;
37 std::vector<double> vec_in;
38
39 // Graph parametersm
40 // std::vector<std::vector<double> > DistMat(0);
41 // std::vector<std::vector<double> > ResDistMat(0);
42 // std::vector<std::vector<double> > AdjMat(0);
43 // std::vector<std::vector<double> > AdmMat(0); // Admittance (Kirchhoff)
44 // std::vector<std::vector<double> > LapMat(0); // Laplacian
45 // std::vector<double> k_vector(0);
46
47
48 // GNG parameters
49 int lambda = 100;
50 int maxNodeCnt = 100; // Change to 100 for randomized vectors
51
52 // Parameters for list of gestures:
53 int numEpochs;
54 int hoodRadius;
55
56
57 int main(int argc, char **argv)
58 {
59
60   srand(time(0));
61
62   /*
63   printf("+-----------------------------------------+\n");
64   printf("| Running:  gestureLrnList                |\n");
65   printf("+-----------------------------------------+\n");
66   */
67
68   // Parse the command line for number of epochs to run.
69   if (argc != 4)
70   {
71     printf("Usage:  gestureLrnList <DI_fileName> <numEpochs> <hoodRadius>\n");
72     return(0);
73   }
74   else
75   {
76
77     // Read in DIs from file.
78
79     char* descriptor_fname = argv[1];
80     descriptor_list = read_descriptor_list(descriptor_fname, featureVecSize);
81     int numSamples = descriptor_list.size();
82     // printf("Read %d DIs from file %s.\n", numSamples, descriptor_fname);
83
84     numEpochs = atoi(argv[2]);
85     hoodRadius = atoi(argv[3]);
86     // printf("Running %d epochs.\n", numEpochs);
87
88
89
90     for(int p=1; p<=numEpochs; p++)
91     {
92
93       for(int v=0; v<numSamples; v++)
94       {
95         vec_in = descriptor_list[v].featureVec;
96         gestureClass = descriptor_list[v].classNum;
97
98         // Apply the representation to the GNG algorithm.
99         NN = gng(A_fname, C_fname, A, C, vec_in, N, lambda, maxNodeCnt);
100        ASSN_duplicateConx(C);
101
102        // Generate a response based on neighbors (for Warmer/Cooler feedback scheme).
103        // Put the response into NN.goal.
104        genNeighborhoodAction_xyt(NN, N, A, C, err_tol, hoodRadius);
105
106        // Generate a response by turtleSim.
107        // moveTurtle(NN);
108
109        // Collect feedback on the generated response.
110        // Only use gestureClass if the response is autogenerated.
111        getResponseFeedback_warmerColder(A, NN, autoGen, gestureClass, err_tol, p);
```

```
112
113         // int nodes = A.size();
114         // printf("nodes = %3d \n",nodes);
115
116       }
117
118       // Write once per epoch.
119       // write_A(A_fname, A);
120       // write_C(C_fname, C);
121
122     } // for p = numEpochs
123
124     write_A(A_fname, A);
125     write_C(C_fname, C);
126
127   } // num args
128
129
130   return(1);
131
132 } // end main
```

A.1.3 gng.h

```
1 // This file contains classes/functions and prototypes for use with the
2 // Growing Neural Gas (GNG) algorithm.
3
4 // ------------------------------------------------------------------
5 // Class name:  goalConfig
6 // Description:  This class describes the robotic goal configuration in 2D
7 // associated with a gng reference node.  Currently (12/27/2011),
8 // this is just the goal configuration (x,y,theta).  Trajectory needs
9 // to be added in the future for sociability.
10 // ------------------------------------------------------------------
11 class config2D
12 {
13 public:
14   double x, y; // in meters
15   double theta; // in radians
16   double vel; // in cm/s
17
18   // Constructor
19   config2D(double x_in = 0.0, double y_in = 0.0,
20     double theta_in = 0.0, double vel_in = 0.0)
21   {
22     x = x_in;
23     y = y_in;
24     theta = theta_in;
25     vel = vel_in;
26   }
27
28 }; // goalConfig
29
30 // ------------------------------------------------------------------
31 // Class name:  knownGoals2
32 // Description:  This class stores the goals of known gestures when
33 // the gestureLrn algorithm is run in automatic mode.  These are the
34 // goals for Come, Go, Stop, Eat, Read, Sleep, Get, Give and Therapy
35 // gestures.  This class is similar to "knownGoals" except that
36 // config2D members are stored in a single vector.
37 // ------------------------------------------------------------------
38
39 class knownGoals2
40 {
41   public:
42     std::vector<config2D> goalVec;
43     config2D come;
44     config2D go;
45     config2D stop;
46     config2D eat;
47     config2D read;
48     config2D sleep;
49     config2D get;
50     config2D give;
51     config2D therapy;
52     config2D origin;
53     // config2D gest4;
54
55
56   // Constructor
57   knownGoals2() {}
58
59   void init()
60   {
```

```
    // These values are RELATIVE to the turtleSim origin (5.55, 5.55, 0)
    come.x   = 3.95; come.y   = 3.95; come.theta   =1*M_PI/4; come.vel   =1.0; // 1
    go.x     = 3.95; go.y     =-3.95; go.theta     =7*M_PI/4; go.vel     =1.0; // 2
    stop.x   =-3.95; stop.y   =-3.95; stop.theta   =5*M_PI/4; stop.vel   =0.0; // 3

    eat.x    =-3.95; eat.y    = 3.95; eat.theta    =3*M_PI/4; eat.vel    =1.0; // 4
    read.x   = 3.95; read.y   = 0.0 ; read.theta   =0*M_PI/4; read.vel   =1.0; // 5
    sleep.x  = 0.0 ; sleep.y  = 3.95; sleep.theta  =2*M_PI/4; sleep.vel  =1.0; // 6

    get.x    =-3.95; get.y    = 0.0 ; get.theta    =4*M_PI/4; get.vel    =1.0; // 7
    give.x   = 0.0 ; give.y   =-3.95; give.theta   =6*M_PI/4; give.vel   =1.0; // 8
    therapy.x= 3.95; therapy.y= 1.98; therapy.theta=1*M_PI/8; therapy.vel=1.0; // 9


    // Describe the turtle's origin for comparison to generated responses.
    // origin.x = 5.55; origin.y = 5.55; origin.theta = 0.0; origin.vel = 0.0;
    origin.x = 0.0; origin.y = 0.0; origin.theta = 0.0; origin.vel = 0.0;

    // Store the 2D configurations in a vector where the entry subscript
    // corresponds to the gestureType encoding.
    goalVec.push_back(origin);
    goalVec.push_back(come);
    goalVec.push_back(go);
    goalVec.push_back(stop);
    goalVec.push_back(eat);
    goalVec.push_back(read);
    goalVec.push_back(sleep);
    goalVec.push_back(get);
    goalVec.push_back(give);
    goalVec.push_back(therapy);
    // Add more to the known goals as we consider them.
  }


}; // autoGoals


// ------------------------------------------------------------------
// Class name:  knownGoals
// Description:  This class stores the goals of known gestures when
// the gestureLrn algorithm is run in automatic mode.  These are the
// goals for Come, Go, and Stop gestures.
//
// OBSOLETE - 5/24/2013.
// ------------------------------------------------------------------
class knownGoals
{
  public:
    config2D come;
    config2D go;
    config2D stop;
    config2D origin;
    config2D gest4;

  // Constructor
  knownGoals() {}

  void init()
  {

    // These values are RELATIVE to the turtleSim origin (5.55, 5.55, 0)
    come.x =   3.95; come.y =   3.95; come.theta =    M_PI/4; come.vel  = 1.0;
    go.x =     3.95;   go.y =  -3.95;   go.theta =  7*M_PI/8;   go.vel  = 1.0;
    stop.x =  -3.95; stop.y =   3.95; stop.theta =  5*M_PI/4; stop.vel  = 0.0;
    gest4.x = -3.95; gest4.y = -3.95; gest4.theta = 3*M_PI/4; gest4.vel = 1.0;

    // Describe the turtle's origin for comparison to generated responses.
    // origin.x = 5.55; origin.y = 5.55; origin.theta = 0.0; origin.vel = 0.0;
    origin.x = 0.0; origin.y = 0.0; origin.theta = 0.0; origin.vel = 0.0;
  }


}; // autoGoals

// ------------------------------------------------------------------
// Class name:  refNode
// Description:  This class describes a node in a GNG cloud.  A node
// consists of a featureVec, a label number, the nodes number of
// connections, and a local error variable.
// ------------------------------------------------------------------
class refNode
{
public:
  std::vector<double> featureVec;
  int nodeLabel;
  int numConnections;
```

```
148   double E;
149   config2D action; // current action x,y,t,v configuration
150   config2D last; // last action x,y,t,v configuration
151   int numObservations;
152   int reward; // most recent user feedback (reward)
153   double Q;
154   int ancestor; // the nodeLabel from which the current response was taken
155
156
157   // Constructor
158   refNode(): featureVec(0) {};
159 }; // refNode;
160
161
162 // ------------------------------------------------------------------
163 // Class name:  connection
164 // Description:  This class describes a connection between refNodes
165 // in a GNG cloud.
166 // ------------------------------------------------------------------
167 class connection
168 {
169 public:
170   // Connection endpoints (refNode vertices) and age.
171   int v1, v2, age;
172
173   // Length of the connection (a cost value).
174   double length;
175
176   // Constructor
177   connection(int v1_in = -1, int v2_in = -1, int age_in = -1,
178     double length_in = -1.0)
179   {
180     v1 = v1_in;
181     v2 = v2_in;
182     age = age_in;
183     length = length_in;
184   }
185 }; // connection
186
187 // ------------------------------------------------------------------
188 // Class name:  distPt
189 // Description:  A distPt is the distance from a current input vector
190 // to a reference node along with the reference node's label.
191 // ------------------------------------------------------------------
192 class distPt
193 {
194 public:
195   double distance;
196   int nodeLabel;
197
198   // Constructor
199   distPt(double distance_in = -1.0, int nodeLabel_in = -1) {
200     distance = distance_in;
201     nodeLabel = nodeLabel_in;
202   }
203 }; // distPt
204
205 // ------------------------------------------------------------------
206 // Prototypes
207 // ------------------------------------------------------------------
208
209 refNode genNode(std::vector<refNode> &A, const int numFeatures,
210   int &numObservations);
211
212 double calc_Q(refNode NN);
213
214 double calc_avgE(std::vector<refNode> &A);
215
216 double vecNorm2(std::vector<double> &v1, std::vector<double> &v2);
217
218 bool compareDistances(distPt d1, distPt d2);
219
220 void get2ClosestNodes (std::vector<refNode> &A, std::vector<double> &vec_in,
221   std::vector<distPt> &Dv);
222
223 void read_A(const char* fname, std::vector<refNode> &A,
224   const int &numFeatures);
225
226 void read_C(const char* fname, std::vector<connection> &C);
227
228 void write_A(const char* fname, std::vector<refNode> &A);
229 void write_C(const char* fname, std::vector<connection> &C);
230
231 void check4Connection(std::vector<refNode> &A, std::vector<connection> &C,
232   std::vector<distPt> &Dv);
233
234 void adjustWinner(std::vector<refNode> &A, std::vector<distPt> &Dv,
```

```
std::vector<double> v_in, const double &ep_w);

void adjustNeighbors(std::vector<refNode> &A, std::vector<connection> &C,
std::vector<distPt> &Dv, std::vector<double> &v_in, const double &ep_n);

void removeOldConnections(std::vector<refNode> &A,
  std::vector<connection> &C, const int ageMax);

void printC(std::vector<connection> &C);
void print_refNodes(std::vector<refNode> &A, const char* nodeListName);

void interpolateNodes(refNode &q, refNode &f, refNode &r,
  const int &feedback_dim);

void insertNewNode(std::vector<refNode> &A, std::vector<connection> &C,
  const int &lambda, const double &alpha);

void decreaseNodeError(std::vector<refNode> &A, const double &beta);

refNode getNN(std::vector<distPt> &Dv, std::vector<refNode> &A);

void getNeighbors(refNode &NN, std::vector<refNode> &N, std::vector<refNode>
  &A, std::vector<connection> &C);

refNode gng(const char* A_fname, const char* C_fname,
  std::vector<refNode> &A, std::vector<connection> &C,
  std::vector<double> &vec_in, std::vector<refNode> &N,
  const int &lambda, const int maxNodeCnt);
```

A.1.4 gng.cpp

```
#include "kinect_includes.h"
#include "points.h"
#include "utilities.h"
#include "gng.h"

// ------------------------------------------------------------------
// Function name:  genNode
// Description:  This function generates a random nx1 reference
// vector to seed the GNG function.  It also generates a randomized
// set of goalConfig parameters.
// ------------------------------------------------------------------
refNode genNode(std::vector<refNode> &A, const int numFeatures, int &numObservations)
{
  refNode newNode;
  std::vector<double> featureVec(0);
  int reward;
  int numNodes = A.size();
  int maxNodeLabel = -1;



  // Generate the feature vector.
  for (int i=0; i<numFeatures; i++) {
    double r = rand_in_range(0.0,0.5);
    // printf("r = %8.5f\n",r);
    featureVec.push_back(r);
  }

  // Initialize reward - pessimistic (will cause a random initial guess at configuration).
  reward = -1;


  // Generate the goalConfig.
  // Use the x,y,theta constraints of the turtlesim arena
  // at this writing (01/03/2012).  Constrain new nodes to stay
  // close to the center (x,y) = (5.5, 5.5).  Note that (x,y,t)
  // values below are _RELATIVE_ and not absolute values (this is
  // how the turtueControl_server works).
  double x = 0; // rand_in_range(-0.5, 0.5); // x can be on [0.5, 10.5]
  double y = 0; // rand_in_range(-0.5, 0.5); // y can be on [0.5, 10.5]
  double t = 0; // rand_in_range(0, 2*M_PI-0.00001); // theta can be on [0, 2*PI]

  // Consider assigning velocity to either 0 or 1. <<<<<<<<<<<
  double v = 0; //  rand01();

  // Generate the nodeLabel.
  if (numNodes == 0)
  {
    maxNodeLabel = 0;
  }
  else
  {
    for(int i=0; i<numNodes; i++) {
```

```
54        if (A[i].nodeLabel > maxNodeLabel) {
55          maxNodeLabel = A[i].nodeLabel;
56        }
57      }
58    }
59
60    // Generate the new node.
61    newNode.featureVec = featureVec;
62    newNode.nodeLabel = maxNodeLabel + 1;
63    newNode.numConnections = 0;
64    newNode.E = 0;
65    newNode.action.x = x;
66    newNode.action.y = y;
67    newNode.action.theta = t;
68    newNode.action.vel = v;
69
70    // Let the intitial 'last' config be the origin.
71    newNode.last.x = 0;
72    newNode.last.y = 0;
73    newNode.last.theta = 0;
74    newNode.last.vel = 0;
75
76    newNode.numObservations = numObservations;
77    newNode.reward = reward;
78    newNode.Q = 0;
79    newNode.ancestor = newNode.nodeLabel; // newNode is its own ancestor
80
81    return(newNode);
82  }
83
84  // ------------------------------------------------------------------
85  // Function name:  calc_Q
86  // Description:  This function calculates the Q-Learning Q value
87  // for a configuration vector (6/30/2012).  May change in the future.
88  // ------------------------------------------------------------------
89  double calc_Q(refNode NN)
90  {
91    double Q;
92
93    Q = sqrt( pow(NN.action.x,2) + pow(NN.action.y,2) + pow(NN.action.theta,2) );
94
95    return(Q);
96  }
97
98  // ------------------------------------------------------------------
99  // Function name:  calc_avgE
100 // Description:  This function calculates the average error for
101 // the GNG cloud [A] matrix.
102 // ------------------------------------------------------------------
103 double calc_avgE(std::vector<refNode> &A)
104 {
105     int numNodes = A.size();
106     double total_E = 0;
107     double avg_E = 0;
108     for (int i=0; i<numNodes; i++)
109     {
110       total_E += A[i].E;
111     }
112     avg_E = (double) total_E/numNodes;
113
114     return(avg_E);
115 }
116
117
118 // ------------------------------------------------------------------
119 // Function name: vecNorm2
120 // Description:  This function finds the Euclidean distance between
121 // two nx1 vectors.  Vectors are assumed to have the same number of
122 // elements.
123 // ------------------------------------------------------------------
124 double vecNorm2(std::vector<double> &v1, std::vector<double> &v2)
125 {
126   int numElements = v1.size();
127   double sum = 0.0;
128   // std::vector<double> diffVec = v1;
129
130   for(int i=0; i<numElements; i++) {
131     sum = sum + pow(v1[i]-v2[i], 2);
132   }
133   return(sqrt(sum));
134 }
135
136 // ------------------------------------------------------------------
137 // Function name:  get2ClosestNodes
138 // Description:  This function gets the two closest nodes from the
139 // list of existing reference nodes in a GNG cloud.
140 // ------------------------------------------------------------------
```

```
141 bool compareDistances(distPt d1, distPt d2)
142 {
143   return(d1.distance < d2.distance);
144 }
145
146 void get2ClosestNodes (std::vector<refNode> &A, std::vector<double> &vec_in,
147 std::vector<distPt> &Dv)
148 {
149   Dv.clear();
150   distPt q, s1, s2;
151   double dist;
152   int numNodes = A.size();
153
154   // Calculate the distances from vec_in to all nodes in [A].
155   for(int i=0; i<numNodes; i++) {
156     dist = vecNorm2(vec_in, A[i].featureVec);
157     q.distance = dist;
158     q.nodeLabel = A[i].nodeLabel;
159     Dv.push_back(q);
160   }
161
162   // DEBUG
163   /*
164   int u = Dv.size();
165   printf("Dv unsorted: \n");
166   for (int i=0; i<u; i++) {
167     printf("Dv[%d].dist = %5.3f, nodeLabel = %d\n", i, Dv[i].distance, Dv[i].nodeLabel);
168   }
169   */
170
171   /* DEBUG
172   int v = A[0].featureVec.size();
173   printf("fvec= ");
174   for(int i=0; i<v; i++) {
175     printf("%5.3f ",vec_in[i]);
176   }
177   printf("\n");
178
179   for(int h=0; h<numNodes; h++){
180     printf("A[%d]= ",h);
181     for(int i=0; i<v; i++) {
182       printf("%5.3f ",A[h].featureVec[i]);
183     }
184     printf(": D=%5.3f\n", Dv[h].distance);
185   }
186   */
187
188   // Sort the distances
189   sort(Dv.begin(), Dv.end(), compareDistances);
190   s1 = Dv[0];
191   s2 = Dv[1];
192   Dv.clear();
193   Dv.push_back(s1);
194   Dv.push_back(s2);
195
196   // DEBUG
197   /*
198   u = Dv.size();
199   printf("Dv sorted: \n");
200   for (int i=0; i<u; i++) {
201     printf("Dv[%d].dist = %5.3f, nodeLabel = %d\n", i, Dv[i].distance, Dv[i].nodeLabel);
202   }
203   */
204
205   /*
206   printf("Dis0 = %6.3f [node %d], Dis1 = %6.3f [node %d] \n", Dv[0].distance, Dv[0].nodeLabel,
207     Dv[1].distance, Dv[1].nodeLabel);
208   */
209
210 }
211
212 // ------------------------------------------------------------------
213 // Function name: read_A
214 // Description:  This function reads in the A matrix for the GNG
215 // algorithm.  If the file does not exist, the function creates an
216 // initialized A matrix.
217 //
218 // An input file of specific format is assumed.  At this writing (12/21/2011),
219 // each line of the file includes a reference node descriptor consisting
220 // of [nodeLabel, numConnections, E, featureVec(numFeatures), goal].  Each featureVec
221 // consists of five dynamic instants of [frameNum, x, y, z].
222 // ------------------------------------------------------------------
223 void read_A(const char* fname, std::vector<refNode> &A, const int &numFeatures)
224 {
225   refNode newNode;
226   std::vector<double> featureVec(0);
227   int nodeLabel, numConnections, numObservations, reward, ancestor;
```

```
228   double feature, E, x, y, theta, Q;
229   int numNodes = A.size();
230   int initial_numNodes = 2;
231
232   // Do not read this from file currently (01Sep2012)
233   double vel = 0;
234
235   // At startup or restart, [A] may be empty (numNodes may be zero).
236   // Otherwise, use an existing [A] matrix.
237   if (numNodes == 0) // Try to read in or generate [A].
238   {
239     // printf("\n");
240     FILE *pFile;
241     pFile = fopen(fname,"r");
242     A.clear();
243     numObservations = 0;
244
245     if (pFile == NULL) {
246       // Initialize [A]
247       // printf("File %s does not exist. Initializing [A] with 2 nodes. \n", fname);
248       for (int i=0; i<initial_numNodes; i++) {
249         newNode = genNode(A, numFeatures, numObservations);
250         A.push_back(newNode);
251         // printf("New node number = %d.  A has %d nodes. \n", newNode.nodeLabel, A.size());
252       }
253     }
254     else // Read in [A]
255     {
256       // printf("Reading [A] from existing %s file.\n", fname);
257
258       while(fscanf(pFile, "%d %d %d %d %d", &numObservations, &nodeLabel, &numConnections, &reward, &ancestor) !=
        EOF)
259       {
260         newNode.numObservations = numObservations;
261         newNode.nodeLabel = nodeLabel;
262         newNode.numConnections = numConnections;
263         newNode.reward = reward;
264         newNode.ancestor = ancestor;
265
266         /*
267         newNode.reward.clear();
268         for(int i=0; i<reward_dim; i++) {
269           if(fscanf(pFile, "%d", &fb) != EOF) {
270             newNode.reward.push_back(fb);
271             // Wprintf("Reading from A.txt --- reward = %2d\n", fb);
272           }
273         }
274         */
275
276         if(fscanf(pFile, "%lf", &Q) != EOF) {
277           newNode.Q = Q;
278         }
279
280         if (fscanf(pFile, "%lf", &E) != EOF) {
281           newNode.E = E;
282         }
283         if (fscanf(pFile, "%lf %lf %lf", &x, &y, &theta) != EOF) {
284           newNode.action.x = x;
285           newNode.action.y = y;
286           newNode.action.theta = theta;
287           newNode.action.vel = vel;
288         }
289
290         if (fscanf(pFile, "%lf %lf %lf", &x, &y, &theta) != EOF) {
291           newNode.last.x = x;
292           newNode.last.y = y;
293           newNode.last.theta = theta;
294           newNode.last.vel = vel;
295         }
296
297         newNode.featureVec.clear();
298         for(int i=0; i<numFeatures; i++) {
299           if(fscanf(pFile, "%lf", &feature) != EOF) {
300             newNode.featureVec.push_back(feature);
301             // printf("Reading from A.txt --- fvec[%2d] = %8.3f\n", i, feature);
302           }
303         }
304
305         A.push_back(newNode);
306
307       } // while
308     } // else read in [A]
309   }
310   else
311   {
312     // Use existing [A] matrix.
313   }
```

```
314
315
316   // print_refNodes(A, "A");
317
318 } // read_A
319
320 // ------------------------------------------------------------------
321 // Function name:  read_C
322 // Description:  This function reads in the C matrix for the GNG
323 // algorithm.  If the file does not exist, the function creates an
324 // initialized C matrix.
325 //
326 // An input file of specific format is assumed.  At this writing (12/21/2011),
327 // each line of the file includes a connection descriptor consisting of
328 // [vertex1, vertex2, age].
329 // ------------------------------------------------------------------
330 void read_C(const char* fname, std::vector<connection> &C)
331 {
332   int vertex1, vertex2, age;
333   double length;
334   connection cnxIn;
335   int numConx = C.size();
336
337   // At startup or restart, [C] may be empty (numConx may be zero).
338   // Otherwise, use an existing [C] matrix.
339   if (numConx == 0)
340   {
341     FILE *pFile;
342     pFile = fopen(fname,"r");
343
344     // printf("\n");
345     // Try to read in an existing [C] from a file.
346     if (pFile == NULL) {
347       // printf("File %s does not exist. Initializing [C] with 0 connections. \n", fname);
348       C.clear();
349     }
350     else
351     {
352       // printf("Reading [C] from existing %s file.\n", fname);
353       C.clear();
354       while(fscanf(pFile,"%d %d %d %lf", &vertex1, &vertex2, &age, &length) != EOF)
355       {
356         cnxIn.v1 = vertex1;
357         cnxIn.v2 = vertex2;
358         cnxIn.age = age;
359         cnxIn.length = length;
360         C.push_back(cnxIn);
361       }
362     }
363
364     // Print [C] to screen as a sanity check.
365     numConx = C.size();
366     // printf("[C] contains %d connections: \n", numConx);
367     for(int i=0; i<numConx; i++) {
368       /*
369       printf("Connection %3d: vertex 1 = %3d, vertex 2 = %3d, age = %3d \n",
370         i+1, C[i].v1, C[i].v2, C[i].age);
371       */
372     }
373   }
374   else
375   {
376     // Use existing [C] matrix.
377   }
378
379 } // read_C
380
381 // ------------------------------------------------------------------
382 // Function name:  write_A, write_C
383 // Description:  This function writes out the [A], [C] matrices as needed.
384 // (Not sure what 'as needed' means yet. 12/26/2011)
385 // ------------------------------------------------------------------
386 void write_A(const char* fname, std::vector<refNode> &A)
387 {
388   int numFeatures = A[0].featureVec.size();
389   int numNodes = A.size();
390
391   // Increment numObservations on each write, since writes only occur
392   // after a gesture observation vector has been received.
393   // int numObservations = A[0].numObservations + 1;
394
395   // Incrementing numObservations now occurs in the GNG algorithm.
396
397   FILE *pFile;
398   pFile = fopen(fname,"w");
399
400   // Print [A].numObservations, nodeLabel, .numConnections, .featureVec, .E,
```

```
401   // .action.x, .action.y, .action.theta, .action.vel
402   // .last.x, .last.y, .last.theta, .last.vel
403   // reward<>
404
405   /*
406   // This version worked on 01Sep2012.  It was revised to the versio below
407   // to place the feature vector last.
408
409   for (int i=0; i<numNodes; i++) {
410     fprintf(pFile, "%d %d %d %d ",A[i].numObservations, A[i].nodeLabel, A[i].numConnections, A[i].reward);
411
412     fprintf(pFile, "%lf ", A[i].Q);
413
414     for (int j=0; j<numFeatures; j++){
415       fprintf(pFile,"%f ", A[i].featureVec[j]);
416     }
417
418     fprintf(pFile, "%lf ", A[i].E);
419     fprintf(pFile, "%lf %lf %lf %lf ", A[i].action.x, A[i].action.y, A[i].action.theta, A[i].action.vel);
420     fprintf(pFile, "%lf %lf %lf %lf\n", A[i].last.x, A[i].last.y, A[i].last.theta, A[i].last.vel);
421   }
422   */
423
424   for (int i=0; i<numNodes; i++) {
425     fprintf(pFile, "%d %d %d %d %d ",A[i].numObservations, A[i].nodeLabel, A[i].numConnections, A[i].reward, A[i
      ].ancestor);
426
427     fprintf(pFile, "%lf ", A[i].Q);
428
429     fprintf(pFile, "%lf ", A[i].E);
430     fprintf(pFile, "%lf %lf %lf ", A[i].action.x, A[i].action.y, A[i].action.theta);
431     fprintf(pFile, "%lf %lf %lf ", A[i].last.x, A[i].last.y, A[i].last.theta);
432
433     for (int j=0; j<numFeatures; j++){
434       fprintf(pFile,"%f ", A[i].featureVec[j]);
435     }
436
437
438     fprintf(pFile, "\n");
439   }
440
441
442
443   fclose(pFile);
444
445 } // writeA
446
447 void write_C(const char* fname, std::vector<connection> &C)
448 {
449   int numConx = C.size();
450   FILE *pFile;
451   pFile = fopen(fname,"w");
452
453   for (int i=0; i<numConx; i++) {
454     fprintf(pFile, "%d %d %d %f\n", C[i].v1, C[i].v2, C[i].age, C[i].length);
455   }
456   fclose(pFile);
457
458 } // writeC
459
460
461
462 // ------------------------------------------------------------------
463 // Function name:  check4Connection
464 // Description:  This function checks a connection between two nodes (in [C]).
465 // If none exists, it establishes one.  If one exists, it refreshes
466 // the its age (set age = 0).  The function then updates the number of
467 // connections (in [A]) for each of the vertex nodes of the new
468 // connections.
469 // ------------------------------------------------------------------
470 void check4Connection(std::vector<refNode> &A,
471 std::vector<connection> &C, std::vector<distPt> &Dv)
472 {
473   int s1_label = Dv[0].nodeLabel;
474   int s2_label = Dv[1].nodeLabel;
475
476   // printf("D[0].nodeLabel = %d, D[1].nodeLabel = %d \n", s1_label, s2_label);
477
478   int numConx = C.size();
479   int numNodes = A.size();
480   bool connectionExists = false;
481   connection newConx;
482
483   // Check for / refresh an existing connection.
484   for(int i=0; i<numConx; i++) {
485     if (( (C[i].v1==s1_label) && (C[i].v2==s2_label) ) ||
486         ( (C[i].v1==s2_label) && (C[i].v2==s1_label) ))
```

```
487     {
488       connectionExists = true;
489       C[i].age = 0;
490     }
491   }
492   // Establish a connection if none exists.
493   if (connectionExists == false)
494   {
495     newConx.v1 = s1_label;
496     newConx.v2 = s2_label;
497     newConx.age = 0;
498     newConx.length = 1.0;
499     C.push_back(newConx);
500
501     // Update connection counts in [A].
502     for(int i=0; i<numNodes; i++) {
503       if(A[i].nodeLabel == s1_label) {A[i].numConnections++;}
504       if(A[i].nodeLabel == s2_label) {A[i].numConnections++;}
505     }
506   }
507
508 } // check4Connection
509
510 // ------------------------------------------------------------------
511 // Function Name:  adjustWinner
512 // Description:  This function adds to the error of the node closest
513 // to the input data vector.
514 // ------------------------------------------------------------------
515 void adjustWinner(std::vector<refNode> &A, std::vector<distPt> &Dv,
516 std::vector<double> v_in, const double &ep_w)
517 {
518   int numNodes = A.size();
519   int s1_label = Dv[0].nodeLabel;
520   double s1_distance = Dv[0].distance;
521   int numFeatures = v_in.size();
522
523   for(int i=0; i<numNodes; i++) {
524     if (A[i].nodeLabel == s1_label)
525     {
526       // Step 6:  Adjust the winner's local error.
527         A[i].E += pow(s1_distance, 2);
528
529       // Step 7:  Move the winner toward the input vector
530       // by a fraction of its current distance (if reward is not HOT).
531       if (A[i].reward != 0) {
532         for(int j=0; j<numFeatures; j++) {
533           A[i].featureVec[j] += ep_w * (v_in[j] - A[i].featureVec[j]);
534         } // for j
535       }
536
537
538     } // if
539   } // for i
540 } // adjustWinner
541
542
543 // ------------------------------------------------------------------
544 // Function name:  adjustNeighbors
545 // Description:  This function moves the winner's topological
546 // neighbors toward the input by a fraction (ep_n) of their distance
547 // to the input vector.  The ages of all connections emanating from
548 // the winner are incremented.
549 // ------------------------------------------------------------------
550 void adjustNeighbors(std::vector<refNode> &A, std::vector<connection> &C,
551 std::vector<distPt> &Dv, std::vector<double> &v_in, const double &ep_n)
552 {
553   int neighbor = -99;
554   int numCnx = C.size();
555   int numNodes = A.size();
556   int numFeatures = v_in.size();
557   int s1_label = Dv[0].nodeLabel;
558
559
560   // The found_nbr variable fixes a bug discovered on 9/3/2012.
561   // The [A] matrix neighbors were being updated for every entry
562   // in [C] regardless of whether the elements in [C] were neighbors.
563   // The bug'ed soruce code can be found in Archive/src_03Sep12...
564   // This code appears to work better for hoodRadius < mean but
565   // worse for hoodRadius = all neighbors. (9/3/2012).
566   bool found_nbr;
567
568   for (int i=0; i<numCnx; i++) {
569
570     found_nbr = false;
571     neighbor = -99;
572
573     // Check one end.  Increment connection age.
```

```
    if (C[i].v1 == s1_label) {

      found_nbr = true;
      neighbor = C[i].v2;
      C[i].age += 1;
    }
    // Check the other end.  Increment connection age.
    if (C[i].v2 == s1_label) {

      found_nbr = true;
      neighbor = C[i].v1;
      C[i].age += 1;
    }

    // Move (adapt) the neighbor.
    if (found_nbr == true)
    {
      for (int j=0; j<numNodes; j++) {
        if (A[j].nodeLabel == neighbor) {
          for(int k=0; k<numFeatures; k++) {
            A[j].featureVec[k] += ep_n * (v_in[k] - A[j].featureVec[k]);
          }
        }
      }
    }


  } // for i

} // adjustNeighbors

// ------------------------------------------------------------------
// Function name:  removeOldConnections
// Description:  This function removes connections which have aged
// beyond a prescribed maximum (ageMax).  If this leaves any nodes
// with no connections, those nodes are deleted also.
// ------------------------------------------------------------------
void removeOldConnections(std::vector<refNode> &A,
std::vector<connection> &C, const int ageMax)
{
  int numNodes = A.size();
  int numConx = C.size();
  int vertex1 = -99;
  int vertex2 = -99;
  std::vector<refNode> newA(0);
  std::vector<connection> newC(0);


  for(int i=0; i<numConx; i++) {
    if (C[i].age <= ageMax) {
      // Build a new [C] of young connections.
      newC.push_back(C[i]);
    }
    else {

      // Find vertices of old connections
      vertex1 = C[i].v1;
      vertex2 = C[i].v2;

      // Decrement the connection counts for vertex nodes.
      for(int j=0; j<numNodes; j++) {
        if (A[j].nodeLabel == vertex1) {
          A[j].numConnections -= 1;
        }
        if (A[j].nodeLabel == vertex2) {
          A[j].numConnections -= 1;
        }
      }
    }
  }

  C.clear();
  C = newC;

  // Build a new [A] of nodes with >0 connections.
  for(int j=0; j<numNodes; j++) {
    if (A[j].numConnections > 0) {
      newA.push_back(A[j]);
    }
  }

  // Put new [A] into old [A].
  A.clear();
  A = newA;

} // removeOldConnections

```

```
661
662 // ------------------------------------------------------------------
663 // Functions:  print_refNodes, printC
664 // Description:  Print out the [A] (or other vector of reference nodes)
665 //  and [C] matrix for diagnostics.
666 // ------------------------------------------------------------------
667 void printC(std::vector<connection> &C)
668 {
669   int numConx = C.size();
670   for(int i=0; i<numConx; i++) {
671     printf("C[%d]:  v1 = %d, v2 = %d \n", i, C[i].v1, C[i].v2);
672   }
673 } // printC
674
675 void print_refNodes(std::vector<refNode> &A, const char* nodeListName)
676 {
677   int numNodes = A.size();
678   int numFeatures = A[0].featureVec.size();
679
680
681   printf("---------------------------------------------\n");
682   printf(" Printing vector of refNodes: %s \n", nodeListName);
683   printf("---------------------------------------------\n");
684
685   for(int i=0; i<numNodes; i++) {
686     printf("Node number %d: \n", i);
687     printf("rwd = %d\n", A[i].reward);
688     printf("Q = %6.3f\n", A[i].Q);
689     printf("nodeLabel = %d\n", A[i].nodeLabel);
690     printf("numCnx = %d\n", A[i].numConnections);
691     printf("E = %8.5f\n", A[i].E);
692     printf("fvec[0] = %8.5f, ", A[i].featureVec[0]);
693     printf("fvec[%2d] = %8.5f\n", numFeatures-1, A[i].featureVec[numFeatures-1]);
694     printf("action: x = %5.3f, y = %5.3f, theta = %5.3f\n", A[i].action.x, A[i].action.y, A[i].action.theta);
695     printf("last:   x = %5.3f, y = %5.3f, theta = %5.3f\n", A[i].last.x, A[i].last.y, A[i].last.theta);
696     printf("\n");
697   }
698 } // print_refNodes
699
700 // -------------------------------------------------------------------
701 // Function name:  interpolateNodes
702 // Description:  This function sets parameters for a new node to be
703 // inserted between the nearest refNode and its nearest neighbor.  It
704 // accepts 2 input refNodes and produces an interpolated output refNode.
705 // -------------------------------------------------------------------
706 void interpolateNodes(refNode &q, refNode &f, refNode &r)
707 {
708
709   int numFeatures = q.featureVec.size();
710   r.featureVec.clear();
711
712   // Interpolate featureVecs between f and q
713   for(int i=0; i<numFeatures; i++) {
714     r.featureVec.push_back( (q.featureVec[i] + f.featureVec[i])/2.0 );
715   }
716
717
718
719   // Interpolate actionConfig between f and q.
720   r.action.x     = 0; // (q.action.x     + f.action.x    )/2.0;
721   r.action.y     = 0; // (q.action.y     + f.action.y    )/2.0;
722   r.action.theta = 0; // (q.action.theta + f.action.theta)/2.0;
723   r.action.vel   = 0; // (q.action.vel   + f.action.vel  )/2.0;
724
725   r.Q = calc_Q(r);
726
727   r.last.x = 0; // q.last.x;
728   r.last.y = 0; // q.last.y;
729   r.last.theta = 0; // q.last.theta;
730   r.last.vel = 0; // q.last.vel;
731
732   // Interpolate last actions between f and q
733   // r.last.x     = (q.last.x     + f.last.x    )/2.0;
734   // r.last.y     = (q.last.y     + f.last.y    )/2.0;
735   // r.last.theta = (q.last.theta + f.last.theta)/2.0;
736   // r.last.vel   = (q.last.vel   + f.last.vel  )/2.0;
737
738   // Initialize reward pessimistically.
739   // May not be the right thing to do.
740    r.reward = -1;
741
742
743   r.numObservations = q.numObservations;
744
745 } // interpolateNodes
746
747 // -------------------------------------------------------------------
```

```
// Function name:  calcAvgError
// Description:  This function calculates the average node error
// across all nodes in [A].
// ------------------------------------------------------------------
double calcAvgError(std::vector<refNode> &A)
{
  int numNodes = A.size();
  double avgE = 0.0;

  for(int i=0; i<numNodes; i++)
  {
    avgE += A[i].E;
  }

  avgE = (double) avgE/numNodes;

  return(avgE);
}

// ------------------------------------------------------------------
// Function name:  insertNewNode
// Description:  This function inserts a new node into [A].  [C] is
// updated accordingly.  Insertion occurs ever lamda input signals.
// ------------------------------------------------------------------
void insertNewNode(std::vector<refNode> &A, std::vector<connection> &C,
  const int &lambda, const double &alpha, const int maxNodeCnt)
{

  int numNodes = A.size();
  int numCnx = C.size();
  std::vector<int> Nbrs(0); // Set of neighbors of q.
  refNode q, f, r;
  connection C_rf, C_rq;
  // double avgE;
  int numObservations = A[0].numObservations + 1;

  // Calculate average Node Error
  // avgE = calcAvgError(A);


  if((numObservations % lambda == 0) && (numNodes < maxNodeCnt))  {
  // if(numObservations % lambda == 0)  {

    // printA(1,A);

    // Find the node with maximum accumulated error.
    double maxError = -99.0;
    for(int i=0; i<numNodes; i++) {
      if (A[i].E > maxError) {
        maxError = A[i].E;
        q = A[i];
      }
    }


    // Find the neighbors of q.
    for (int i=0; i<numCnx; i++) {
      if (C[i].v1 == q.nodeLabel) {Nbrs.push_back(C[i].v2);}
      if (C[i].v2 == q.nodeLabel) {Nbrs.push_back(C[i].v1);}
    }

    // Find the neighbor f of q with max error.
    int numNeighbors = Nbrs.size();
    maxError = -99.0;
    for(int i=0; i<numNeighbors; i++) {
      for(int j=0; j<numNodes; j++) {
        if ((A[j].nodeLabel == Nbrs[i]) && (A[j].E > maxError)) {
          maxError = A[j].E;
          f = A[j];
        }
      }
    }

    // printA(2,A);

    // Generate a new node r.
    // Interpolate the feature vector and goalConfig between f and q.
    interpolateNodes(q, f, r);

    // printA(1,A);

    // Get the next node label.
    int maxNodeLabel = -1;
    for(int i=0; i<numNodes; i++){
      if (A[i].nodeLabel > maxNodeLabel) {maxNodeLabel = A[i].nodeLabel;}
    }
    int newLabel = maxNodeLabel + 1;
```

```
835     r.nodeLabel = newLabel;
836     r.ancestor = newLabel;
837     r.numConnections = 2; // Connect to q and f (done below).
838     // The node r is added below after its error is calculated.
839
840     // Remove the original connection between q and f.
841     // printC(1,C);
842     int cnx2delete = -1;
843     for (int i=0; i<numCnx; i++) {
844       if  ( ((C[i].v1==q.nodeLabel) && (C[i].v2==f.nodeLabel)) ||
845             ((C[i].v1==f.nodeLabel) && (C[i].v2==q.nodeLabel)) )
846       {
847         cnx2delete = i;
848       }
849     }
850     C.erase(C.begin() + cnx2delete);
851     // printC(2,C);
852
853     // Add new connections: {r,q} and {r,f}.
854     // printf("q=%d, f=%d, maxNodeLabel=%d \n",q, f, maxNodeLabel);
855     C_rf.v1 = newLabel; C_rf.v2 = f.nodeLabel; C_rf.age = 0; C_rf.length = 1.0;
856     C_rq.v1 = newLabel; C_rq.v2 = q.nodeLabel; C_rq.age = 0; C_rq.length = 1.0;
857     C.push_back(C_rf);
858     C.push_back(C_rq);
859
860     // printC(3,C);
861
862     // Decrease the error of q and f by a fraction (alpha).
863     // Interpolate the error of r from q and f.
864     double Er = 0.0;
865     for(int i=0; i<numNodes; i++) {
866       if (A[i].nodeLabel == f.nodeLabel) {
867         A[i].E -= alpha*(A[i].E);
868         Er += A[i].E/2.0;
869       }
870       if (A[i].nodeLabel == q.nodeLabel) {
871         A[i].E -= alpha*(A[i].E);
872         Er += A[i].E/2.0;
873       }
874     }
875     r.E = Er;
876     A.push_back(r);
877
878
879   } // if [new node needed]
880
881 } // insertNewNode
882
883
884 // ------------------------------------------------------------------
885 // Function name:  decreaseNodeError
886 // Description:  This function decreases the error variables on all
887 // nodes by a factor (beta).  This function also increments the number
888 // of observations (stored in [A]).
889 // ------------------------------------------------------------------
890 void decreaseNodeError(std::vector<refNode> &A, const double &beta)
891 {
892   int numNodes = A.size();
893
894   for(int i=0; i<numNodes; i++)
895   {
896     A[i].E -= (beta*A[i].E);
897     A[i].numObservations += 1;
898   }
899
900 }
901
902
903
904 // ------------------------------------------------------------------
905 // Function name:  getNN
906 // Description:  This function returns the reference node closest
907 // to the input vector (i.e. it's Nearest Neighbor (NN) ).  This node's
908 // attached goalConfig comprises the best guess at the robotic goalConfig
909 // for the input gesture.
910 //
911 // 6/30/2012 - the Q-Learning Q value is also calculated as the length
912 // of the goal configuration vector from the origin.  This is based on the
913 // assertion that longer config vectors reflect the most positive past
914 // rewards.
915 // ------------------------------------------------------------------
916 refNode getNN(std::vector<distPt> &Dv, std::vector<refNode> &A)
917 {
918   refNode NN;
919   int numNodes = A.size();
920
921   int nearestNodeLabel = Dv[0].nodeLabel;
```

```
  for(int i=0; i<numNodes; i++) {
    if (A[i].nodeLabel == nearestNodeLabel) {
      NN = A[i];
    }
  }

  // Estimate Q value -> length of NN.action as an (x, y, theta) vector.
  // NN.Q = calc_Q(NN);


  return(NN);
}

// ------------------------------------------------------------------
// Function name:  getNeighbors
// Description:  This function returns the set of reference nodes N
// connected to the NN of the input vector.
// ------------------------------------------------------------------
void getNeighbors(refNode &NN, std::vector<refNode> &N, std::vector<refNode> &A, std::vector<connection> &C)
{
  N.clear();
  int nodeLabel = NN.nodeLabel;

  int numConx = C.size();
  int numNodes = A.size();
  std::vector<int> nodeList(0);

  // Build the list of neighbor nodeLabels.
  for(int i=0; i<numConx; i++)
  {
    if (C[i].v1 == nodeLabel) {nodeList.push_back(C[i].v2);}
    if (C[i].v2 == nodeLabel) {nodeList.push_back(C[i].v1);}
  }

  int numNeighbors = nodeList.size();

  /*
  printf("Node %d has %d neighbors:  these include nodes: ", nodeLabel, numNeighbors);
  for (int i=0; i<numNeighbors; i++)
  {
    printf("%d ", nodeList[i]);
  }
  printf("\n");

  printf("Initially, N.size = %d, ", N.size());
  */


  // Build the vector of neighbors
  for (int i=0; i<numNeighbors; i++)
  {
    for (int j=0; j<numNodes; j++)
    {
      if (nodeList[i] == A[j].nodeLabel)
      {
        N.push_back(A[j]);
      }
    }
  }

  // printf("after buiding, N.size = %d.\n", N.size());

}

// ------------------------------------------------------------------
// Function name:  gng
// Description:  This function performs the GNG algorithm.
// A: the set of reference nodes.
// C: the set of all connections.
// ------------------------------------------------------------------
refNode gng(const char* A_fname, const char* C_fname,
  std::vector<refNode> &A, std::vector<connection> &C,
  std::vector<double> &vec_in, std::vector<refNode> &N,
  const int &lambda, const int maxNodeCnt)
{
  std::vector<distPt> Dv(0);
  double ep_w = 0.05;
  double ep_n = 0.0006; // Fritzke uses 0.0006
  int ageMax = 88; // Fritzke uses 88; use a large number for resistance distance
  double alpha = 0.5;
  double beta = 0.0005; // Fritzke uses 0.0005
  int numFeatures = vec_in.size();

  // Return the nearest neighbor (NN) of vec_in.
  refNode NN;

```

```
  // Step 1: Read in [A] and [C] matrices or use existing ones.
  // These functions default to using existing [A] and [C] matrices having
  // non-zero sizes.  Otherwise fnames are read in.
  read_A(A_fname, A, numFeatures);
  read_C(C_fname, C);

  // Step 2: Select a vector (use the vector passed in).

  // Step 3: Find the input vector's two closest neighbors
  // from among the collection of refNodes [A].
  get2ClosestNodes(A, vec_in, Dv);

  // Step 4: Establish or refresh connections (set age=0)
  // between the two nearest nodes.
  check4Connection(A, C, Dv);

  // Step 5:  Add to the nearest node (Dv[0]) local error.
  // Step 6:  Move the winner a fraction (ep_w) of its current distance
  // toward the input vector.
  adjustWinner(A, Dv, vec_in, ep_w);

  // Step 7:  Move the winner's topolgical neighbors toward
  // the input by a fraction (ep_n) of their distance to it.
  // Increment the ages of all connections emmanating from the winner.
  adjustNeighbors(A, C, Dv, vec_in, ep_n);

  // Step 8:  Remove edges with age greater than ageMax.  If this
  // leaves any nodes with no connections, remove the nodes.
  removeOldConnections(A, C, ageMax);

  // Step 9:  Insert a new node (if necessary based on lambda).
  insertNewNode(A, C, lambda, alpha, maxNodeCnt);

  // Step 10:  Decrease error of all nodes.
  // Also, increment numObservations for all nodes in [A] here.
  decreaseNodeError(A, beta);

  // Return the nearest neighbor from just prior to node movement.
  NN = getNN(Dv, A);
  getNeighbors(NN, N, A, C);
  return(NN);



} // gng
```

A.1.5 kinect_includes.h

```
#include <stdio.h>
#include <iostream>
#include <string>
#include <istream>
#include <cstring>
#include <time.h>
#include <ros/ros.h>
#include <gtest/gtest.h>
#include <ros/init.h>
#include <std_msgs/String.h>
#include <std_msgs/Float64.h>
#include <sensor_msgs/PointCloud.h>
#include <sensor_msgs/PointCloud2.h>
#include <sensor_msgs/point_cloud_conversion.h>
#include <opencv/highgui.h>
#include <sensor_msgs/Image.h>
#include <cv_bridge/cv_bridge.h>
#include <opencv/cv.h>
#include <opencv/cvaux.h>
#include <sstream>
#include <boost/foreach.hpp>
#include <rosbag/bag.h>
#include <rosbag/view.h>
#include <rosbag/query.h>
#include <turtlesim/Pose.h>
#include <turtlesim/Velocity.h>
#include <cmath>
#include <math.h>
#include <angles/angles.h>
#include <body_msgs/Skeletons.h>
#include <actionlib/client/simple_action_client.h>
#include <actionlib/client/terminal_state.h>
#include <turtleControl/moveTurtleAction.h>
```

A.1.6 points.h

```
// ------------------------------------------------------------------
// Class name:  pt3
// Descriptin:  This class defines a 3-d point.
// ------------------------------------------------------------------
class pt3
{

public:
  double x,y,z; // 3d point coordinates.

  // Constructor
  pt3(double x_in = -999.99, double y_in = -999.99, double z_in = -999.99) {
    x = x_in;
    y = y_in;
    z = z_in;
  }

}; //  pt3
// ------------------------------------------------------------------
// Class name:  pt3sph
// Descriptin:  This class defines a 3-d point in spherical
// (r, theta, phi) coordinates.
// ------------------------------------------------------------------
class pt3sph
{

public:
  double r,t,p; // 3d shperical point coordinates.

  // Constructor
  pt3sph(double r_in = -999.99, double t_in = -999.99, double p_in = -999.99) {
    r = r_in;
    t = t_in;
    p = p_in;
  }

}; //  pt3sph

// ------------------------------------------------------------------
// Class name:  pt_curvature
// Description:  This class defines a 2-d point containing a curvature
// value and its frame number in a time sequence.
// ------------------------------------------------------------------
class pt_curvature
{

public:
  // 2d point of curvature containing curvature value (k) and frame number.
  double k;
  double frameNum;

  // Constructor
  pt_curvature(double k_in = -999.99, int frameNum_in = -1) {
    k = k_in;
    frameNum = frameNum_in;
  }

}; //  pt_curvature

// ------------------------------------------------------------------
// Functions for comparing curvature fields.
// ------------------------------------------------------------------
bool CompareCurvature(pt_curvature A, pt_curvature B);
bool CompareFrameNum(pt_curvature A, pt_curvature B);
// ------------------------------------------------------------------
// Class name:  dynamic_instant
// Descriptin:  This class defines a dynamic_instant (See Rao et al.)
// consisting of various fields including curvature, frame number, and
// [sign of?] velocity (x,y,z components) for a point in 3-space.
// ------------------------------------------------------------------
class dynamic_instant
{

public:
  double frameNum; // k, frameNum
  double totalFrames; // Use this to normalize frameNum
  pt3 pos; // position vector
  pt3 vel; // velocity vector

  // Constructor
  dynamic_instant() {}
};

// ------------------------------------------------------------------
```

```
// Prototypes
// ----------------------------------------------------------------
void write_PVA_data(const char *PVA_fname,
  std::vector<pt3> &posVec_LH, std::vector<pt3> &velVec_LH,
  std::vector<pt3> &accVec_LH);

void write_POS_data(const char *POS_fname, int &sampleNum,
  int &dataType, std::vector<pt3> &posVec);

void write_DI_data(const char *DI_fname, std::vector<dynamic_instant> &DI_list);

void write_to_descriptor_list(const char *descriptor_list_fname,
  std::vector<double> &vec_in, int &gestureClass);

std::vector<double> find_DIs(const char* DI_fname,
  std::vector<pt3> &posVec, std::vector<pt3> &velVec,
  std::vector<pt3> &accVec, int numInstants);
```

A.1.7 points.cpp

```
#include "kinect_includes.h"
#include "points.h"

// ----------------------------------------------------------------
// Functions for comparing curvature fields.
// ----------------------------------------------------------------
bool CompareCurvature(pt_curvature A, pt_curvature B) {
  return(fabs(A.k) > fabs(B.k));
}

bool CompareFrameNum(pt_curvature A, pt_curvature B) {
  return(A.frameNum < B.frameNum);
}

// ----------------------------------------------------------------
// Function name:  write_PVA_data
// Description:  This function writes a text file containing (P)osition,
// (V)elocity and (A)cceleration data collected from kinect.
// As of 12/28/2011, this includes only left hand (LH) data.
// ----------------------------------------------------------------
void write_PVA_data(const char *PVA_fname,
std::vector<pt3> &posVec_LH, std::vector<pt3> &velVec_LH, std::vector<pt3> &accVec_LH)
{
  int msgCnt = posVec_LH.size();

  FILE * pFile;
  pFile = fopen(PVA_fname, "w");
  for (int i=0; i<msgCnt; i++) {
    fprintf(pFile,"%d %12.8f %12.8f %12.8f %12.8f %12.8f %12.8f %12.8f %12.8f %12.8f \n", i,
      posVec_LH[i].x, posVec_LH[i].y, posVec_LH[i].z,
      velVec_LH[i].x, velVec_LH[i].y, velVec_LH[i].z,
      accVec_LH[i].x, accVec_LH[i].y, accVec_LH[i].z);
  }
  fclose(pFile);

} // write_PVA_data


// ----------------------------------------------------------------
// Function name:  write_POS_data
// Description:  This function writes a text file containing all
// x,y,z points for a connect data stream.
//
// 07/08/2012 - "dataType" = "gestureClass"
// ----------------------------------------------------------------
void write_POS_data(const char *POS_fname, int &sampleNum, int &dataType, std::vector<pt3> &posVec)
{
  int msgCnt = posVec.size();

  FILE * pFile;
  pFile = fopen(POS_fname, "a");
  for (int i=0; i<msgCnt; i++)
  {
    fprintf(pFile, "%d %d %d %12.8f %12.8f %12.8f \n", sampleNum, dataType, i+1, posVec[i].x, posVec[i].y, posVec
      [i].z);
  }
}


// ----------------------------------------------------------------
// Function name:  write_DI_data
// Description:  This function writes a text file containing dynamic
// Instants (DI) generated during a single gesture motion.
// ----------------------------------------------------------------
```

```
64 void write_DI_data(const char *DI_fname, std::vector<dynamic_instant> &DI_list)
65 {
66
67   int numDIs = DI_list.size();
68
69   FILE * pFile;
70   pFile = fopen(DI_fname, "w");
71   for (int i=0; i<numDIs; i++) {
72
73   /* Includes velocities
74     fprintf(pFile,"%12.8f %12.8f %12.8f %12.8f %12.8f %12.8f %12.8f %12.8f\n",
75       DI_list[i].frameNum, DI_list[i].totalFrames,
76       DI_list[i].pos.x, DI_list[i].pos.y, DI_list[i].pos.z,
77       DI_list[i].vel.x, DI_list[i].vel.y, DI_list[i].vel.z);
78   */
79
80     fprintf(pFile,"%12.8f %12.8f %12.8f %12.8f\n",
81       DI_list[i].frameNum, DI_list[i].pos.x,
82       DI_list[i].pos.y, DI_list[i].pos.z);
83
84   }
85   fclose(pFile);
86
87 } // write_DI_data
88
89 // ----------------------------------------------------------------
90 // Function name:  write_to_descriptor_list
91 // Description:  This function appends a new descriptor to an existing file containing
92 // of nx1 descriptors.
93 // ----------------------------------------------------------------
94 void write_to_descriptor_list(const char *descriptor_list_fname, std::vector<double> &vec_in, int &gestureClass)
95 {
96   int numFeatures = vec_in.size();
97
98   FILE * pFile;
99   pFile = fopen(descriptor_list_fname, "a");
100
101   fprintf(pFile,"%d ", gestureClass);
102   for (int i=0; i<numFeatures; i++)
103   {
104     fprintf(pFile,"%12.8f ", vec_in[i]);
105   }
106   fprintf(pFile,"\n");
107
108   fclose(pFile);
109 }
110
111 // ----------------------------------------------------------------
112 // Function name:  find_DIs
113 // Description:  This function finds Dynamic Instants (DI) for
114 // motion signatures collected using a 3d depth image.  The function
115 // returns a vector of the top N frame numbers from an input vector
116 // of curvatures.  The index of curvature value is the frame number.
117 // A single vector consists of five concatenated instances of
118 // (frameNum, x, y, z).  A DI file is also written containing individual
119 // DIs along with velocity at the instants.
120 // ----------------------------------------------------------------
121 std::vector<double> find_DIs(const char* DI_fname,
122   std::vector<pt3> &posVec, std::vector<pt3> &velVec,
123   std::vector<pt3> &accVec, int numInstants)
124 {
125
126   std::vector<pt3> k; // curvatures (abs of acceleration).
127   std::vector<pt_curvature> maxima;
128   pt_curvature local_max;
129   k = accVec;
130   int numPts = accVec.size();
131   std::vector<pt_curvature> max_curvatures(0);
132   std::vector<dynamic_instant> DI_list(0);
133   dynamic_instant my_inst;
134   std::vector<double> vec_in(0);
135
136   // Generate 1D curvature (= abs(acceleration)).
137   for(int i=0; i<numPts; i++) {
138     k[i].x = fabs(accVec[i].x);
139     k[i].y = fabs(accVec[i].y);
140     k[i].z = fabs(accVec[i].z);
141   }
142
143   // Find candidates for local maxima of acceleration.
144   int count = -1;
145   for(int i=1; i<numPts-1; i++) {
146     if ((k[i-1].x < k[i].x) && (k[i+1].x < k[i].x)) {
147       count += 1;
148       local_max.k = k[i].x;
149       local_max.frameNum = (double) i;
150       maxima.push_back(local_max);
```

```
    }
    if ((k[i-1].y < k[i].y) && (k[i+1].y < k[i].y)) {
      count += 1;
      local_max.k = k[i].y;
      local_max.frameNum = (double) i;
      maxima.push_back(local_max);
    }
    if ((k[i-1].z < k[i].z) && (k[i+1].z < k[i].z)) {
      count += 1;
      local_max.k = k[i].z;
      local_max.frameNum = (double) i;
      maxima.push_back(local_max);
    }
  }

  // Sort by curvature
  sort(maxima.begin(), maxima.end(), CompareCurvature);

  // Create a vector of the top numInstants curvatures.
  for(int i=0; i<numInstants; i++) {
    max_curvatures.push_back(maxima[i]);
  }

  // Sort max_curvatures by frameNum (put in time order).
  sort(max_curvatures.begin(), max_curvatures.end(), CompareFrameNum);

  // Generate a list of dynamic_instants.
  for (int i=0; i<numInstants; i++)
  {
    my_inst.frameNum = max_curvatures[i].frameNum;
    my_inst.totalFrames = posVec.size();
    my_inst.pos = posVec[max_curvatures[i].frameNum];
    my_inst.vel = velVec[max_curvatures[i].frameNum];
    DI_list.push_back(my_inst);

    vec_in.push_back(my_inst.frameNum/numPts);
    vec_in.push_back(my_inst.pos.x);
    vec_in.push_back(my_inst.pos.y);
    vec_in.push_back(my_inst.pos.z);

    /*
    printf("Compare:  vec_in[%3d](%6.3f, %6.3f, %6.3f), pos[%3d](%6.3f, %6.3f, %6.3f)\n",
        (int)my_inst.frameNum, my_inst.pos.x, my_inst.pos.y, my_inst.pos.z,
        (int)my_inst.frameNum, posVec[my_inst.frameNum].x, posVec[my_inst.frameNum].y, posVec[my_inst.frameNum].z
     );
    */

  }

  // Experiment:
  // Artificially set the first and last DIs to be
  // the start and end points of the motion.
  /*
  vec_in[0] = 1/numPts;
  vec_in[1] = posVec[0].x;
  vec_in[2] = posVec[0].y;
  vec_in[3] = posVec[0].z;
  vec_in[16] = numPts/numPts;
  vec_in[17] = posVec[numPts - 1].x;
  vec_in[18] = posVec[numPts - 1].y;
  vec_in[19] = posVec[numPts - 1].z;
  */

  // Write out the DI list to a file (OPTIONAL).
  // write_DI_data(DI_fname, DI_list);

  // Convert the DI list to a vector suitable for the GNG algorithm.


  return(vec_in);

}  // find_DIs
```

A.1.8 utilities.h

```
// ----------------------------------------------------------------
// Prototypes
// ----------------------------------------------------------------
double rand01();

double rand_in_range(double a, double b);

void scale01(std::vector<pt3> &din);

void smoothGauss5x1(std::vector<double> &p);
```

```
11
12 void smoothEvolutionTime(std::vector<double> &a, int winSize);
13
14 void smoothMovingAvg(std::vector<double> &a, int winSize);
15
16 void smooth3Dpoints(std::vector<pt3> &q);
17
18 std::vector<double> derivative(std::vector<double> &x, int winSize);
19
20 std::vector<pt3> deriv3D(std::vector<pt3> &d3D, int winSize);
21
22 double round(double d);
23
24 double calc_stdDev(std::vector<double> a);
25
26 double calc_mean(std::vector<double> a);
27
28 double vecLen(std::vector<double> v);
29
30 pt3sph cart2sph(double x, double y, double z);
31
32 pt3 sph2cart(double r, double p, double t);
33
34 void arrayInit(std::vector<std::vector<double> > &a,
35   int rows, int cols, double initVal);
```

A.1.9 utilities.cpp

```
1 #include "kinect_includes.h"
2 #include "points.h"
3
4 // This file contains general utility functions which support
5 // gesture recogntion and the GNG algorithm.
6
7 // ----------------------------------------------------------------
8 // Function name: rand01
9 // Description:  This function generates a random number on [0,1].
10 // ----------------------------------------------------------------
11 double rand01()
12 {
13   // Make sure to issue the command below in main(), not here.
14   // srand(time(NULL));
15
16   double x = (double)rand()/(double)RAND_MAX;
17
18   return(x);
19 }
20
21 // ----------------------------------------------------------------
22 // Function name:  rand_in_range
23 // Description:  This function generates a random number (double) within
24 // a specified range (between a and b parameters).
25 // ----------------------------------------------------------------
26 double rand_in_range(double a, double b)
27 {
28   double max = 0.0;
29   double min = 0.0;
30   double range = 0.0;
31   double x = 0.0;
32
33   if (a == b) {
34     printf ("ERROR (rand_in_range): Input parameters must not be equal.\n");
35     x = rand01();
36   }
37   else
38   {
39     if (a > b) {max = a; min = b;}
40     else {max = b, min = a;
41     }
42     range = max - min;
43
44     x = rand01();
45
46     x = (x * range) + min;
47   }
48
49   return(x);
50
51 }
52 // ----------------------------------------------------------------
53 // Function name:  scale01
54 // Description:  This function scales position data on [0,1].
55 // It uses the largest value in x, y, and z data vectors to generate
56 // a relative scale of all values against the largest changing
```

```
// variable.
// ----------------------------------------------------------------
void scale01(std::vector<pt3> &din)
{
  int numPts = din.size();
  double max_x, max_y, max_z;
  double min_x, min_y, min_z;
  double dif_x, dif_y, dif_z, maxDiff;

  max_x = din[0].x;
  max_y = din[0].y;
  max_z = din[0].z;

  min_x = din[0].x;
  min_y = din[0].y;
  min_z = din[0].z;

  // Find the variable with the maximum swing.
  for(int i=1; i<numPts; i++) {
    if (din[i].x > max_x) {max_x = din[i].x;}
    if (din[i].y > max_y) {max_y = din[i].y;}
    if (din[i].z > max_z) {max_z = din[i].z;}


    if (din[i].x < min_x) {min_x = din[i].x;}
    if (din[i].y < min_y) {min_y = din[i].y;}
    if (din[i].z < min_z) {min_z = din[i].z;}
  }
  dif_x = max_x - min_x;
  dif_y = max_y - min_y;
  dif_z = max_z - min_z;

  maxDiff = dif_x;
  if (dif_y > maxDiff) {maxDiff = dif_y;}
  if (dif_z > maxDiff) {maxDiff = dif_z;}

  // Bring all data vectors to bottom at zero.
  for(int i=0; i<numPts; i++) {
    din[i].x = din[i].x - min_x;
    din[i].y = din[i].y - min_y;
    din[i].z = din[i].z - min_z;
  }
  // Scale all values based on the maximum difference in all variables.
  for(int i=0; i<numPts; i++) {
    din[i].x = (din[i].x)/maxDiff;
    din[i].y = (din[i].y)/maxDiff;
    din[i].z = (din[i].z)/maxDiff;
  }

}

// ----------------------------------------------------------------
// Function name:  smoothGauss5x1
// Description:  This functin performs covolution with a 5x1 fixed
// Gaussian kernel.
// ----------------------------------------------------------------
void smoothGauss5x1(std::vector<double> &p)
{
  int numPts = p.size();
  std::vector<double> pnew(0);

  // Replicate the ends in the original vector - don't worry about borders.
  p[0] = p[2];
  p[1] = p[2];
  p[numPts-1] = p[numPts-3];
  p[numPts-2] = p[numPts-3];

  // Operate on a new copy of the vector.
  pnew = p;

  // Kernel = {1 4 6 4 1}/16
  for(int i=2; i<(numPts-2); i++) {
    pnew[i] = (p[i-2] + 4*p[i-1] + 6*p[i] + 4*p[i+1] + p[i+1])/16;
  }
  // Replicate the ends in the smoothed signal - don't worry about borders.
  pnew[0] = pnew[2];
  pnew[1] = pnew[2];
  pnew[numPts-1] = pnew[numPts-3];
  pnew[numPts-2] = pnew[numPts-3];

  // The original vector is now smoothed.
  p = pnew;
}


// ----------------------------------------------------------------
// Function name:  smoothEvolutionTime
```

```
// Description:  This function applies an evolution time window
// to each point in a vector.  An eveN-sized window is input.
// Odd window sizes are rounded down.  The function calculates
// the average between the data point value at the beginning and
// the end of the evolution time window.
//
// Question:  Could a moving average be better?
// ----------------------------------------------------------------
void smoothEvolutionTime(std::vector<double> &a, int winSize)
{
  int numPts = a.size();
  int halfWin = (int) round(winSize/2);
  std::vector<double> new_a(0);
  double avg;

  for(int i=0; i<numPts; i++)
  {
    if (i < halfWin)
    {
      new_a.push_back(a[i]);
    }
    else if (i + halfWin > numPts-1)
    {
      new_a.push_back(a[i]);
    }
    else
    {
      avg = (a[i-halfWin] + a[i+halfWin])/2.0;
      new_a.push_back(avg);
    }
  }

  a = new_a;


} // smoothEvolutionTime

// ----------------------------------------------------------------
// Function name:  smoothMovingAvg
// Description:  This function calculates the moving average over
// a window of points.  Use only odd window sizes.
// ----------------------------------------------------------------
void smoothMovingAvg(std::vector<double> &a, int winSize)
{
  int numPts = a.size();
  int halfWin = (int) floor(winSize/2);
  std::vector<double> new_a(0);
  double avg;
  double sum;



  for(int i=0; i<numPts-1; i++)
  {
    sum = 0;
    for (int j=(i-halfWin); j<=(i+halfWin); j++)
    {
      if ((j<0) || (j>=numPts))
      {
        // Replicate end points when window falls outside vector dimensions.
        sum = sum + a[i];
      }
      else
      {
        sum = sum + a[j];
      }
    }
    avg = sum/winSize;
    new_a.push_back(avg);
  }

  a = new_a;

} // smoothMovingAvg

// ----------------------------------------------------------------
// Function name:  smooth3Dpoints
// Description:  This function breaks a vector of 3d points into its
// constituent x,y,z signals and smooths each one with a fixed
// Gaussian kernel.
// ----------------------------------------------------------------
void smooth3Dpoints(std::vector<pt3> &q)
{
  int numPts = q.size();
  std::vector<double> q_x(0);
  std::vector<double> q_y(0);
  std::vector<double> q_z(0);
```

```
  for(int i=0; i<numPts; i++) {
    q_x.push_back(q[i].x);
    q_y.push_back(q[i].y);
    q_z.push_back(q[i].z);
  }

  // Smooth each component signal.
  smoothGauss5x1(q_x);
  smoothGauss5x1(q_y);
  smoothGauss5x1(q_z);

/*
  // Smooth again using evolution time.
  smoothEvolutionTime(q_x,7);
  smoothEvolutionTime(q_y,7);
  smoothEvolutionTime(q_z,7);
*/

  // Smooth again using moving average time.
  smoothMovingAvg(q_x,7);
  smoothMovingAvg(q_y,7);
  smoothMovingAvg(q_z,7);

  for(int i=0; i<numPts; i++) {
    q[i].x = q_x[i];
    q[i].y = q_y[i];
    q[i].z = q_z[i];
  }
}

// ----------------------------------------------------------------
// Function name:  derivative
// Description:  This function takes the derivative of a vector
// using an evolution window. This is a qualitative slope of a
// function by subtracting values of <time>-sequence data over a
// window of evolution.  The initial application for this function
// is to take velocity and acceleration of position data from a
// Kinect depth image.
// ----------------------------------------------------------------
std::vector<double> derivative(std::vector<double> &x, int winSize)
{
  int numPts = x.size();
  std::vector<double> xDot(0);
  double halfWin = floor(winSize/2.0);
  double slope = 0.0;

  for(int i=0; i<numPts-1; i++) {
    if ((winSize == 0) || (winSize == 1)) {
      slope = x[i+1] - x[i];
    }
    else if (i<halfWin) {
      slope = x[2*halfWin] - x[0];
    }
    else if (i>numPts-halfWin-1) {
      slope = slope; // hold last value
    }
    else {
      slope = x[i+halfWin] - x[i-halfWin];
    }
    xDot.push_back(slope);
  } // for i

  return(xDot);
}


// ----------------------------------------------------------------
// Function name:  deriv3D
// Description:  This function takes the derivatives of the x, y and z
// components of a pt3 vector.  It explodes the pt3 into its components
// and calls the derivative function 3 times.
// ----------------------------------------------------------------
std::vector<pt3> deriv3D(std::vector<pt3> &d3D, int winSize)
{
  int numPts = d3D.size();
  std::vector<double> d1D_x(0);
  std::vector<double> d1D_y(0);
  std::vector<double> d1D_z(0);
  std::vector<pt3> d3D_dot(0);
  pt3 newPt;

  // Extract x, y and z components of 3D data.
  for(int i=0; i<numPts; i++) {
    d1D_x.push_back(d3D[i].x);
    d1D_y.push_back(d3D[i].y);
    d1D_z.push_back(d3D[i].z);
```

```
  }

  // Take the derivatives of each component.
  std::vector<double> xDot = derivative(d1D_x, winSize);
  std::vector<double> yDot = derivative(d1D_y, winSize);
  std::vector<double> zDot = derivative(d1D_z, winSize);

  for(int i=0; i<numPts; i++) {
    newPt.x = xDot[i];
    newPt.y = yDot[i];
    newPt.z = zDot[i];
    d3D_dot.push_back(newPt);
  }

  return(d3D_dot);
}


// ----------------------------------------------------------------
// Function name:  round
// Description:  This function rounds a double to the nearest
// integer.
// ----------------------------------------------------------------
double round(double d)
{
  return( (double)floor(d + 0.5) );
}

// ----------------------------------------------------------------
// Function name:  calc_mean
// Description:  This function computes the mean for
// a vector of doubles.
// ----------------------------------------------------------------
double calc_mean(std::vector<double> a)
{
  int numVals = a.size();
  double mean;
  double sum = 0;

  for (int i=0; i<numVals; i++)
  {
    sum += a[i];
  }
  mean = sum/numVals;

  return(mean);
}

// ----------------------------------------------------------------
// Function name:  calc_stdDev
// Description:  This function computes the standard deviation for
// a vector of doubles.
//
// Formula found at wikipedia/Standard_Deviation.
// ----------------------------------------------------------------
double calc_stdDev(std::vector<double> a)
{
  int numVals = a.size();
  double sigma;
  // double meanOfSquares = 0;
  // double squareOfMean = 0;
  double sum = 0;
  double bias = 1/(numVals - 1);

  // Sample Standard Deviation (Bessel's Correction):
  double mean = calc_mean(a);
  printf("mean = %f\n", mean);
  for (int i=0; i<numVals; i++)
  {
    sum += pow(a[i]-mean,2);
  }
  sigma = sqrt(bias * sum);

  /*
  // Maximum likelihood method.
  for (int i=0; i<numVals; i++)
  {
    meanOfSquares = meanOfSquares + pow(a[i],2);
    squareOfMean = squareOfMean + a[i];
  }
  meanOfSquares = meanOfSquares/numVals;
  squareOfMean = pow(squareOfMean/numVals,2);

  sigma = sqrt( meanOfSquares - squareOfMean );
  */

  return(sigma);
```

```
}

// ----------------------------------------------------------------
// Function Name:  vecLen
// Descriotion:  This function computes the length of an n-dimensional
// vector.
// ----------------------------------------------------------------
double vecLen(std::vector<double> v)
{
  int numVals = v.size();
  double sum = 0;
  double vLength = 0;

  for (int i=0; i<numVals; i++)
  {
    sum = sum + pow(v[i],2);
  }

  vLength = sqrt(sum);

  return(vLength);

}

// ----------------------------------------------------------------
// Function Name:  cart2sph
// Descriotion:  This function converts cartesian coordinates
// (x,y,z) to shperical coordinates (r,p,t).
// ----------------------------------------------------------------
pt3sph cart2sph(double x, double y, double z)
{
  pt3sph s;

  // Note:  MATLAB cart2sph calls
  // phi = elevation, theta = azimuth

  s.r = sqrt( pow(x,2) + pow(y,2) + pow(z,2) ); // radius
  s.t = (s.r==0) ? 0 : acos(z/s.r); // theta: inclination
  s.p = atan2(y,x); // phi: azimuth (counterclockwise form +x)

  return(s);
}

// ----------------------------------------------------------------
// Function Name:  shp2cart
// Descriotion:  This function converts sherical coordinates
// (r,t,p) to shperical coordinates (x,y,z).
// ----------------------------------------------------------------
pt3 sph2cart(double r, double phi, double theta)
{
  pt3 c;

  c.x = r * sin(theta) * cos(phi);
  c.y = r * sin(theta) * sin(phi);
  c.z = r * cos(theta);

  return(c);
}

// ----------------------------------------------------------------
// Function name:  arrayInit
// Description:  This function initializes a 2D array structured
// as a vector of vectors<double>.
// ----------------------------------------------------------------
std::vector<std::vector<double> > arrayInit(int rows, int cols, double initVal)
{
  std::vector<double> newRow(0);
  std::vector<std::vector<double> > a(0);
  int i;

  for(i=0; i<cols; i++) {
    newRow.push_back(initVal);
  }
  for(i=0; i<rows; i++) {
    a.push_back(newRow);
  }
  newRow.clear();

  return(a);

}
```

A.1.10 assertions.h

```
// This file contains ASSERTIONS to flag runtime errors.
```

```
2
3 // ----------------------------------------------------------------
4 // Function name:  ASSN_duplicateConx
5 // Description:  This function checks for connections which originate
6 // and terminate on the same node.
7 // ----------------------------------------------------------------
8 void ASSN_duplicateConx(std::vector<connection> &C)
9 {
10   int numConx = C.size();
11
12   for (int i=0; i<numConx; i++) {
13     if(C[i].v1 == C[i].v2) {
14       printf("ASSERTION:  Duplicate Conx: C[%d].v1 = %d, C[%d].v2 = %d\n",
15         i, C[i].v1, i, C[i].v2);
16     }
17   }
18 }
19
20
21 // ----------------------------------------------------------------
22 // Function name:  ASSN_gestureClassScan
23 // Description:  This function checks for bad command line scans.
24 // ----------------------------------------------------------------
25 void ASSN_gestureClassScan(int scanResult)
26 {
27   if (scanResult != 1) {
28     printf("ASSERTION:  Bad gesture class scan.  Num results = %d\n",
29       scanResult);
30   }
31 }
32
33 // ----------------------------------------------------------------
34 // Function name:  ASSN_feedbackCheck
35 // Description:  This function checks for a feedback value out of bounds.
36 // If the feedback is out of bounds, the feedback is set to 5 (no motion).
37 // ----------------------------------------------------------------
38 void ASSN_feedbackCheck(std::vector<int> &feedback)
39 {
40   int minFeedback = 0;
41   int maxFeedback = 10;
42
43   int numFeedbacks = feedback.size();
44
45   for (int i=0; i<numFeedbacks; i++) {
46
47     if (feedback[i] < minFeedback || feedback[i] > maxFeedback)
48     {
49       printf("ASSERTION:  Bad feedback value found.  Feedback = %d\n",
50         feedback[i]);
51
52       feedback[i] = 5;
53     }
54   }
55
56 } // ASSN_feedbackCheck
```

A.1.11 lists.h

```
1 #include "kinect_includes.h"
2
3 class descriptor
4 {
5
6 public:
7   std::vector<double> featureVec;
8   int classNum;
9
10   // Constructor
11   descriptor(): featureVec(0) {};
12
13
14 }; //  descriptorList
15
16 // ----------------------------------------------------------------
17 // Function name: read_DIs
18 // Description:  This function reads in a list of Dynamic Instants (DIs)
19 // from a file to a descriptorList data structure.
20 //
21 // The file structure is assumed to be:
22 // (int class#),(std::vector<double> featureVec[numFeatures]) per line.
23 // ----------------------------------------------------------------
24 std::vector<descriptor> read_descriptor_list(const char* fname, const int &numFeatures)
25 {
26   std::vector<descriptor> descriptor_list(0);
27   std::vector<double> featureVec(0);
```

```
  descriptor newD;
  int c; // class number
  double feature;

  FILE *pFile;
  pFile = fopen(fname,"r");

  if (pFile == NULL)
  {
    printf("File %s does not exist.\n", fname);
  }
  else // Read in
  {
    while(fscanf(pFile, "%d", &c) != EOF)
    {
      newD.classNum = c;

      newD.featureVec.clear();
      for(int i=0; i<numFeatures; i++)
      {
        if(fscanf(pFile, "%lf", &feature) != EOF)
        {
          newD.featureVec.push_back(feature);
        }
      }
    descriptor_list.push_back(newD);
    } // while reading a line
  }

  return(descriptor_list);

} // read_A
```

A.1.12 graphs.h

```
#include "kinect_includes.h"
#include "matrixOps.h"

// ------------------------------------------------------------------
// This .h file contains tools related to graph theory.
// ------------------------------------------------------------------


// ------------------------------------------------------------------
// Function name:  getMaxNodeLabelA
// Description:  This function scans the [A] matrix for the largest
// nodeLabel.
// ------------------------------------------------------------------
int getMaxNodeLabelA(std::vector<refNode> &A)
{
  int numNodes = A.size();
  int maxNodeLabel = -1;

  for (int i=0; i<numNodes; i++)
  {
    if (A[i].nodeLabel > maxNodeLabel)
    {
      maxNodeLabel = A[i].nodeLabel;
    }
  }

  return(maxNodeLabel);
}

// ------------------------------------------------------------------
// Function name:  getMaxNodeLabelC
// Description:  This function scans the [C] matrix for the largest
// nodeLabel.
// ------------------------------------------------------------------
int getMaxNodeLabelC(std::vector<connection> &C)
{
  int numEdges = C.size();
  int maxNodeLabel = -1;

  // Scan vertex 1
  for (int i=0; i<numEdges; i++){
    if (C[i].v1 > maxNodeLabel){
      maxNodeLabel = C[i].v1;
    }
  }

  for (int i=0; i<numEdges; i++){
    if (C[i].v2 > maxNodeLabel){
      maxNodeLabel = C[i].v2;
```

```
50      }
51    }
52
53    return(maxNodeLabel);
54  }
55
56
57  // ------------------------------------------------------------------
58  // Function name: floyd
59  // Description:  This function reads in C matrix (from the GNG algorithm)
60  // and creates a 2D vector of vectors of distances betwen all pairs of
61  // nodes using Floyd's algorithm (see A. Tucker, "Applied Combinatorics").
62  // ------------------------------------------------------------------
63  std::vector<std::vector<double> > floyd(std::vector<refNode> &A,
64    std::vector<connection> &C)
65  {
66    int numEdges = C.size();
67    std::vector<double> newVector(0);
68    double huge_distance = 9999999.0; // = inf
69    int v1, v2;
70    double edgeLength;
71    std::vector<std::vector<double> > D(0);
72
73    // Initialize maxNodeLabel to be larger by one so that we can address
74    // nodes in a vector without dealing with the zero-th entry.
75    int maxNodeLabel = getMaxNodeLabelC(C);
76
77    // D will have entries for nodes that do not exist
78    // since the A matrix may have larger nodeLabels than the
79    // maxNodeCnt due to deletions by GNG.
80    for (int i=0; i<=maxNodeLabel; i++)
81    {
82      newVector.clear();
83      for (int j=0; j<=maxNodeLabel; j++)
84      {
85        newVector.push_back(huge_distance);
86      }
87      // Set nodes' distances from themselves to 0.
88      if (i > 0) {newVector[i] = 0;}
89      D.push_back(newVector);
90    }
91
92    // Set distances between adjacenct nodes.
93    for (int i=0; i<numEdges; i++)
94    {
95      v1 = C[i].v1;
96      v2 = C[i].v2;
97      edgeLength = C[i].age;
98
99      D[v1][v2] = (double)edgeLength;
100     D[v2][v1] = (double)edgeLength;
101   }
102
103   double temp = 0;
104   for (int k=1; k<=maxNodeLabel; k++)
105   {
106     for (int i=1; i<=maxNodeLabel; i++)
107     {
108       for (int j=1; j<=maxNodeLabel; j++)
109       {
110         temp = (D[i][k] + D[k][j]);
111
112         if (temp < D[i][j])
113         {
114           D[i][j] = temp;
115         }
116       }
117     }
118   }
119
120
121   // Adjust the distances based on whether the
122   // ancestor of the current action gave us good advice.
123   // Set distances to infinity if bad advice was given.
124   int my_ancestor;
125   int my_nodeLabel;
126   int Arows = A.size();
127   for (int i=0; i<Arows; i++)
128   {
129     my_ancestor = A[i].ancestor;
130     my_nodeLabel = A[i].nodeLabel;
131
132     if ((A[i].reward == -1) && (my_ancestor != my_nodeLabel))
133     {
134       D[my_nodeLabel][my_ancestor] = huge_distance;
135       D[my_ancestor][my_nodeLabel] = huge_distance;
136     }
```

```
137   }
138
139   return(D);
140
141 } // end floyd
142
143
144
145 // ------------------------------------------------------------------
146 // Function name: nodeAdjacency
147 // Description:  This function reads in C matrix (from the GNG algorithm)
148 // and creates a 2D matrix of node adjacencies.  C is understood to be
149 // an undirected graph.  A square matrix is allocated in case the maxNodeLabel
150 // is greater than the number of nodes.
151 // ------------------------------------------------------------------
152 std::vector<std::vector<double> > nodeAdjacency(
153   std::vector<connection> &C)
154 {
155
156   std::vector<std::vector<double> > AdjMat(0);
157
158   // Add one to ignore zero-th entries in vectors.
159   int maxNodeLabel = getMaxNodeLabelC(C);
160
161   int numEdges = C.size();
162   std::vector<double> newVector(0);
163   int vertex1, vertex2;
164
165   // Initialize the Adjacency matrix.
166   AdjMat.clear();
167   for (int i=0; i<=maxNodeLabel; i++)
168   {
169     newVector.clear();
170     for (int j=0; j<=maxNodeLabel; j++)
171     {
172       newVector.push_back(0);
173     }
174     AdjMat.push_back(newVector);
175   }
176
177
178   for (int i=0; i<numEdges; i++)
179   {
180     vertex1 = C[i].v1;
181     vertex2 = C[i].v2;
182
183     // AdjMat is symmetric (Since C is undirected).
184     AdjMat[vertex1][vertex2] = 1.0;
185     AdjMat[vertex2][vertex1] = 1.0;
186   }
187
188   return(AdjMat);
189
190 } // end nodeAdjacency
191
192
193 // ------------------------------------------------------------------
194 // Function name: nodeDegree
195 // Description:  This function computes the k_vector (node degree)
196 // for a C graph matrix.
197 // ------------------------------------------------------------------
198 std::vector<double> nodeDegree(std::vector<connection> &C)
199 {
200   std::vector<double> k_vector(0);
201   int vertex1, vertex2;
202   int maxNodeLabel = getMaxNodeLabelC(C);
203   int numEdges = C.size();
204
205   // Initialize k_vector.
206   for (int i=0; i<=maxNodeLabel; i++)
207   {
208     k_vector.push_back(0);
209   }
210
211   for (int i=0; i<numEdges; i++)
212   {
213     vertex1 = C[i].v1;
214     vertex2 = C[i].v2;
215
216     k_vector[vertex1] += 1.0;
217     k_vector[vertex2] += 1.0;
218   }
219
220   return(k_vector);
221
222 }
223
```

```
// ------------------------------------------------------------------
// Function name:  clumpiness
// Description:  This function computes the clumpiness matrix
// for a network based on the degrees of nodes and the distance
// between them.  The clumpiness, distance and adjacency matrices are
// structured as vectors of vectors.
// ------------------------------------------------------------------
std::vector<std::vector<double> > clumpiness(std::vector<refNode> &A,
  std::vector<connection> &C)
{
  int maxNodeLabel = getMaxNodeLabelC(C);
  std::vector<double> k_vector(0);
  std::vector<double> newVector(0);
  std::vector<std::vector<double> > DistMat(0);
  std::vector<std::vector<double> > ClumpMat(0);

  k_vector = nodeDegree(C);
  DistMat = floyd(A,C);

  // Initialize ClumpMat (2D clumpiness matrix).
  for (int i=0; i<=maxNodeLabel; i++){
    newVector.clear();
    for (int j=0; j<=maxNodeLabel; j++){
      newVector.push_back(0);
    }
    ClumpMat.push_back(newVector);
  }


  for (int i=1; i<=maxNodeLabel; i++) {
    for (int j=1; j<=i; j++) {
      if (i != j) {
        ClumpMat[i][j] = (double)((k_vector[i] * k_vector[j]) / pow(DistMat[i][j],2));
        ClumpMat[j][i] = ClumpMat[i][j];
      }
    }
  }

  return(ClumpMat);

} // clumpiness

// ------------------------------------------------------------------
// Function name:  admittanceMatrix
// Description:  This function computes the admittance (Kirchhoff) matrix
// Av for a GNG network based on the C matrix.
// ------------------------------------------------------------------
std::vector<std::vector<double> > admittanceMatrix(
  std::vector<connection> &C)
{
  int numEdges = C.size();
  int numNodes = getMaxNodeLabelC(C);
  std::vector<std::vector<double> > Av(0);
  std::vector<double> newVector;

  printf("admittanceMatrix 1\n");

  // Initialize Av;
  for (int i=0; i<=numNodes; i++) {
    for (int j=0; j<numNodes; j++) {
      newVector.push_back(0);
    }
    Av.push_back(newVector);
  }


  printf("admittanceMatrix 2, Av size = %d\n", Av.size());

  int v1, v2;
  double admittance = 0.0;
  double age = 0.0;
  for (int i=0; i<numEdges; i++) {
    v1 = C[i].v1;
    v2 = C[i].v2;
    age = (double)C[i].age;
    admittance = (1/(age+1));
    // printf("admittance = %.3f\n",admittance);

    Av[v1][v2] = admittance;
    Av[v2][v1] = admittance;
    printf("i = %d, v1 = %d, v2 = %d, age = %.2f, admittance = %.3f\n",i, v1, v2, age, admittance);
  }

  printf("admittanceMatrix 3\n");

  return(Av);

```

```
311 } // admittanceMatrix
312
313 // ------------------------------------------------------------------
314 // Function name:  laplacian
315 // Description:  This function computes the laplacian matrix
316 // for a GNG network based on the adjacency (degree) and admittance
317 // matrices.
318 // ------------------------------------------------------------------
319 std::vector<std::vector<double> > laplacian(
320   std::vector<std::vector<double> > &K, // Adjacency matrix
321   std::vector<std::vector<double> > &Av) // Admittance matrix
322 {
323   std::vector<std::vector<double> > L(0);
324   std::vector<double> newVector(0);
325
326   int L_size = K.size();
327
328   printf("L_size = %d\n", L_size);
329   printf("K_size = %d\n", K.size());
330   printf("Av_size = %d\n", Av.size());
331
332   // Initialize L
333   for(int i=0; i<L_size; i++){
334     newVector.push_back(0);
335   }
336   for(int i=0; i<L_size; i++){
337     L.push_back(newVector);
338   }
339
340
341   for(int i=0; i<L_size; i++) {
342     for(int j=0; j<L_size; j++) {
343
344       /*
345       printf("i = %d, j = %d, K = %.3f, Av = %.3f.\n",
346         i,j,K[i][j],Av[i][j]);
347       */
348
349       L[i][j] = K[i][j] - Av[i][j];
350     }
351   }
352
353   printf("laplacian 1\n");
354
355   return(L);
356
357 } // laplacian
358
359 // ------------------------------------------------------------------
360 // Function name:  resDist
361 // Description:  This function computes the resistance distance matrix
362 // for a GNG network based on the C matrix.
363 // ------------------------------------------------------------------
364 std::vector<std::vector<double> > resDist(std::vector<connection> &C)
365 {
366   // int numEdges = C.size();
367   int numNodes = getMaxNodeLabelC(C);
368   std::vector<double> k_vector(0);
369
370   printf("resDist 1: numNodes = %d\n", numNodes);
371
372   // Calculate the Admittance (Kirchhoff) matrix Av.
373   std::vector<std::vector<double> > Av(0);
374   Av = admittanceMatrix(C);
375
376   printf("resDist 2: Av size = %d\n", Av.size());
377
378   // Calculate the degree vector (k_vector) and adjacency matrix K.
379   std::vector<std::vector<double> > K(0);
380   k_vector = nodeDegree(C);
381   K = nodeAdjacency(C);
382
383   printf("resDist 3: K size = %d\n", K.size());
384
385   // Calculate the Laplacian of the network
386   std::vector<std::vector<double> > L(0);
387   L = laplacian(K,Av);
388
389   printf("resDist 4: L size = %d\n", L.size());
390
391   // Calculate the auxiliary matrix.
392   std::vector<std::vector<double> > sumMat(0);
393   std::vector<double> newVector(0);
394   for(int i=0; i<=numNodes; i++) {
395     newVector.clear();
396     for(int j=0; j<numNodes; j++) {
397       newVector.push_back(L[i][j] + 1/numNodes);
```

```
    }
    sumMat.push_back(newVector);
  }
  printf("resDist 4a: sumMat size = %d\n", sumMat.size());

  int n = sumMat.size();
  std::vector<std::vector<double> > sumMatInv(0);
  sumMatInv = matrix_inverse(sumMat,n);



  ////////////////////////////////////
  ////////////////////////////////////
  ////////////////////////////////////
  //////////////////////////////////// NOT WORKING - use matlab implementation.
  ////////////////////////////////////
  ////////////////////////////////////
  ////////////////////////////////////

  n = sumMatInv.size();
  printf("resDist 5: sumMatInv size = %d\n",n);

  double resistance = 0;
  std::vector<std::vector<double> > Omega(0);
  printf("resDist5a:\n");
  newVector.clear();
  for (int i=0; i<n; i++){
   newVector.push_back(0);
  }
  for (int i=0; i<n; i++){
    Omega.push_back(newVector);
  }

  n = Omega.size();
  printf("resDist 6: Omega_size = %d\n", n);

  for(int i=0; i<=numNodes; i++){
    for (int j=0; j<=i; j++) {
      // printf("i = %d, j = %d, smi[%d][%d]=%.2f, smi[%d][%d]=%.2f, smi[%d][%d]=%.2f\n", i,j,i,i,sumMatInv[i][i
    ],i,j,sumMatInv[i][j],j,j,sumMatInv[j][j]);
      resistance = sumMatInv[i][i] - 2*sumMatInv[i][j] + sumMatInv[j][j];

      Omega[i][j] = resistance;
      Omega[j][i] = resistance;
    }

    // Prevent a node from having its lowest R-distance to itself.
    Omega[i][i] = 9999999;
  }

  return(Omega);


}
```

A.1.13 gngTrain.cpp

```
#include "kinect_includes.h"
#include "points.h"
#include "utilities.h"
#include "gng.h"
#include "graphs.h"
#include "assertions.h"
#include "gestureLrn.h"
#include "lists.h"

using namespace sensor_msgs;
using namespace ros;
using namespace std;

char A_fname[40] = "A.txt";
char C_fname[40] = "C.txt";
char* baseFileName;
int autoGen = 1;
std::vector<refNode> A(0);
std::vector<refNode> N(0);
std::vector<connection> C(0);
knownGoals autoGoals;
refNode NN;
int observationNum = 0;
int gestureClass = 99;
double err_tol = 0.2;
V_string args;

```

```
28 // Data representation parameters
29 std::vector<descriptor> descriptor_list(0);
30 // char descriptor_fname[40] = "DIs.txt";
31 int featureVecSize = 20;
32 // char descriptor_fname[40] = "HOGs.txt";
33 // int featureVecSize = 8;
34 std::vector<double> vec_in;
35
36 // GNG parameters
37 int lambda = 100;
38 int maxNodeCnt = 100; // Change to 100 for randomized samples.
39 bool done = false;
40 double avgE = 0;
41 std::vector<double> avgE_history(0);
42 int history_limit = 10;
43 int history_size;
44 double err_thold = 0.005*maxNodeCnt;
45 int numNodes;
46 int numEpochs = 0;
47 int top, bottom;
48 int precision = 100;
49
50
51 int main(int argc, char **argv)
52 {
53
54   srand(time(0));
55
56   /*
57   printf("+-----------------------------------------+\n");
58   printf("| Running:  gestureLrnList                |\n");
59   printf("+-----------------------------------------+\n");
60   */
61
62   // Parse the command line for number of epochs to run.
63   if (argc != 2)
64   {
65     printf("Usage:  gngTrain <DI_training_data_file>\n");
66     return(0);
67   }
68   else
69   {
70
71     char* descriptor_fname = argv[1];
72
73
74     // Read in DIs from file.
75     descriptor_list = read_descriptor_list(descriptor_fname, featureVecSize);
76     int numSamples = descriptor_list.size();
77     printf("Read %d DIs from file %s.\n", numSamples, descriptor_fname);
78
79
80     while(done == false)
81     {
82
83       // One epoch
84       for(int v=0; v<numSamples; v++)
85       {
86         vec_in = descriptor_list[v].featureVec;
87         gestureClass = descriptor_list[v].classNum;
88
89         // Apply the representation to the GNG algorithm.
90         NN = gng(A_fname, C_fname, A, C, vec_in, N, lambda, maxNodeCnt);
91         ASSN_duplicateConx(C);
92
93       } // for v
94
95       // How many epochs did it take?
96       numEpochs += 1;
97
98
99       // Store the last history_limit error samples.
100      avgE = calc_avgE(A);
101      history_size = avgE_history.size();
102      if (history_size == history_limit)
103      {
104        for (int i=0; i<history_limit-1; i++)
105        {
106          avgE_history[i] = avgE_history[i+1];
107        }
108        avgE_history[history_limit-1] = avgE;
109
110        top = (int) round(avgE_history[0]*precision);
111        bottom = (int) round(avgE_history[history_limit-1]*precision);
112      }
113      else
114      {
```

```
115         avgE_history.push_back(avgE);
116       }
117
118
119
120       numNodes = A.size();
121
122       printf("%6d epochs, avgE = %f\n", numEpochs, avgE);
123
124
125       // Check for a history of n common
126
127
128
129       if ((top == bottom) && (history_size == history_limit) && (numNodes == maxNodeCnt))
130       {
131         done = true;
132       }
133
134     }  //while
135
136
137
138   // Write [A] and [C].
139   write_A(A_fname, A);
140   write_C(C_fname, C);
141
142   int numEdges = C.size();
143   printf("GNG trained in %d epochs: \n",numEpochs);
144   printf("numNodes = %d\n", numNodes);
145   printf("numEdges = %d\n", numEdges);
146   printf("avgE = %.3f\n", avgE);
147
148   } // else (arg cnt)
149
150
151   return(1);
152
153 } // end main
```

A.1.14 getSkelData.h

```
1 #include "kinect_includes.h"
2
3 class getSkelData {
4
5   public:
6     getSkelData();
7
8     const char* A_fname;
9     const char* C_fname;
10    const char* participantName;
11    std::vector<refNode> A;
12    std::vector<connection> C;
13    std::vector<double> vec_in;
14    int numFrames;
15    int autoGen;
16
17    // Goal configurations for known gestures (autoGen mode only)
18    knownGoals autoGoals;
19
20    // Motion data capture (from Kinect).
21    // void getJointsCallback_1(const body_msgs::Skeletons &skel); // obsolete
22    void getJointsCB(const body_msgs::Skeletons &skel);
23
24    // Get the gesture type from command line input.
25    int getGestureType(char* gestureType, char* baseFileName, char* bagFileName, int obsNum);
26
27    // Node Handles
28    ros::NodeHandle nh1; // Kinect skeleton subscriber
29
30    // Publishers and subscribers
31    ros::Subscriber skel_sub; // Kinect skeleton
32
33    // Skeleton data.
34    rosbag::Bag gesture_bag;
35
36 };
37
38 // Constructor
39 getSkelData::getSkelData()
40 {
41   skel_sub = nh1.subscribe("/skeletons",3,&getSkelData::getJointsCB,this);
42 }
```

```
43
44
45 // ------------------------------------------------------------------
46 // Function name:  getGestureType
47 // Description:  This function queries the user for the type of gesture
48 // that is about to be performed.  For initial testing, one of three
49 // designations is possible:  0=unknown, 1=come, 2=go, 3=stop.
50 // Designatins > "3" are also assigned to "unknown."
51 //
52 // "gestureType" is used to compare the gesture response to known
53 // goals when this program is run in autoGen mode.  It is also used
54 // as a string field in the bag file name to help identify it for
55 // any future testing.
56 // ------------------------------------------------------------------
57 int getSkelData::getGestureType(char* gestureType, char* baseFileName, char* bagFileName, int obsNum)
58 {
59   /*
60   static int unknown = 0;
61   static int come = 1;
62   static int go = 2;
63   static int stop = 3;
64   static int eat = 4;
65   static int read = 5;
66   static int rest = 6;
67   static int get = 7;
68   static int give = 8;
69   static int therapy = 9;
70   */
71   int gestureClassNum = 0;
72   // int retVal = 0;
73
74   printf("Gesture types:\n");
75   printf("0 = unknown\n");
76   printf("1 = come\n");
77   printf("2 = go\n");
78   printf("3 = stop\n");
79   printf("4 = eat\n");
80   printf("5 = read\n");
81   printf("6 = sleep\n");
82   printf("7 = get\n");
83   printf("8 = give\n");
84   printf("9 = therapy\n");
85
86   printf("Enter the type of gesture to be made: ");
87   std::cin >> gestureClassNum;
88   std::cin.ignore(); // Need this to ignore the carriage return.
89
90   // retVal = scanf("%d", &gestureClassNum);
91   // ASSN_gestureClassScan(retVal);
92
93   switch (gestureClassNum)
94   {
95     case 1: sprintf(gestureType,"come"); break;
96     case 2: sprintf(gestureType,"go"); break;
97     case 3: sprintf(gestureType,"stop"); break;
98     case 4: sprintf(gestureType,"eat"); break;
99     case 5: sprintf(gestureType,"read"); break;
100    case 6: sprintf(gestureType,"sleep"); break;
101    case 7: sprintf(gestureType,"get"); break;
102    case 8: sprintf(gestureType,"give"); break;
103    case 9: sprintf(gestureType,"therapy"); break;
104    default: sprintf(gestureType,"unknown"); break;
105  }
106
107 /*  Commented this block out in favor of the switch statement above on 4/17/2013.
108
109  if (gestureClassNum == come)
110  {
111    // printf("sprintf come.\n");
112    sprintf(gestureType,"come");
113  }
114  else if (gestureClassNum == go)
115  {
116    // printf("sprintf go.\n");
117    sprintf(gestureType,"go");
118  }
119  else if (gestureClassNum == stop)
120  {
121    // printf("sprintf stop.\n");
122    sprintf(gestureType,"stop");
123  }
124  else
125  {
126    // printf("sprintf unkown.\n");
127    gestureClassNum = unknown;
128    sprintf(gestureType,"unknown");
129  }
```

```
*/


  // printf("GOT HERE before a char()\n");
  sprintf(baseFileName,"%s_%d_%s", participantName, obsNum, gestureType);
  sprintf(bagFileName, "%s.bag", baseFileName);
  printf("Press <enter> to begin collecting to %s.", bagFileName);

  char temp = 'x';
  while(temp != '\n') {
    std::cin.get(temp);
  }

  return(gestureClassNum);

} // getGestureType

// ------------------------------------------------------------------
// Function name:  getJointsCallback_2
// Description:  This function generates a ROS bag file of skeleton
// motion data consisting of desiredFrames worth of messages.  The
// user is then prompted to continue.  Subsequent bag files are numbered.
// A participant's name is used as the filename root.
// ------------------------------------------------------------------
void getSkelData::getJointsCB(const body_msgs::Skeletons &skel)
{

  // static int come = 1;
  // static int go = 2;
  // static int stop = 3;
  static int numFrames = 0;
  static int observationNum = 0;
  static int desiredFrames = 150;
  static int gestureClass = 0;
  static char bagFileName[40];
  static char baseFileName[40];
  static char gestureType[40];
  refNode NN;

  if (numFrames == 0)
  {
    // Open a new ROS bag file on the first frame.
    observationNum++;
    gestureClass = getGestureType(gestureType, baseFileName, bagFileName, observationNum);
    gesture_bag.open(bagFileName, rosbag::bagmode::Write);

  }
  if (numFrames < desiredFrames) {
    gesture_bag.write("Skeletons", ros::Time::now(), skel);
    numFrames++;
  }
  else
  {
    numFrames = 0;
    gesture_bag.close();
    printf("Bagfile %s saved. \n", bagFileName);

    // Finished.  After <return> go to top and collect a new bag file.
    printf("Press [ENTER] to continue. ");
    getchar();
    printf("\n");
  }

} // end getJointsCB
```

A.1.15 getSkelData.cpp

```
#include "kinect_includes.h"
#include "points.h"
#include "utilities.h"
#include "gng.h"
#include "assertions.h"
#include "getSkelData.h"

using namespace sensor_msgs;

char A_fname[40] = "A.txt";
char C_fname[40] = "C.txt";
char participantName[40] = "p14";

int main(int argc, char **argv)
{
  ros::init(argc, argv, "GSD");
```

```
  getSkelData GSD;

  GSD.A_fname = A_fname;
  GSD.C_fname = C_fname;
  GSD.participantName = participantName;
  GSD.autoGen = 1;

  ros::spin();

} // end main
```

A.1.16 genDI.cpp

```
#include "kinect_includes.h"
#include "points.h"
#include "utilities.h"
#include "gng.h"
#include "assertions.h"
#include "graphs.h"
#include "gestureLrn.h"

using namespace sensor_msgs;
using namespace ros;
using namespace std;

char DI_fname[40] = "DIs.txt";
char* baseFileName;
int gestureClass = 99;
std::vector<double> vec_in;
V_string args;


int main(int argc, char **argv)
{

  /*
  printf("+-----------------------------------------+\n");
  printf("| Running:  genDIs                        |\n");
  printf("+-----------------------------------------+\n");
  */

  if (argc != 2)
  {
    printf("Missing bag file name.\n");
    return(0);
  }
  else
  {

    baseFileName = argv[1];
    gestureClass = getBagType(baseFileName);

    // printf("Reading bag file:  %s.bag\n", baseFileName);

    // Clear the arguments before initializing ROS
    ros::removeROSArgs(argc, argv, args);
    ros::init(argc, argv, "generate_DI");

    // Generate a motion representation.
    vec_in = genRep_dynamicInstants(baseFileName);
    write_to_descriptor_list(DI_fname, vec_in, gestureClass);


    // Apply the representation to the GNG algorithm.

  }

  return(1);

} // end main
```

A.1.17 TurtleControl.h

```
/*
#include <ros/ros.h>
#include <turtlesim/Pose.h>
#include <turtlesim/Velocity.h>
#include <actionlib/server/simple_action_server.h>
#include <turtleControl/moveTurtleAction.h>
#include <cmath>
#include <math.h>
#include <angles/angles.h>
```

```
*/

// This class computes the command_velocities for turtlesim
// to rotate through and angle and to move through a distance.
class moveTurtle
{
public:

  moveTurtle(std::string name) :
    as_(nh_, name),
    action_name_(name)
  {
    //register the goal and feeback callbacks
    as_.registerGoalCallback(boost::bind(&moveTurtle::goalCB, this));
    as_.registerPreemptCallback(boost::bind(&moveTurtle::preemptCB, this));

    //subscribe to the data topic of interest
    sub_ = nh_.subscribe("/turtle1/pose", 1, &moveTurtle::controlCB, this);
    pub_ = nh_.advertise<turtlesim::Velocity>("/turtle1/command_velocity", 1);
  }

  ~moveTurtle(void)
  {
  }

  void goalCB()
  {
    // accept the new goal
    turtleControl::moveTurtleGoal goal = *as_.acceptNewGoal();
    //save the goal as private variables

    x = goal.x;
    y = goal.y;
    theta = goal.theta;

    target_angle_1 = atan2(y,x);
    target_distance = sqrt(pow(x,2) + pow(y,2));
    target_angle_2 = theta;

    result.x = x;
    result.y = y;
    result.theta = theta;
    // turtle_step = 0;
    start_edge = true;

    // Reset helper variables

/*
    edges_ = goal.edges;
    radius_ = goal.radius;

    // reset helper variables
    interior_angle_ = ((edges_-2)*M_PI)/edges_;
    apothem_ = radius_*cos(M_PI/edges_);
    //compute the side length of the polygon
    side_len_ = apothem_ * 2* tan( M_PI/edges_);
    //store the result values
    result_.apothem = apothem_;
    result_.interior_angle = interior_angle_;
    turtle_step =0;
    start_edge_ = true;
*/
  } // goalCB

  void preemptCB()
  {
    ROS_INFO("%s: Preempted", action_name_.c_str());
    // set the action state to preempted
    as_.setPreempted();
  }

  void controlCB(const turtlesim::Pose::ConstPtr& msg)
  {
    // make sure that the action hasn't been canceled
    if (!as_.isActive()) {
      // printf("Shape action not active.  Returning.\n");
      return;
    }

    // scalar values for driving the turtle faster and straighter
    double l_scale = 6.0;
    double a_scale = 6.0;
    double error_tol = 0.00001;

    if (start_edge)
    {
      start_x = msg->x;
```

```
      start_y = msg->y;
      start_angle = msg->theta;
      start_edge = false;
    }

    theta_1_error = angles::normalize_angle_positive(target_angle_1 -
      (msg->theta - start_angle));

    theta_2_error = angles::normalize_angle_positive(target_angle_2 -
      (msg->theta - start_angle));

    distance_error = target_distance -
      fabs(sqrt((start_x- msg->x)*(start_x-msg->x) + (start_y-msg->y)*(start_y-msg->y)));

    // Angle 1
    if (fabs(theta_1_error) > error_tol && distance_error > error_tol)
    {
      printf("Rotating to target angle 1: %8.5f.\n", target_angle_1);
      command.linear = 0;
      command.angular = a_scale * theta_1_error;
    }
    // Linear distance
    else if (distance_error > error_tol)
    {
      printf("Moving distance: %8.5f.\n", target_distance);
      command.linear = l_scale * distance_error;
      command.angular = 0;
    }
    // Angle 2
    else if (fabs(theta_2_error) > error_tol)
    {
      printf("Rotating to target angle 2: %8.5f.\n", target_angle_2);
      command.linear = 0;
      command.angular = a_scale * theta_2_error;
    }
    else
    {
      command.linear = 0;
      command.angular = 0;
      start_edge = true;

      ROS_INFO("%s: Succeeded", action_name_.c_str());
      as_.setSucceeded(result);
    }

    // Publish the velocity command.
    pub_.publish(command);


  }

protected:

  ros::NodeHandle nh_;
  actionlib::SimpleActionServer<turtleControl::moveTurtleAction> as_;
  std::string action_name_;

  double x, y, theta;
  double target_angle_1, target_angle_2, target_distance;

  // scalar values for driving the turtle faster and straighter
  // double l_scale = 6.0;
  // double a_scale = 6.0;
  // double error_tol = 0.00001;

  // double radius_, apothem_, interior_angle_, side_len_;
  double start_x, start_y, start_angle;
  double theta_1_error, theta_2_error, distance_error;
  // int edges_ , turtle_step;
  bool start_edge;
  turtlesim::Velocity command;
  turtleControl::moveTurtleFeedback feedback_;
  turtleControl::moveTurtleResult result;
  ros::Subscriber sub_;
  ros::Publisher pub_;
};

/*


int main(int argc, char** argv)
{
  ros::init(argc, argv, "turtle_motion");

  moveTurtle action(ros::this_node::getName());
  ros::spin();

```

```
  return 0;
}

*/
```

A.1.18 turtleControl_client.h

```
#include <ros/ros.h>
#include <actionlib/client/simple_action_client.h>
#include <actionlib/client/terminal_state.h>
#include <turtleControl/moveTurtleAction.h>

int main (int argc, char **argv)
{
  ros::init(argc, argv, "test_shape");

  // create the action client
  // true causes the client to spin it's own thread
  // actionlib::SimpleActionClient<turtle_actionlib::ShapeAction> ac("turtle_shape", true);
  actionlib::SimpleActionClient<turtleControl::moveTurtleAction> ac("turtle_motion", true);

  ROS_INFO("Waiting for action server to start.");
  // wait for the action server to start
  ac.waitForServer(); //will wait for infinite time

  ROS_INFO("Action server started, sending goal.");
  // send a goal to the action
  turtleControl::moveTurtleGoal goal;


  goal.x = -2.0;
  goal.y = 0;
  goal.theta = M_PI/3;
  ac.sendGoal(goal);

  //wait for the action to return
  bool finished_before_timeout = ac.waitForResult(ros::Duration(40.0));

  if (finished_before_timeout)
  {
    actionlib::SimpleClientGoalState state = ac.getState();
    ROS_INFO("Action finished: %s",state.toString().c_str());
  }
  else
  {
    ROS_INFO("Action did not finish before the time out.");
  }

/*
  goal.x = 3.0;
  goal.y = 3.0;
  goal.theta = 2*M_PI/3;
  ac.sendGoal(goal);

  //wait for the action to return
  finished_before_timeout = ac.waitForResult(ros::Duration(40.0));

  if (finished_before_timeout)
  {
    actionlib::SimpleClientGoalState state = ac.getState();
    ROS_INFO("Action finished: %s",state.toString().c_str());
  }
  else
  {
    ROS_INFO("Action did not finish before the time out.");
  }

*/


  //exit
  return 0;

}
```

A.1.19 turtleControl_server.h

```
#include <ros/ros.h>
#include <turtlesim/Pose.h>
#include <turtlesim/Velocity.h>
#include <actionlib/server/simple_action_server.h>
#include <turtleControl/moveTurtleAction.h>
```

```
6 #include <cmath>
7 #include <math.h>
8 #include <angles/angles.h>
9 #include "turtleControl.h"
10
11
12 int main(int argc, char** argv)
13 {
14   ros::init(argc, argv, "turtle_motion");
15
16   moveTurtle action(ros::this_node::getName());
17   ros::spin();
18
19   return 0;
20 }
```

A.2 Matlab Code

This appendix includes Matlab code to support the research of chapters 2, 3 and 4. Programs specifically related to the experimentation of chapter 2 may be found in Appendix A.2.5. Top level programs related to the experimentation of chapters 3 and 4 include:

- `gestureLrnList.m`: This program emulates `gestureLrnList.cpp`. This program is capable of all methods tested in chapter 3 including those not covered by the C++ implementation (node insertion/deletion, policy freezing for trained nodes, and resistance distance).
- `gl_oneShot.m`: This program implements the use model described in section 4.1.2.
- `gl_kNN.m`: This program emulates `gl_oneShot.m` for the kNN algorithm.

A.2.1 Gesture Recognition Tools

A.2.1.1 `calc_di_ideal.m`

```
% This script generates a collection of seperable points 3D points.
% Each set of three 3d points constitutes a Dynamic Instant (DI).

numGestureTypes = 3;
numDIsPerSample = 5;

% ---------------------------------------------------------------
% Read in DI data
% ---------------------------------------------------------------
% Read in an existing DI list to match its randomness.
fprintf('Reading in DI data\n');
DIdata = dlmread('/home/pyanik/ros_workspace/kinect/bin/DIs_750_ideal.txt');

% Each DI consists of [gestureType],[DInum], [x,y,z] = 4x1

[rows,cols] = size(DIdata);

% k = 0;
% for i=1:rows
%     rowType = DI_Data(i,1);
%     for j=1:numDIsPerSample
%         start = (j*3)+(j-1);
%         stop = (start + 2);
%         DI_array{rowType,j}(end+1,1:5) = [rowType, j, DI_Data(i,start:stop)];
%     end
% end

% ---------------------------------------------------------------
% Generate the randomized ideal DIs
% ---------------------------------------------------------------
outFileName = '/home/pyanik/ros_workspace/kinect/bin/DIs_750_ideal_3vec.txt';

seeds = cell(numGestureTypes, numDIsPerSample);

for i=1:numGestureTypes
    for j=1:numDIsPerSample
        seeds{i,j}(1,1:3) = [2*j, 2*1, 2*i];
    end
end

numSamples = 250;
numDIsPerSample = 5;
DIs = cell(numGestureTypes,numDIsPerSample);
marginVal = 0.3;


for i=1:numGestureTypes
```

```
    for j=1:numDIsPerSample

        m = seeds{i,j}(1,1:3);

        rmin = m(1,1:3) - marginVal;
        rmax = m(1,1:3) + marginVal;


        for k=1:numSamples

            DIs{i,j}(k,1) = i+0.0001;

            for q=1:3
                a = rmin(1,q);
                b = rmax(1,q);

                num = a + (b-a).*rand(1,1);
                DIs{i,j}(k,q+1) = num;
            end

        end
    end
end


% ----------------------------------------------------------------
% Plot DIs
% ----------------------------------------------------------------
fprintf('Plotting Ideal DIs.\n');


for i=1:numGestureTypes
    for j=1:numDIsPerSample;
        switch j
            case 1
                c = 'r.';
            case 2
                c = 'g.';
            case 3
                c = 'b.';
            case 4
                c = 'm.';
            case 5
                c = 'c.';
            otherwise
                fprintf('Bad numPtClouds %d.\n',j);
        end

        plot3(DIs{i,j}(:,2), DIs{i,j}(:,3), DIs{i,j}(:,4),c);
        hold on;


    end
end

hold off;

xlabel('X'); ylabel('Y'); zlabel('Z');
title('Idealized Dynamic Instants for straight trajectories');
legend('DI 1', 'DI 2', 'DI 3', 'DI 4', 'DI 5', 'Location', 'Northeast');


% Reformat DIs to the format expected by gestureLrn.cpp
outArray = zeros(rows,cols);
typePtrs = zeros(numGestureTypes, 1); % Pointers into gen_DI_array


for i=1:rows
    rowType = DIdata(i,1);
    typePtrs(rowType,1) = typePtrs(rowType,1) + 1;
    a = typePtrs(rowType,1);

    outArray(i,1) = rowType;

    for j=1:numDIsPerSample
        start = (j*4)-2;
        stop = (start + 3);
        oneDI = DIs{rowType,j}(a,1:4);
        outArray(i,start:stop) = oneDI;
    end

end
```

```
% Write out the idealized DI file.

dlmwrite(outFileName,outArray,'delimiter',' ','precision', '%d', 'precision', '%12.8f');



fprintf('Printed %d DIs to file %s.  Done.\n', size(outArray,1), outFileName);
fprintf('Done.\n');
```

A.2.1.2 calc_err.m

```
% Function name:  calc_err
%
% This function calculates the average error for a number of runs
% of gestureLrnList over a common number of epochs.  This is for the
% purpose of comparing different scenarios (e.g. neighborhood vs. no
% neighborhood).

function [] = calc_err(realIdeal, hoodRadius, numEpochs, numRuns);

baseFileStr = ['/home/pyanik/ros_workspace/kinect/bin/batch_data/rawData_',realIdeal,'_radius',num2str(hoodRadius
    ),'_run'];

per_run_error = zeros(numRuns,1);
avg_run_error = 0;


for run=1:numRuns

    fileName = [baseFileStr,num2str(run)];
    fileData = dlmread(fileName);

    numGestures = 3;
    come = 1;
    go = 2;
    stop = 3;


    totalSamples = size(fileData,1);
    samplesPerEpoch = totalSamples/numEpochs;

    sampleCounts = zeros(numGestures,1);
    epochAvgs = zeros(numGestures, numEpochs);

    epoch_GNG_AvgE = zeros(numEpochs,1);


    % Find the number of gestures of each type sampled.
    for i=1:totalSamples

        gestType = fileData(i,1);

        sampleCounts(gestType,1) = sampleCounts(gestType,1) + 1;

    end

    sampleCounts = sampleCounts / numEpochs;

    % fprintf('Come = %3d, go = %3d, stop = %3d.\n', ...
    %     sampleCounts(1,1), sampleCounts(2,1), sampleCounts(3,1));


    k = 0;
    for i=1:numEpochs

        oneEpochAvgs = zeros(numGestures, 1);

        avgE = 0;

        for j=1:samplesPerEpoch

            k = k + 1;

            gestType = fileData(k,1);
            error = fileData(k,2);

            % Add up error for each gesture type.
            oneEpochAvgs(gestType,1) = oneEpochAvgs(gestType,1) + error;

            % Add up the GNG cloud error (column 3).
            % avgE = avgE + fileData(k,3);

        end

```

```
        % epoch_GNG_AvgE(i,1) = avgE / samplesPerEpoch;


        for j=1:numGestures

            oneEpochAvgs(j,1) = oneEpochAvgs(j,1)/sampleCounts(j,1);

        end


        epochAvgs(:,i) = oneEpochAvgs;

        per_run_error(run,1) = per_run_error(run,1) + sum(oneEpochAvgs);


    end

    % Average the per_run_error over the number of epochs.
    per_run_error(run,1) = per_run_error(run,1);

    fprintf('Run %3d error = %12.3f\n', run, per_run_error(run,1));


end % runs

avg_run_error = sum(per_run_error)/numRuns;

fprintf('Avg run error for %d runs is %12.3f\n', numRuns, avg_run_error);
fprintf('Done.\n');
```

A.2.1.3 countNodeRwds.m

```
function [hotCnt,warmCnt,coldCnt] = countNodeRwds(A,P)
% ----------------------------------------------------------------------
% Function name:  countNodeRwds
% Author:  Paul Yanik
%
% Description:  This function counts the number of nodes in the GNG A
% matrix having hot/warm/cold rewards and returns the respective count
% values. It may be used to create another factor for adding new GNG nodes
% such as a condition for new_node_needed.
% ----------------------------------------------------------------------

numNodes = size(A,1);

hotCnt = 0;
warmCnt = 0;
coldCnt = 0;


for i = 1:numNodes

    thisRwd = A(i,P.reward);

    switch thisRwd

        case P.warm
            warmCnt = warmCnt + 1;

        case P.hot
            hotCnt = hotCnt + 1;

        case P.cold
            coldCnt = coldCnt + 1;

        otherwise
            fprintf('ERROR - Invalid reward value = %d\n', thisRwd);

    end % switch
end


end % function
```

A.2.1.4 findNearHood.m

```
% ----------------------------------------------------------------------
% Script name:  findNearHood
% Author:  Paul Yanik
%
% Description:  This script selects the nodes within a mean neighborhood
% radius of a given reference node (NN).  It uses the list of all connected
% nodes generated in gng.adjustNeighbors (N).
```

```
% ----------------------------------------------------------------------

numNeighbors = size(N,1);

fvec1 = P.fvec1;
fvec2 = P.fvec2;

% Store the distances of each neighbor from the winner (NN).
distances = zeros(numNeighbors,1);
for i=1:numNeighbors

    distances(i,1) = norm(NN(fvec1:fvec2) - N(i,fvec1:fvec2));

    N(i,P.Q) = norm(N(i,fvec1:fvec2));

end
mean_distance = mean(distances);

% Generate the nearHood list.
for i=1:numNeighbors

    if (distances(i,1) < mean_distance)

        nearN(end+1,:) = N(i,:);

    end

end
```

A.2.1.5 genAction_xyt.m

```
% ----------------------------------------------------------------------
% Script name: genAction_xyt
% Author:  Paul Yanik
%
% Description:
% This script generates a [robotic] action on (x,y,t) space.
% Several methods for generating this action are currently populated.
% Others may be added by augmenting the options for the 'hoodRadius'
% variable.  The GNG network cloud is used to determine the node
% with the best action to emulate based on expected reward and distance
% from a winning refrence node (NN).
% ----------------------------------------------------------------------

% compareANN(A,NN,P,'GenAct1');

reward = P.reward;
Q = P.Q;
act1 = P.act1;
act2 = P.act2;
last1 = P.last1;
last2 = P.last2;
nodeLabel = P.nodeLabel;
ancestor = P.ancestor;
warm = P.warm;
hot = P.hot;
cold = P.cold;
step_size = P.step_size;
angle_delta = P.angle_delta;

numNodes = max(A(:,nodeLabel),1);
Arows = size(A,1);


% Add the winner node (NN) to its neighborhood list (N).
NN_Q = norm(NN(1,act1:act2));



% Choose neighborhood radius and assign neighbors within it to nearN.
nearN = [];
if (hoodRadius == 7) % kNN

    % The neighborhood is already defined.
    nearN = kNN_hood;

    % Set NN to be the first in the kNN_hood.
    % NN = kNN_hood(1,:);


elseif (hoodRadius == 6) % Resistance distance

    Omega = resDist(C, P.v1, P.v2, P.edgeLen_col);

    thisNode = NN(1,nodeLabel);
```

```
    [val, minNodeLabel] = min(Omega(thisNode,:));


    % Assign nearN to be the node with minNodeLabel.
    for z=1:Arows

        % A(z,Q) = norm(A(z,act1:act2));

        if (A(z,nodeLabel) == minNodeLabel)
            nearN(1,:) = A(z,:);

            % fprintf('Got One.\n');

            % Recalculate Q values
            % A(z,Q) = norm(A(z,act1:act2));

        end
    end

    % Add NN to the near neighborhood if not already there.
    if (size(nearN,1) == 0)
        nearN = NN;
    elseif (nearN(1,nodeLabel) ~= NN(1,nodeLabel))
        nearN(end+1,:) = NN;
        % fprintf('GOT ONE\n');
    end

elseif (hoodRadius == 5) % clumpiness

    clumpMat = clumpiness(C,P,A);

    % Choose the node with largest clumpiness coefficient for thisNode.
    thisNode = NN(1,nodeLabel);
    [val,max_nodeLabel] = max(clumpMat(thisNode,:));


    % Assign nearN to be the node with max_nodeLabel.
    for z=1:numNodes

        if ((A(z,nodeLabel) == max_nodeLabel))
            nearN(1,:) = A(z,:);
        end
    end

    % Add NN to the near neighborhood if not already there.
    if (size(nearN,1) == 0)
        nearN = NN;
    elseif (nearN(1,nodeLabel) ~= NN(1,nodeLabel))
        nearN(end+1,:) = NN;
    end



elseif (hoodRadius == 4) % Floyd

    thisNode = NN(1,nodeLabel);

    D = floyd(C,P,A);

    % Disallow a node choosing itself as nearest neighbor.
    for i=1:size(D,1);
        D(i,i) = inf;
    end

    % Search all nodes with reward = 1.
    % Choose the nearest one that is longer than the current Q.

    % Rule out nodes with reward = -1 by making their distance inf in D.
    for z=1:Arows
        if ((A(z,reward) == -1))

            D_index = A(z,nodeLabel);

            % Rule out nodes with reward = -1.
            D(D_index,thisNode) = inf;
            D(thisNode,D_index) = inf;
        end
    end

    % Rule out nodes with shorter Q values shorter than NN.
    for z=1:Arows
        if (A(z,reward) == 1)

            % Recalculate Q.
            A(z,Q) = norm(A(z,act1:act2));

```

```
142             if (A(z,Q) <= NN_Q)
143
144                 D_index = A(z,nodeLabel);
145
146                 D(D_index,thisNode) = inf;
147                 D(thisNode,D_index) = inf;
148
149             end
150         end
151     end
152
153
154     [val,min_nodeLabel] = min(D(thisNode,:));
155
156     % Assign nearN to be the node with min_nodeLabel (i.e. the closest).
157     for z=1:Arows
158
159         if (A(z,nodeLabel) == min_nodeLabel)
160
161             nearN = A(z,:);
162             break;
163
164         end
165     end
166
167     % Add NN to the near neighborhood if not already there.
168     if (size(nearN,1) == 0)
169         nearN = NN;
170     elseif (nearN(1,nodeLabel) ~= NN(1,nodeLabel))
171         nearN(end+1,:) = NN;
172     end
173
174
175
176 elseif (hoodRadius == 3)
177
178     % Use all neighbors.
179     nearN = N;
180     nearN(end+1,:) = NN;
181
182 elseif (hoodRadius == 2)
183
184     % Use neighbors within a lesser radius.
185     nearN(end+1,:) = NN;
186     findNearHood;
187     % print_N(nearN,P);
188
189 elseif (hoodRadius == 1)
190
191     % Use only the winner itself (NN).
192     nearN(end+1,:) = NN;
193     if (size(nearN,1) > 1)
194         fprintf('BADLY SIZED neighborhood!!!!!\n');
195     end
196
197 else fprintf('Bad scenario:  hoodRadius = %d.\n', hoodRadius);
198 end
199
200
201 % t5 = tic;
202
203 if (NN(1,reward) == hot)
204
205     % Do nothing.
206     % fprintf('HOT=%3d ', NN(1,nodeLabel));
207
208 else ...
209
210     % Find the near neighborhood member with max Q and
211     % reward = 1 (if it exists).
212     numNbrs = size(nearN,1);
213
214     maxQ = -99;
215     maxQnode = -99;
216     foundOne = 0;
217     x = -99;
218     y = -99;
219     theta = -99;
220     this_reward = -99;
221
222
223     for i=1:numNbrs
224
225         if (nearN(i,reward) == cold)
226             action = nearN(i,last1:last2);
227         else action = nearN(i,act1:act2);
228         end
```

```
229
230         this_Q = norm(action);
231
232         if (this_Q > maxQ)
233
234            maxQ = this_Q;
235
236            x = action(1);
237            y = action(2);
238            theta = action(3);
239            this_reward = nearN(i,reward);
240
241            maxQNode = i;
242
243         end
244
245     end % for numNbrs
246
247
248     [t,p,r] = cart2sph(x,y,theta);
249
250
251     % Lengthen action vector if needed.
252     % if ((this_reward == warm) || (r == 0))
253     if (this_reward == warm) % Maintain current angle.
254
255         r = r + step_size;
256
257     end
258
259     if (this_reward == cold)
260
261         if (r == 0)
262             % Cast about until we find a warmer direction.
263             p = rand_in_range(-pi,pi);
264             t = rand_in_range(-pi,pi);
265             r = r + step_size;
266         else ...
267
268             % Adjust the action vector by a small angular correction.
269             p = p + rand_in_range(-angle_delta/r, angle_delta/r);
270             t = t + rand_in_range(-angle_delta/r, angle_delta/r);
271         end
272
273     end
274
275     [x,y,theta] = sph2cart(t,p,r);
276
277     % Apply the action/last/ancestor/reward of nearN(maxQnode) to NN.
278     switch(this_reward)
279
280         case hot
281
282             NN(1,last1:last2) = nearN(maxQNode,act1:act2);
283             NN(1,act1:act2) = nearN(maxQNode,act1:act2);
284             NN(1,ancestor) = nearN(maxQNode,nodeLabel);
285
286             % Do not assign reward.  The neighbor from which the
287             % reward is taken may be hot, but reward by the user may
288             % end up being warm or cold.
289             % NN(1,reward) = nearN(maxQNode,reward);
290
291         case warm
292
293             % Set last to the current action.
294             NN(1,last1:last2) = nearN(maxQNode,act1:act2);
295             NN(1,act1:act2) = [x,y,theta];
296             NN(1,ancestor) = nearN(maxQNode,nodeLabel);
297
298         case cold
299
300             % No change to 'last'
301             NN(1,act1:act2) = [x,y,theta];
302             NN(1,ancestor) = nearN(maxQNode,nodeLabel);
303
304         otherwise % Do nothing.
305
306     end % switch
307
308
309 end
310
311 % t5e = toc(t5);
312
313
314 NN(1,Q) = norm(NN(1,act1:act2));
315
```

```
% compareANN(A,NN,P,'GenAct2');
```

A.2.1.6 genGaussDIs.m

```
function [] = genGaussDIs(N,DIs_outFile)
% ----------------------------------------------------------------------
% This function generates a gaussian distribution of dynamic instants (DI)
% based on a set of input training gestures (1 sample per gesture).
% The output is intended to be similar to an ideal data set with nominal
% variation among DIs.  Note that this assumes the covariance matrix for
% each canddiate gesture is the identity matrix.
%
% The input training data is assumed to be if the form:
% [class, vector(1:vecSize)]
%
% The computation of the unbiased covariance matrix estimate is taken
% from Pattern Recognition (Schalkoff,1992), p. 62.  The matlab cov function
% emulates this computation.
%
% DIs are written to the fileName specified by the user as "DIs_outFile".
% ----------------------------------------------------------------------

% Load the input file.
inFile_data = load('/home/pyanik/ros_workspace/kinect/bin/DIs_750_real.txt', 'ascii');


% Find the mean DI for each type among the input data.
d = size(inFile_data,2) - 1;
types2find = unique(inFile_data(:,1))';
numSamples = size(inFile_data,1);
numTypes = size(types2find,2);
meanDIs = zeros(numTypes,d+1);

k = 0;
for i = types2find

    thisType = i;

    % Store the gesture type in column 1.
    k = k + 1;
    meanDIs(k,1) = thisType;

    % Extract the samples of the current type
    samples = [];
    for j = 1:numSamples
        if (inFile_data(j,1) == thisType)
            samples(end+1,1:d) = inFile_data(j,2:end);
        end
    end

    for j = 1:d
        meanDIs(k,1+j) = mean(samples(:,j));
    end

end

% Set up a matrix for the output points.
DIs_gauss = zeros(N*numTypes,d+1);

k = 0;
b = 0;
for i = types2find

    thisType = i;

    % The data (1 sample) is the mean for that gesture type.
    b = b + 1;
    u = meanDIs(b,2:end);

    % Assume a spherical point cloud for the data by making covariance
    % matrix equal to the identity matrix.
    covMat = eye(d)*(0.001*max(max(inFile_data(:,2:end))));

    % Generate N Gaussian DIs.
    generated_vectors = mgd(N,d,u,covMat);

    % Store the generated vectors.
    for m = 1:N
        k = k + 1;
        temp(k,2:d+1) = generated_vectors(m,:);
        temp(k,1) = int32(thisType);
    end

end

```

```


% Randomize the generated vectors.
k = size(DIs_gauss,1);
for i = 1:k

    foundOne = 0;

    % Generate random indices until an unused
    % location in DIs_gauss is found.
    while(foundOne == 0)

        g = randi(k,[1,1]);

        if (DIs_gauss(g,1) == 0)
            DIs_gauss(g,:) = temp(i,:);
            foundOne = 1;
        end
    end
end


% Write the output file.
save(DIs_outFile,'DIs_gauss','-ascii');

% % Plot the data (for a test).
% plot(temp(1:N,2),temp(1:N,3),'r.', ...
%     temp(N+1:2*N,2),temp(N+1:2*N,3),'b.', ...
%     temp(2*N+1:N*3,2),temp(2*N+1:3*N,3),'g.');




end % function
```

A.2.1.7 gestureLrnList.m

```
function[] = gestureLrnList(descr_file, numEpochs, hoodRadius, kNN)
% ---------------------------------------------------------------------
% Function name:  gestureLrnList
% Author:  Paul Yanik
%
% Description:
% This function emulates gestureLrnList.cpp.  It reads in a descriptor list
% (of DIs) from a file and applies them to the GNG algorithm in series.
% One pass throught the input data constitutes an epoch.  The 'hoodRadius'
% variable denotes the number/types of neighbors used to accelerate
% learning.  The meanings of values for hoodRadius can be found in
% genAction_xyt.m.
% ---------------------------------------------------------------------

params;
read_A;
read_C;


new_node_needed = 0;
neighbors_used = 0;
neighbors_used_successfully = 0;
oneShot = 0;



% Read in DIs from file.
[featureVecs, classNums, numSamples] = read_descriptor_list(descr_file);

% Results array for one run of numEpochs.
% Format of results:  [classNum, Err].
results_array = zeros(numSamples*numEpochs, 2);


% Run numEpochs
for epoch = 1:numEpochs

    fprintf('Epoch = %3d, nodes = %3d\n', epoch, size(A,1));

    for sample = 1:numSamples

        vec_in = featureVecs(sample,:);
        gestureClass = classNums(sample,1);


        gng;

        genAction_xyt;
```

```
49
50          getResponse_warmerColder;
51
52
53          if ((hoodRadius == 5) || (hoodRadius == 4))
54              fprintf('Ep=%2d, Smpl=%3d, n=%3d\n',epoch,sample,size(A,1));
55          end
56
57
58
59
60      end
61
62  end
63
64
65  write_results;
66  write_A;
67  write_C;
```

A.2.1.8 getResponse_warmerColder.m

```
1  % ----------------------------------------------------------------------
2  % Script name:  getResponse_warmerColder
3  % Author:  Paul Yanik
4  %
5  % Description:
6  % This script emulates getResponseFeedback_warmerColder.cpp
7  % It accepts a NN.action and compares it to a known goal (as would a human
8  % user).  If the action moves closer to action than the last action then
9  % a reward of +1 is assigned.  Otherwise a reward of -1 is assigned.  If
10 % the goal has been achieved, a reward of 0 is assigned.
11 %
12 % Determine the desired goal for the current classNum
13 % Format of knownGoals:  [x, y, t] for [come; go; stop ...] respectively.
14 % ----------------------------------------------------------------------
15 goal = knownGoals(gestureClass,:);
16
17 % compareANN(A,NN,P,'GetResp1');
18
19
20 action = NN(1,P.act1:P.act2);
21 last = NN(1,P.last1:P.last2);
22
23 mag_dist2goal = norm(goal - action);
24 mag_last2goal = norm(goal - last);
25
26
27 fb = -99;
28 if (mag_dist2goal < P.err_tol)
29
30     % Goal achieved.
31     fb = P.hot;
32
33 elseif (mag_dist2goal < mag_last2goal)
34
35     % Warmer.
36     fb = P.warm;
37
38 elseif (mag_dist2goal >= mag_last2goal)
39
40     % Colder
41     fb = P.cold;
42
43 else  ...
44
45     fprintf('---------- FEEDBACK ERROR ----------\n');
46
47 end
48
49
50
51 if (kNN ~= 0)
52
53     % Need to alter this according to gl_kNN (training data) or
54     % gl_kNN2 (history buffer).
55
56     NN(1,P.reward) = fb;
57
58     if (kNN_buff ~= 0) % Using a kNN buffer.
59
60         % Comment these lines out for gl_kNN
61         NN(1,P.fvec1:P.fvec2) = vec_in;
62         NN(1,P.nodeLabel) = max(A(:,P.nodeLabel)) + 1;
63         A(end+1,:) = NN;
```

```
    else ... % Using a fixed number of training data points.

        % Put the updated node NN back into A
        Arows = size(A,1);
        for i=1:Arows

            if (A(i,P.nodeLabel) == NN(1,P.nodeLabel))

                A(i,:) = NN(1,:);

                break;
            end
        end

    end

else ...

    % This section of code is for the scenario where a node is being
    % fully trained before any other vectors are considered.

    % Watch for cases where two samples are in the same node's receptive
    % field, but represent different gestures.  If this happens, ignore that
    % sample going forward.  In the future, I may add a GNG node when this
    % happens.

    if ( ((fb == P.cold) || (fb == P.warm)) && (NN(1,P.reward) == P.hot))

%     % If NN is using a response from its neighbor when this happens, set
%     % the length of the edge between the two nodes to a large value.
%     if (NN(1,P.nodeLabel) ~= NN(1,P.ancestor))
%
%         thisEdge = [NN(1,P.nodeLabel), NN(1,P.ancestor)];
%
%         [found, index] = edgeExists(C, P.v1, P.v2, thisEdge);
%
%         if (found == 1)
%
%             C(index, P.edgeLen_col) = 999;
%
%             fprintf('Setting length = 999\n');
%
%         else
%
%             ignore(sample,1) = 1;
%
%             fprintf('Ignoring\n');
%
%         end
%
%     else
%
%         % Ignore this sample for now.  This may not be the right thing to
%         % do in the long run.  We may need to add a new node to the GNG
%         % cloud since receptive fields may be large.
%
%         ignore(sample,1) = 1;
%
%     end

%         % Determine if this node site is a good candidate
%         % for insertion of a new node.
%
%         % Find the node adjacency matrix
%         Av = nodeAdjacency(C,P);
%         % Find the nodeDegree matrix (number of connections)
%         [k, K_matrix] = nodeDegree(Av);
%         thisNode = NN(1,P.nodeLabel);
%         % A good insertion site has node degree less than average.
%         goodInsertionSite = (k(thisNode,1)/mean(k) < 1);


        if (oneShot == 1)

            ignore(sample,1) = 1;

            % Inflate the error at this node so that a new node
            % is more likely to be added here.

            maxE = max(A(:,P.E));
            NN(1,P.E) = maxE + 1;

            % Artificially age the node with oldest connections if
            % numNodes > maxNodeCnt so that it will be most likely
            % to be deleted.
            ageColdNode;
```

```
151
152         end
153
154
155     else
156
157
158        NN(1,P.reward) = fb;
159
160     end
161
162     % Put the updated node NN back into A
163     Arows = size(A,1);
164     for i=1:Arows
165
166         if (A(i,P.nodeLabel) == NN(1,P.nodeLabel))
167
168             A(i,:) = NN(1,:);
169
170             break;
171         end
172     end
173
174 end
175
176
177
178 % Put results in the results matrix.
179 index = ((epoch-1)*numSamples)+sample;
180 results_array(index, 1:3) = [gestureClass, epoch, mag_dist2goal];
181
182 % % Count the number of non-neighbor successful responses.
183 % if (NN(1,P.ancestor) ~= NN(1,P.nodeLabel))
184 %
185 %     neighbors_used = neighbors_used + 1;
186 %
187 %     NN_reward = NN(1,P.reward);
188 %
189 %     if ((NN_reward == 1) || (NN_reward == 0))
190 %         neighbors_used_successfully = neighbors_used_successfully + 1;
191 %     end
192 %
193 % end
```

A.2.1.9 gl_oneShot.m

```
1 function[] = gl_oneShot(descr_file, numEpochs, hoodRadius, kNN)
2 % ----------------------------------------------------------------------
3 % Function name:  gl_oneShot
4 % Author:  Paul Yanik
5 %
6 % Description:
7 % This function emulates gestureLrnList.cpp except that a single gesture
8 % sample is allowed to receive feedback until it is fully trained.
9 %
10 % kNN is a true/false (1/0) value that turns on a block of code in
11 % getResponse_warmerColder.m
12 % ----------------------------------------------------------------------
13
14 tic
15
16 % descr_file = 'DIs_450_real_tst.txt';
17 % numEpochs = 250;
18 % hoodRadius = 1;
19
20 % This file contains runtime parameters for gestureLrn.
21 params;
22
23
24
25 neighbors_used = 0;
26 neighbors_used_successfully = 0;
27 hot_nodes = 0;
28 oneShot = 1;
29 max_training_iterations = 1000;
30
31 % Read in the trained A and C matrices.
32 read_A;
33 read_C;
34 compareAC(A,C,P,'gl_oneShot');
35
36 % Read in DIs from file.
37 [featureVecs, classNums, numSamples] = read_descriptor_list(descr_file);
38
39
```

```
40 % Results array for one run of numEpochs.
41 % Format of results:  [classNum, Err].
42 results_array = zeros(numSamples*numEpochs, 2);
43
44 % Create an array of samples to ignore.
45 ignore = zeros(numSamples,1);
46
47 % Create an indicator for when new nodes should be added.
48 new_node_needed = 0;
49
50 % Record the number of times neighbors were used and
51 % the number of times they were used with improvement.
52 neighbors_used = 0;
53 neighbors_used_successfully = 0;
54 hot_nodes = 0;
55
56
57 % Run numEpochs
58 total_iterations = 0;
59 for epoch = 1:numEpochs
60
61
62     fprintf('Epoch = %d \n', epoch);
63     ignore = zeros(numSamples,1);
64     numIgnored = 0;
65
66
67     for sample = 1:numSamples
68
69         vec_in = featureVecs(sample,:);
70         gestureClass = classNums(sample,1);
71
72         % Add a new node to the GNG cloud if a sample was ignored.
73         % Look for change:
74         [hotCnt,warmCnt,coldCnt] = countNodeRwds(A,P);
75
76         % new_node_needed = (((sum(ignore(:,1)) > numIgnored) && ...
77         %    (hoodRadius ~= 6)) || (coldCnt == 0));
78
79         new_node_needed = (((sum(ignore(:,1)) > numIgnored)) || ...
80             (coldCnt == 0));
81
82
83         numIgnored = sum(ignore(:,1));
84         % gng;
85
86         % fprintf('new_node_needed = %d\n',new_node_needed);
87
88
89         fb = -99;
90         sample_iterations = 0;
91         % sample_error = 0;
92         while ((fb ~= 0) && (ignore(sample,1) == 0))
93
94             gng;
95             new_node_needed = 0;
96
97             genAction_xyt;
98
99             getResponse_warmerColder;
100
101            sample_iterations = sample_iterations + 1;
102            if (sample_iterations > max_training_iterations)
103                ignore(sample,1) = 1;
104            end
105
106            % Floyd and clumpiness - very slow (show progress)
107            if ((hoodRadius == 5) || (hoodRadius == 4))
108                fprintf('sample_iterations = %4d\n',sample_iterations);
109            end
110
111
112        end
113
114
115
116
117        if (sample_iterations > 0)
118            sample_iterations = sample_iterations - 1;
119        end
120
121        % Put results in the results matrix.
122        index = ((epoch-1)*numSamples)+sample;
123        results_array(index, 1:2) = [gestureClass, mag_dist2goal];
124
125        total_iterations = total_iterations + sample_iterations;
126
```

```
        fprintf('ep=%d, smpl = %d, itns= %d, tot= %d, nodes= %d\n', ...
            epoch,sample,sample_iterations,total_iterations,size(A,1));

        % compareAC(A,C,P,'Epochs2');

    end

    fprintf(' Ignrd = %d\n',sum(ignore));




end


ignoreCnt = sum(ignore);
avg_iterations_per_sample = total_iterations / (numSamples-ignoreCnt);



% Count the number of fully trained nodes.
Arows = size(A,1);
for z=1:Arows
    if (A(z,P.reward) == 0)
        hot_nodes = hot_nodes + 1;
    end
end


fprintf('\n');
fprintf('SUMMARY RESULTS:\n');
fprintf('Scenario         = %8d\n', hoodRadius);
fprintf('Total nodes      = %8d.\n', Arows);
fprintf('Trained nodes    = %8d.\n', hot_nodes);
fprintf('Total iterations = %8d.\n', total_iterations);
fprintf('Samples ignored  = %8d.\n', ignoreCnt);
fprintf('Average          = %8.2f\n', avg_iterations_per_sample);
fprintf('Neighbors used   = %8.2f\n', neighbors_used);
fprintf('Successful Nbrs  = %8.2f\n', neighbors_used_successfully);




write_results;
write_A;
write_C;

% fprintf('Neighbors used = %d.\n', neighbors_used);

toc;
```

A.2.1.10 gngTrain.m

```



function [] = gngTrain(descr_file, history, precision)
% ---------------------------------------------------------------------------
% This function trains a Growing Neural Gas cloud based on a collection of
% input descriptors.  The output is a C.txt and A.txt file that contain the
% descriptor fields associated with gesture recognition (params.m).
% ---------------------------------------------------------------------------


% Read in DIs from file.
[featureVecs, classNums, numSamples] = read_descriptor_list(descr_file);

done = 0;
numEpochs = 0;

% Create a list of average errors (initially large).
avgE_history(1:history+1,1) = 9999;


while(done == 0)

   for sample=1:numSamples

       vec_in = featureVecs(sample,:);
       gestureClass = classNums(sample,1);

       gng;
   end

   % Count the number of training epochs.
   numEpochs = numEpochs + 1;
```

```
33
34     % Compute the average error for the GNG cloud.
35     avgE = mean(A(:,P.E));
36     avgE_history(history+1,1) = avgE;
37     avgE_history(1:history,1) = avgE_history(2:history+1,1);
38
39
40     spread = avgE_history(1,1) - avgE_history(history,1);
41
42     if (abs(spread) < precision)
43
44         done = 1;
45
46     end
47
48     fprintf('Epoch = %3d, avgE = %8.3f\n', ...
49         numEpochs, avgE_history(history+1,1));
50
51
52 end % while
53
54
55 write_A;
56 write_C;
57
58 fprintf('GNG is trained in %3d epochs:  nodes = %3d, avgE = %8.3f.\n', ...
59     numEpochs, numNodes, avgE);
```

A.2.1.11 params.m

```
1 % ----------------------------------------------------------------------
2 % Script name:  params
3 % Author:  Paul Yanik
4 %
5 % Description:
6 % This parameter structure contains static runtime parameters for
7 % gestureLrnList (or similar) calling functions.  The P data structure
8 % passes numerous useful parameters into the functions.
9 % ----------------------------------------------------------------------
10
11 numFeatures = 20;
12
13 kNN = 0;
14
15 if (kNN == 0)
16     num_initial_GNG_nodes = 2;
17 else num_initial_GNG_nodes = 1;
18 end
19
20
21 dataDir = '/home/pyanik/ros_workspace/kinect/bin/';
22 workDir = '/home/pyanik/ros_workspace/kinect/bin/';
23 A_file = [dataDir,'A.txt']; % Use dataDir for C++ generated A matrix.
24 C_file = [dataDir,'C.txt']; % Use dataDir for C++ generated C matrix.
25 results_file = [workDir,'results'];
26
27
28 % Use the gestureClassNum as a row index into goals matrix.
29 knownGoals = [
30     3.95, 3.95, pi/4; % Come
31     3.95, -3.95, 7*pi/4; % Go
32     -3.95, -3.95, 5*pi/4; % Stop
33     -3.95, 3.95, 3*pi/4; % Eat
34     3.95, 0, 0; % Read
35     0, 3.95, 2*pi/4; % Sleep
36     -3.95, 0, 4*pi/4; % Get
37     0, -3.95, 6*pi/4; % Give
38     3.95, 1.98, 1*pi/8]; % Therapy
39
40
41 % featureVec column start/end subscripts.
42 fvec1 = 14;
43 fvec2 = fvec1+numFeatures-1;
44
45 P = struct( ...
46     ... % C matrix column subscripts
47     'v1', 1, ...
48     'v2', 2, ...
49     'age', 3, ...
50     'len', 4, ...
51     'C_cols', 4, ...
52     'edgeLen_col', 3, ... % This will point either to 'age' or 'len'
53     ... % A matrix column subscripts
54     'numObs', 1, ...
55     'nodeLabel', 2, ...
```

```
    'numConx', 3, ...
    'reward', 4, ...
    'ancestor', 5, ...
    'Q', 6, ...
    'E', 7, ...
    'act1', 8, ...
    'act2', 10, ...
    'last1', 11, ...
    'last2', 13, ...
    'fvec1', fvec1, ...
    'fvec2', fvec2, ...
    'A_cols', fvec2, ...
    ... % GNG runtime parameters
    'ep_w', 0.05, ...
    'ep_n', 0.0006, ... % Fritzke uses 0.0006
    'ageMax', 300, ... % Use a high number for resistance distance.
    'alpha', 0.5, ...
    'beta', 0.0005, ... % Fritzke uses 0.0005
    'lambda', 100, ...
    'maxNodeCnt', 100, ...
    ... % Action parameters
    'come', 1, ...
    'go', 2, ...
    'stop', 3, ...
    'eat', 4, ...
    'read', 5, ...
    'sleep', 6, ...
    'get', 7, ...
    'give', 8, ...
    'therapy', 9, ...
    'hot', 0, ...
    'warm', 1, ...
    'cold', -1, ...
    'step_size', 0.1, ...
    'angle_delta', pi/18, ...
    'err_tol', 0.1 ...
    );
```

A.2.1.12 plot_A.m

```
% ------------------------------------------------------------------------
% Script name:  plot_A
% Author:  Paul Yanik
% Description:  This script reads in the A and C matrices for a GNG cloud
% and characterizes each node for its reward characteristics and the total
% ages of its connecting edges.
% ------------------------------------------------------------------------

params;

A = dlmread(A_file);
C = dlmread(C_file);

numNodes = size(A,1);
numEdges = size(C,1);

[Arows, Acols] = size(A);
[Crows, Ccols] = size(C);

% Characterize nodes w.r.t. connection ages.
results = zeros(numNodes,4); % [nodeLabel, rwd, numCnx, totAge]
numHot = 0;
numWarm = 0;
numCold = 0;
other = 0;
numObs = A(1,P.numObs);
for i=1:numNodes

    % Collect node/edge data
    nodeLabel = A(i,P.nodeLabel);
    totAge = 0;
    numCnx = 0;
    rwd = A(i,P.reward);
    for j=1:numEdges

        v1 = C(j,P.v1);
        v2 = C(j,P.v2);

        if (v1 == nodeLabel || v2 == nodeLabel)
            totAge = totAge + C(j,P.age);
            numCnx = numCnx + 1;
        end

    end
    results(i,1:4) = [nodeLabel,rwd,numCnx,totAge];
```

```
    % Count reward frequency
    switch rwd
        case P.hot
            numHot = numHot + 1;
        case P.warm
            numWarm = numWarm + 1;
        case P.cold
            numCold = numCold + 1;
            % coldList(end+1,1) = A(i,P.nodeLabel);
        otherwise
            other = other + 1;
    end

end

fprintf('\n');
fprintf('%d Obs, %d Nodes, %d Edges:  \n', numObs, numNodes,numEdges);
fprintf('%3d hot\n', numHot);
fprintf('%3d warm\n', numWarm);
fprintf('%3d cold\n', numCold);
fprintf('%3d other\n', other);
fprintf('\n');

% fprintf('Node stats:  \n');
%
% for i=1:numNodes
%
%     nodeLabel = results(i,1);
%     rwd = results(i,2);
%     numCnx = results(i,3);
%     totAge = results(i,4);
%
%     fprintf('NodeLabel = %3d, rwd = %2d, numCnx = %2d, totAge = %4d\n', ...
%         nodeLabel, rwd, numCnx, totAge);
%
%
%
% end


fprintf('\nDone.\n');
```

A.2.1.13 plot_di.m

```
function [] = plot_di(mode, gesture_type, DI_filename, participantID);
% ----------------------------------------------------------------------
% This function plots Dynamic instants for gesture data.
% Modes:
% 0: plot all DIs of the specified gesture_type.
% n: plot DI #n (n = 1...5) for all gesture types.
% ----------------------------------------------------------------------

posData = dlmread('/home/pyanik/ros_workspace/kinect/bin/POS.txt');

DI_path = ['/home/pyanik/ros_workspace/kinect/bin/',DI_filename]
DI_Data = dlmread(DI_path);

numGestureTypes = 9;
numDIsPerSample = 5;

% Possible gesture_types:
come = 1;
go = 2;
stop = 3;
eat = 4;
read = 5;
sleep = 6;
get = 7;
give = 8;
therapy = 9;

types = {'COME','GO','STOP','EAT','READ','SLEEP','GET','GIVE','THERAPY'};

% ---------------------------------------------------------------
% Read in POS data
% ---------------------------------------------------------------
fprintf('Reading POS data.\n');

[rows,cols] = size(posData);
numCells = max(posData(:,1));

% Store samples in a cell array.
samples = cell(numCells,1);

```

```
% Each sample consists of:
% [sampleNum],[dataType],[frameNum],[x, y, z]

% 1 3 10   0.00799585   0.99651375   0.22241513

for i=1:rows
   cellNum = posData(i,1);
   cellRow = posData(i,3);

   samples{cellNum}(cellRow,1:5) = posData(i,2:6);
end

% ---------------------------------------------------------------
% Read in DI data
% ---------------------------------------------------------------

DI_array = cell(numGestureTypes, numDIsPerSample);

% Each DI consists of [gestureType],[DInum], [x,y,z] = 5x1

fprintf('Reading in DI data\n');
[rows,cols] = size(DI_Data)
k = 0;
for i=1:rows
    rowType = DI_Data(i,1);
    for j=1:numDIsPerSample
        start = (j*3)+(j-1);
        stop = (start + 2);
        DI_array{rowType,j}(end+1,1:5) = [rowType, j, DI_Data(i,start:stop)];
    end
end


% ---------------------------------------------------------------
% Plot DIs
% ---------------------------------------------------------------
fprintf('Plotting DIs.\n');
clf;

numPtsPerCloud = size(DI_array{1,1},1);
if (mode == 0)
    numPtClouds = numDIsPerSample;
    titleString = ['All DIs for the ',types{gesture_type},' gesture by participant #',num2str(participantID)];
else
    numPtClouds = numGestureTypes;
    titleString = ['DI #', int2str(mode),' for all gesture types.'];
end
scatterPts = cell(numPtClouds);


% Extract the cells from DI_array to be scatter plotted.
if (mode == 0)
    % Collect points for the input gesture type.
    fprintf('Printing all DIs for gesture type %s.\n', types{gesture_type});
    for i=1:numPtClouds
        scatterPts{i} = DI_array{gesture_type,i};
    end
else
    % Collect all DIs of a given index.
    fprintf('Printing DI #%d for all gesture types. \n', mode);
    for i=1:numPtClouds
        scatterPts{i} = DI_array{i,mode};
        % scatterPts{i}(1:5,:)
    end
end

for i=1:numPtClouds
    switch i
        case 1
            c = 'r.';
        case 2
            c = 'g.';
        case 3
            c = 'b.';
        case 4
            c = 'y.';
        case 5
            c = 'c.';
        otherwise
            fprintf('Bad numPtClouds: %d\n',numPtClouds);
    end
    % fprintf('Plotting (numPtClouds = %d, i = %d, c = %s)\n', numPtClouds, i, c);
    plot3(scatterPts{i}(:,5),scatterPts{i}(:,3),-scatterPts{i}(:,4),c);
    hold on;

end
hold off;
```

```
xlabel('X'); ylabel('Y'); zlabel('Z');
title(titleString);
if (mode == 0)
    legend('DI 1', 'DI 2', 'DI 3', 'DI 4', 'DI 5', 'Location', 'Northeast');
else
    legend('Come', 'Go', 'Stop', 'Location', 'Northeast');
end



fprintf('Done.\n');
```

A.2.1.14 plot_epochs2.m

```
function [] = plot_epochs2(scenario, kval);

fdata = dlmread('/home/pyanik/ros_workspace/kinect/bin/results');

% Results data format:
% results = [gestureType, epochNum, E];
type_Col = 1;
epoch_Col = 2;
E_Col = 3;


come = 1;
go = 2;
stop = 3;
eat = 4;
read = 5;
sleep = 6;
get = 7;
give = 8;
therapy = 9;

gestureNames = {'Come','Go','Stop','Eat','Read','Sleep','Get','Give','Therapy'};
plotColors = {'r.-','b.-','g.-','c.-','m.-','y.-','r.-','b.-','g.-'};

what2plot = unique(fdata(:,type_Col))';
numGestures = max(fdata(:,type_Col));
numSamples = size(fdata,1);
numEpochs = max(fdata(:,epoch_Col));

results = zeros(numGestures,numEpochs);
samplesPerEpoch = zeros(numGestures,numEpochs);


% Collect total error per epoch
for i = 1:numSamples

    gestType = fdata(i,type_Col);
    error = fdata(i,E_Col);
    epoch = fdata(i,epoch_Col);

    results(gestType,epoch) = results(gestType,epoch) + error;

    samplesPerEpoch(gestType,epoch) = samplesPerEpoch(gestType,epoch) + 1;

end

% Compute averages.
for i = what2plot

    for j = 1:numEpochs

        results(i,j) = results(i,j) / samplesPerEpoch(i,j);

    end
end



switch scenario
    case 1
        scenario_str = '(no neighbors considered)';
    case 2
        scenario_str = '(neighbors < mean considered)';
    case 3
        scenario_str = '(all neighbors considered)';
    case 4
        scenario_str = '(distance matrix with inf)';
    case 5
        scenario_str = '(clumpiness matrix)';
```

```
    case 6
        scenario_str = '(resistance distance)';
    case 7
        scenario_str = ['(kNN: k = ',num2str(kval),')'];
    otherwise
        fprintf('Bad Scenario:  %d.\n', scenario);
end

titleStr = ['Gesture Response Error ',scenario_str];

t = 1:1:numEpochs;
numCurves = size(what2plot,2);

for i = 1:numCurves

    gest2plot = what2plot(1,i);

    plot(t(1,:), results(gest2plot,:),plotColors{gest2plot});

    hold on;

end

total_error = sum(sum(results));

xlabel('Epoch');
ylabel('Average distance to goal (m)');
title(titleStr);
legend(gestureNames{what2plot}, 'Location', 'Northeast');

hold off;


fprintf('Total error = %.2f\n', total_error);

fprintf('Done.\n');
```

A.2.1.15 plot_epochs3.m

```
function [] = plot_epochs3(filename,scenario,kVal);

params;

fdata = dlmread([workDir,filename]);

% Results data format:
% results = [gestureType, epochNum, E];
type_Col = 1;
epoch_Col = 2;
E_Col = 3;


come = 1;
go = 2;
stop = 3;
eat = 4;
read = 5;
sleep = 6;
get = 7;
give = 8;
therapy = 9;

gestureNames = {'Come','Go','Stop','Eat','Read','Sleep','Get','Give','Therapy'};
plotColors = {'r.-','b.-','g.-','c.-','m.-','y.-','r.-','b.-','g.-'};

what2plot = unique(fdata(:,type_Col))';
numGestures = max(fdata(:,type_Col));
numSamples = size(fdata,1);
numEpochs = max(fdata(:,epoch_Col));

results = zeros(numGestures,numEpochs);
samplesPerEpoch = zeros(numGestures,numEpochs);


% Collect total error per epoch
for i = 1:numSamples

    gestType = fdata(i,type_Col);
    error = fdata(i,E_Col);
    epoch = fdata(i,epoch_Col);

    results(gestType,epoch) = results(gestType,epoch) + error;

    samplesPerEpoch(gestType,epoch) = samplesPerEpoch(gestType,epoch) + 1;

```

```
end

% Compute averages.
for i = what2plot

    for j = 1:numEpochs

        results(i,j) = results(i,j) / samplesPerEpoch(i,j);

    end
end



switch scenario
    case 1
        scenario_str = '(no neighbors considered)';
    case 2
        scenario_str = '(neighbors < mean considered)';
    case 3
        scenario_str = '(all neighbors considered)';
    case 4
        scenario_str = '(Floyd distance matrix)';
    case 5
        scenario_str = '(clumpiness matrix)';
    case 6
        scenario_str = '(resistance distance)';
    case 7
        scenario_str = ['(kNN: k = ',num2str(kVal),')'];
    otherwise
        fprintf('Bad Scenario:  %d.\n', scenario);
end

titleStr = ['Gesture Response Error ',scenario_str];

t = 1:1:numEpochs;
numCurves = size(what2plot,2);

for i = 1:numCurves

    gest2plot = what2plot(1,i);

    plot(t(1,:), results(gest2plot,:),plotColors{gest2plot});

    hold on;

end

total_error = sum(sum(results));

xlabel('Epoch');
ylabel('Average distance to goal (m)');
title(titleStr);
legend(gestureNames{what2plot}, 'Location', 'Northeast');

hold off;


fprintf('Total error = %.2f\n', total_error);

fprintf('Done.\n');
```

A.2.1.16 plot_epochs.m

```
function [] = plot_epochs(numEpochs, scenario, kval);

fileData = dlmread('/home/pyanik/ros_workspace/kinect/bin/results');


numGestures = max(fileData(:,1));
come = 1;
go = 2;
stop = 3;
eat = 4;
read = 5;
sleep = 6;
get = 7;
give = 8;
therapy = 9;

gestureNames = {'Come','Go','Stop','Eat','Read','Sleep','Get','Give','Therapy'};
plotColors = {'r.-','b.-','g.-','c.-','m.-','y.-','r.-','b.-','g.-'};


% Choose any 6 of the gestures above (only 6 colors):
```

```
% what2plot = [4 5 6 7 8 9];
what2plot = [1 2 3];



totalSamples = size(fileData,1);
samplesPerEpoch = totalSamples/numEpochs;

sampleCounts = zeros(numGestures,1);
epochAvgs = zeros(numGestures, numEpochs);

epoch_GNG_AvgE = zeros(numEpochs,1);


% Find the number of gestures of each type sampled.
for i=1:totalSamples

    gestType = fileData(i,1);

    sampleCounts(gestType,1) = sampleCounts(gestType,1) + 1;

end

sampleCounts = sampleCounts / numEpochs;

% fprintf('Come = %3d, go = %3d, stop = %3d.\n', ...
%     sampleCounts(1,1), sampleCounts(2,1), sampleCounts(3,1));


k = 0;
for i=1:numEpochs

    oneEpochAvgs = zeros(numGestures, 1);

    avgE = 0;

    for j=1:samplesPerEpoch

        k = k + 1;

        gestType = fileData(k,1);
        error = fileData(k,2);

        % Add up error for each gesture type.
        oneEpochAvgs(gestType,1) = oneEpochAvgs(gestType,1) + error;

        % Add up the GNG cloud error (column 3).
        % avgE = avgE + fileData(k,3);

    end

    % epoch_GNG_AvgE(i,1) = avgE / samplesPerEpoch;


    for j=1:numGestures

        if (sampleCounts(j,1) ~= 0)

            oneEpochAvgs(j,1) = oneEpochAvgs(j,1)/sampleCounts(j,1);

        end

    end

%     fprintf('ComeAvg = %12.8f, goAvg = %12.8f, stopAvg = %12.8f \n', ...
%         oneEpochAvgs(1,1), oneEpochAvgs(2,1), oneEpochAvgs(3,1));


    epochAvgs(:,i) = oneEpochAvgs;

end

t = 1:1:numEpochs;

% epoch_GNG_AvgE(1,:);

switch scenario
    case 1
        scenario_str = '(no neighbors considered)';
    case 2
        scenario_str = '(neighbors < mean considered)';
    case 3
        scenario_str = '(all neighbors considered)';
    case 4
        scenario_str = '(distance matrix with inf)';
    case 5
        scenario_str = '(clumpiness matrix)';
```

```
    case 6
        scenario_str = '(resistance distance)';
    case 7
        scenario_str = ['(kNN: k = ',num2str(kval),')'];
    otherwise
        fprintf('Bad Scenario:  %d.\n', scenario);
end

titleStr = ['Gesture Response Error ',scenario_str];


numCurves = size(what2plot,2)

for i = 1:numCurves

    gest2plot = double(what2plot(1,i));

    plot(t(1,:), epochAvgs(gest2plot,:),plotColors{gest2plot});

    hold on;

end

total_error = sum(sum(epochAvgs));

xlabel('Epoch');
ylabel('Average distance to goal (m)');
title(titleStr);
legend(gestureNames{what2plot}, 'Location', 'Northeast');

%
% subplot(2,1,2);
% plot(t(1,:), epoch_GNG_AvgE(:,1)', 'r.-');
% xlabel('Epoch');
% ylabel('Average GNG Node Error');
% title('Average GNG Node Error Per Epoch');


hold off;


fprintf('Total error = %.2f\n', total_error);

fprintf('Done.\n');
```

A.2.1.17 plot_pos.m

```
function [] = plot_pos(gestureType);

posData = dlmread('/home/pyanik/ros_workspace/kinect/bin/POS.txt');


% Possible gesture_types:
come = 1;
go = 2;
stop = 3;
eat = 4;
read = 5;
sleep = 6;
get = 7;
give = 8;
therapy = 9;

% ---------------------------------------------------------------
% Read in POS data
% ---------------------------------------------------------------
fprintf('Reading POS data.\n');

[rows,cols] = size(posData);
numCells = max(posData(:,1));

% Store samples in a cell array.
samples = cell(numCells,1);

% Each sample consists of:
% [gestureSample],[gestureType],[frameNum],[x, y, z]

% 1 3 10   0.00799585   0.99651375   0.22241513

for i=1:rows
   cellNum = posData(i,1);
   cellRow = posData(i,3);

   samples{cellNum}(cellRow,1:5) = posData(i,2:6);
end
```

```
% ---------------------------------------------------------------
% Read in POS data
% ---------------------------------------------------------------
fprintf('Plotting POS data.\n');
clf;

gestureTypeCol = 1;
frameNumCol = 2;
xCol = 3;
yCol = 4;
zCol = 5;

for i = 1:numCells

    if (samples{i}(gestureTypeCol) == gestureType)

        xData = samples{i}(:,xCol);
        yData = samples{i}(:,yCol);
        zData = samples{i}(:,zCol);
        tData = samples{i}(:,frameNumCol);


        subplot(3,1,1)
        plot(tData,xData,'b.-'); hold on;

        subplot(3,1,2)
        plot(tData,yData,'r.-'); hold on;

        subplot(3,1,3)
        plot(tData,zData,'g.-'); hold on;

    end


end

hold off;

% % ---------------------------------------------------------------
% % Plot gesture curves with DIs
% % ---------------------------------------------------------------
% fprintf('Plotting gesture curve.\n');
%
% % for i=1:numCells
% %     if (samples{i,1}(1,1) == gesture_type)
% %         plot3(samples{i,1}(:,5), samples{i,1}(:,3), samples{i,1}(:,4));
% %         hold on;
% %     end
% % end
%
% k = 0;
% scatterPts1 = zeros(1,3);
% scatterPts2 = zeros(1,3);
% scatterPts3 = zeros(1,3);
% scatterPts4 = zeros(1,3);
% scatterPts5 = zeros(1,3);
% for i=1:numDIs
%     if (DIs(i,1) == gesture_type)
%         k = k + 1;
%         switch DIs(k,2)
%             case 1
%                 scatterPts1(k,1:3) = [DIs(i,5), DIs(i,3), DIs(i,4)];
%             case 2
%                 scatterPts2(k,1:3) = [DIs(i,5), DIs(i,3), DIs(i,4)];
%             case 3
%                 scatterPts3(k,1:3) = [DIs(i,5), DIs(i,3), DIs(i,4)];
%             case 4
%                 scatterPts4(k,1:3) = [DIs(i,5), DIs(i,3), DIs(i,4)];
%             case 5
%                 scatterPts5(k,1:3) = [DIs(i,5), DIs(i,3), DIs(i,4)];
%             otherwise
%                fprintf('Cannot find DI number');
%
%         end % switch
%     end
% end
%
%
%
% % DIs(1:10,:)
% % scatterPts;
% % samples{2}(1:10,:)
%
% plot3(scatterPts1(:,1), scatterPts1(:,2), scatterPts1(:,3),'r.'); hold on;
% plot3(scatterPts2(:,1), scatterPts2(:,2), scatterPts2(:,3),'g.'); hold on;
% plot3(scatterPts3(:,1), scatterPts3(:,2), scatterPts3(:,3),'b.'); hold on;
```

```
% plot3(scatterPts4(:,1), scatterPts4(:,2), scatterPts4(:,3),'c.'); hold on;
% plot3(scatterPts5(:,1), scatterPts5(:,2), scatterPts5(:,3),'y.'); hold on;
% xlabel('X'); ylabel('Y'); zlabel('Z');
% title('Come gestures (Left Hand trajectory)');
% hold off;
%
%
%
% %
% % t = 1:1:numEpochs;
% %
% % epoch_GNG_AvgE(1,:);
% %
% %
% % % subplot(2,1,1);
% %
% % plot(t(1,:), epochAvgs(1,:), 'r.-', ...
% %      t(1,:), epochAvgs(2,:), 'b.-', ...
% %      t(1,:), epochAvgs(3,:), 'g.-');
% % xlabel('Epoch');
% % ylabel('Average distance to goal');
% % title('Gesture response error');
% % legend('Come closer', 'Go away', 'Stop', 'Location', 'Northeast');
%
% %
% % subplot(2,1,2);
% % plot(t(1,:), epoch_GNG_AvgE(:,1)', 'r.-');
% % xlabel('Epoch');
% % ylabel('Average GNG Node Error');
% % title('Average GNG Node Error Per Epoch');
% %
%

fprintf('Done.\n');
```

A.2.1.18 write_results.m

```
% ----------------------------------------------------------------------
% Script name:  write_results
% Author:  Paul Yanik
%
% Description:
% This script writes out results from gestureLrnList for one run of
% gestureLrnList (numEpochs * numSamples).  The name of the results_file is
% set in params
% ----------------------------------------------------------------------

FID = fopen(results_file, 'w');

numEntries = size(results_array,1);


for i=1:numEntries

    % results_array = [gestureType, epochNum, error]

    fprintf(FID, '%d %d %f\n', results_array(i, 1:3));

end

fclose(FID);
```

A.2.2 GNG Tools

A.2.2.1 adjustNeighbors.m

```
% ----------------------------------------------------------------------
% Script name:  adjustNeighbors
% Author:  Paul Yanik
%
% This script adjusts the topological neighbors of the winner node in
% the GNG cloud [A] by moving them a fraction (ep_n) toward the input
% vector.  The script also increments the ages of all edges emmanating from
% the winner.
%
% This script also yields the neighborhood (N) of the winner node (NN) for
% development of the response vector in later scripts.
% ----------------------------------------------------------------------

numEdges = size(C,1);
numNodes = size(A,1);
```

```

v1 = P.v1;
v2 = P.v2;
age = P.age;
nodeLabel = P.nodeLabel;
ep_n = P.ep_n;
fvec1 = P.fvec1;
fvec2 = P.fvec2;

N = [];
for k=1:numEdges

    found_nbr = 0;
    nbr = -99;

    if (C(k,v1) == s1)

        found_nbr = 1;
        nbr = C(k,v2);
        C(k,age) = C(k,age) + 1;

    end

    if (C(k,v2) == s1)

        found_nbr = 1;
        nbr = C(k,v1);
        C(k,age) = C(k,age) + 1;

    end

    if (found_nbr == 1)

        for p=1:numNodes


%             if (A(p,nodeLabel) == s1)
%
%                 % Store the winning node in NN.
%                 NN = A(p,:);
%
%             end

            if (A(p,nodeLabel) == nbr)

                if (s1_hot == 1)
                    adjustment = ep_n * (vec_in - A(p,fvec1:fvec2));
                else ...
                    adjustment = ep_n * (vec_in - A(p,fvec1:fvec2));
                end

                A(p,fvec1:fvec2) = A(p,fvec1:fvec2) + adjustment;

                % Store the neighborhood of the winner node in N.
                N(end+1,:) = A(p,:);

            end

        end

    end

end % for k
```

A.2.2.2 adjustWinner.m

```
% ---------------------------------------------------------------------
% Script name:  adjustWinner
% Author:  Paul Yanik
%
% Description:  This script adjust the winner node's local error and moves
% its feature vector closer to the input vector.
% ---------------------------------------------------------------------

numEdges = size(C,1);
numNodes = size(A,1);

E = P.E;
fvec1 = P.fvec1;
fvec2 = P.fvec2;
ep_w = P.ep_w;
nodeLabel = P.nodeLabel;
hot = P.hot;
reward = P.reward;

```

```
for k = 1:numNodes

    if (A(k,nodeLabel) == s1)

        % Step 5.
        % Add to the winner's local error.
        distance = Dv(1,1);

%         if (A(k,reward) ~= hot) % No change for trained nodes.
%             A(k,E) = A(k,E) + distance^2;
%         else ...
%             % No change
%         end

        % Error adjustment according to the GNG algorithm.
        A(k,E) = A(k,E) + distance^2;

        % Step 6.
        % Move the winner toward the input by a faction (ep_w)
        % of its current distance.


        fVec = A(k,fvec1:fvec2);

        % Disallow node movement if the node is trained
        s1_hot = 0;
        if (A(k,reward) == hot)
            adjustment = 0;
            s1_hot = 1; % Signal to neighbors not to move
        else ...
            adjustment = ep_w * (vec_in - fVec);
        end

        adjustment = ep_w * (vec_in - fVec);


        A(k,fvec1:fvec2) = A(k,fvec1:fvec2) + adjustment;


    end

end
```

A.2.2.3 ageColdNode.m

```
% ------------------------------------------------------------------------
% Script name:  ageColdNode
% Author:  Paul Yanik
%
% Description:  In the event of a new gesture falling into the receptive
% field of a node which has already been trained to handle a different
% gesture type, this script will find and artificially age the links of the
% oldest node elsewhere in the GNG cloud which has a cold reward.  This
% script should be called in getResponse_warmerColder if the situation
% described above applies.
% ------------------------------------------------------------------------

% Find a node to delete only if the GNG cloud is at max capacity.
maxNodeCnt = P.maxNodeCnt;
numNodes = size(A,1);
numEdges = size(C,1);
if (numNodes >= maxNodeCnt)

    max_ageTot = -99;
    max_ageLabel = -99;

    foundOne = 0;

    for i=1:numNodes

        % Find cold nodes
        if (A(i,P.reward) == P.cold)

            thisNode = A(i,P.nodeLabel);
            ageTot = 0;

            % Add the ages of all edges for this node.
            for j=1:numEdges

                v1 = C(j,P.v1);
                v2 = C(j,P.v2);

                if (v1 == thisNode || v2 == thisNode)
                    ageTot = ageTot + C(j,P.age);
                end
```

```
            end

            if (ageTot > max_ageTot)
                max_ageTot = ageTot;
                max_ageLabel = thisNode;
                foundOne = 1;
            end

        end

    end


    if (hoodRadius ~= 6) % Do not artificially age nodes when using resDist
        for i=1:numEdges

            v1 = C(i,P.v1);
            v2 = C(i,P.v2);

            if ((foundOne == 1) && ...
                    ((v1 == max_ageLabel) || (v2 == max_ageLabel)) )

                % Artificially age connections to the node.
                C(i,P.age) = P.ageMax + 1;
            end

        end

    end

end
```

A.2.2.4 check4Connection.m

```
% ----------------------------------------------------------------------
% Script name:  check4Connection
% Author:  Paul Yanik
%
% Description:
% This script checks for a connection in [C] between two nodes in
% a GNG cloud [A]. If the connection exists, it is refreshed.  Otherwise,
% it is created.
% ----------------------------------------------------------------------


% compareAC(A,C,P,'Check4Cnx1');

numEdges = size(C,1);
numNodes = size(A,1);

connectionExists = 0;
if (numEdges > 0)

    for k=1:numEdges

        edge = [C(k,P.v1), C(k,P.v2)];

        if ( mEq(edge,[s1,s2]) || mEq(edge,[s2,s1]) )

            % Refresh the connection.
            C(k,P.age) = 0;
            connectionExists = 1;

        end

    end

end

if (connectionExists == 0)

    % Establish the connection:  [v1, v2, age, length]
    newConx = [s1, s2, 0, 1];

    C(end+1,:) = newConx;

    % Update connection counts in [A].
    for j=1:numNodes

        if ( (A(j,P.nodeLabel)==s1) || (A(j,P.nodeLabel)==s2) )
            A(j,P.numConx) = A(j,P.numConx) + 1;
        end
    end

```

```
end


% compareAC(A,C,P,'Check4Cnx2');
```

A.2.2.5 compareAC

```
function [] = compareAC(A,C,P,locString)
% ---------------------------------------------------------------------
% Function name:  compareAC
% Author:  Paul  Yanik
%
% Description:  This function compares the GNG [A] and [C] matrices.  The
% number of connections in A are compared with the number of neighbors in
% C.  This is done to find a bug in which A and C did not reconcile when
% insterting a new node.
% ---------------------------------------------------------------------

numNodes = size(A,1);
numEdges = size(C,1);

for i=1:numNodes

    numConx = A(i,P.numConx);
    thisNode = A(i,P.nodeLabel);

    numNbrs = 0;
    for (j=1:numEdges)

        v1 = C(j,P.v1);
        v2 = C(j,P.v2);


        if (v1 == thisNode)
            numNbrs = numNbrs + 1;
        end

        if (v2 == thisNode)
            numNbrs = numNbrs + 1;
        end

    end

    if (numNbrs ~= numConx)
        fprintf('[%s] Node %d has %d conx in A but %d edges in C\n', ...
            locString, thisNode, numConx, numNbrs);
        myChar = input('Press any key: ','s');
    end


end



end % function
```

A.2.2.6 compareANN

```
function [] = compareANN(A,NN,P,locString)
% ---------------------------------------------------------------------
% Function name:  compareANN
% Author:  Paul Yanik
%
% Description:  This function compares certain fields of A(nodeLabel) with
% NN which has been selected by GNG.
% ---------------------------------------------------------------------

numNodes = size(A,1);


for i=1:numNodes

    if (A(i,P.nodeLabel) == NN(1,P.nodeLabel))

        AnumConx = A(i,P.numConx);
        NNnumConx = NN(1,P.numConx);


        if (AnumConx ~= NNnumConx);

            fprintf('[%s] A.numConx = %d, NN.numConx = %d\n', ...
                locString, AnumConx, NNnumConx);

```

```
            myChar = input('Press a key: ','s');

        end
    end

end


end % function
```

A.2.2.7 decreaseNodeError

```
% ----------------------------------------------------------------------
% Script name:  decreaseNodeError
% Author:  Paul Yanik
%
% Description:  This script performs two functions:
% 1) It decreases node error on all nodes in [A],
% 2) It increments the number of observations (numObs) field.
% ----------------------------------------------------------------------

% Decrease node error.
A(:,P.E) = A(:,P.E) - (P.beta * A(:,P.E));

% Increment number of observations.
A(:,P.numObs) = A(:,P.numObs) + 1;
```

A.2.2.8 edgeExists

```
% This function determines the existence of an edge in an undirected graph
% having vertices a and b (order indpendent).  It returns the row number of
% an existing edge in a graph matrix.

function [found, index] = edgeExists(C, v1_col, v2_col, edge)

found = 0;
index = -99;

numEdges = size(C,1);

for i=1:numEdges

    edge1 = [C(i,v1_col),C(i,v2_col)];
    edge2 = [C(i,v2_col),C(i,v1_col)];

   if ( mEq(edge,edge1) || mEq(edge,edge2) )

       found = 1;
       index = i;

   end


end
```

A.2.2.9 findMaxErr

```
% ----------------------------------------------------------------------
% Script name:  findMaxErr
% Author:  Paul Yanik
%
% Description: This script finds the node (q) in [A] with the maximum
% accumulated error and its neighbor (f) with max accumulated error.
% The values of q and f are interpreted as indices into [A].
% ----------------------------------------------------------------------

numNodes = size(A,1);
numEdges = size(C,1);

% compareAC(A,C,P,'FindMaxErr1');

% Column subscripts
nodeLabel = P.nodeLabel;
v1 = P.v1;
v2 = P.v2;
E = P.E;

% Find the node with max error.
% q is the row address of maxErr.
[maxErr,q] = max(A(:,P.E));

```

```

% Generate a list of the neighbors of q (by nodeLabel).
nbrs = [];
for i=1:numEdges

    if (C(i,v1) == A(q,nodeLabel))
        nbrs(end+1,1) = C(i,v2);
    elseif (C(i,v2) == A(q,nodeLabel))
        nbrs(end+1,1) = C(i,v1);
    end

end
numNbrs = size(nbrs,1);

% if (numNbrs == 0)
%
%     fprintf('nodeLabel = %d, numConx = %d\n',A(q,nodeLabel),A(q,P.numConx));
%
%     myChar = input('No neighbors.  Press a key: ','s');
%
% end


% Find the neighbor f of q with max error.
maxNbrErr = -99;
f = -99;
for i=1:numNbrs

    nbr = nbrs(i,1);

    for j=1:numNodes

%         fprintf('nbr = %d, node = %d, E = %.2f, maxE = %.2f\n', ...
%            nbr, A(j,nodeLabel), A(j,P.E), maxNbrErr);
%

        if ( (A(j,nodeLabel) == nbr) && (A(j,E) > maxNbrErr) )

            maxNbrErr = A(j,E);
            f = j; % index into A

        end

    end

end

% compareAC(A,C,P,'FindMaxErr2');
```

A.2.2.10 get2ClosestNodes

```
% ------------------------------------------------------------------------
% Script name:  get2ClosestNodes
% Author:  Paul Yanik
% Description: This script finds the two closest nodes in a GNG cloud
% [A] to an input vector (vec_in).
% ------------------------------------------------------------------------

numNodes = size(A,1);

fvec1 = P.fvec1;
fvec2 = P.fvec2;
nodeLabel = P.nodeLabel;

% Find the input vector's distance to all nodes, sort,
% and choose the two closest to the input vector.  Store the results in Dv.
Dv = zeros(numNodes,2);

for k = 1:numNodes
    A_fvec = A(k,fvec1:fvec2);
    distance = norm(vec_in - A_fvec);
    Dv(k,1:2) = [distance, A(k,nodeLabel)];
end % k

% Sort by distance (ascending order).
Dv = sortrows(Dv,1);
s1 = Dv(1,2); % winner nodeLabel
s2 = Dv(2,2); % 2nd nearest nodeLabel
```

A.2.2.11 getNNHood.m

```
% ------------------------------------------------------------------------
% Script name:  getNNHood
```

```
% Author:  Paul Yanik
%
% Description:  This script uses the nearest reference node number from the
% GNG algorithm (s1) to create two variables:  NN and N.  NN is the
% reference node and all associated variables from the A matrix.  N is the
% neighborhood of NN (all connected nodes).
% ---------------------------------------------------------------------------

numNodes = size(A,1);
numEdges = size(C,1);


NN = [];
for i = 1:numNodes

    if (A(i,P.nodeLabel) == s1)
        NN = A(i,:);
    end
end


% Add constrution of N later.  This is already done in adjustNeighborss.
% I was going to add it here for clarity... but it is redundant.
```

A.2.2.12 gng.m

```
% This script implements the Growing Neural Gas (GNG) algorithm.


% Runtime parameters.
params;

% ---------------
% Step 1
% ---------------
% This step attemps to read in or initialize both the A and C matrices.
% fprintf('Step 1.  Reading A and C.\n');
read_A;
read_C;

% ---------------
% Step 2
% ---------------
% Select a vector.  Use the vec_in read by the calling function.
% fprintf('Step 2.\n');

% ---------------
% Step 3
% ---------------
% fprintf('Step 3. get2ClosestNodes.\n');
% compareAC(A,C,P,'Get2Closest1');
get2ClosestNodes;

% ---------------
% Step 4
% ---------------
% Refresh or establish the connection between the 2 nearest nodes in C.
% fprintf('Step 4. check4Connections.\n');
check4Connection;

% ---------------
% Step 5, Step 6
% ---------------
% Adjust the winner node's error and feature vector.
% fprintf('Steps 5 and 6.  adjustWinner.\n');
adjustWinner;

% ---------------
% Step 7
% ---------------
% Adjust the winner node's topological neighbors and increment
% winner node's connection ages.  Also, construct the neighbohood for
% later use.
% fprintf('Step 7.  adjustNeighbors.\n');
adjustNeighbors;
% compareANN(A,NN,P,'Chk4Conx2');

% ---------------
% Step 8
% ---------------
% Remove connections with an age greater tham ageMax.
% fprintf('Step 8. removeOldConnections.\n');
removeOldConnections;
% compareANN(A,NN,P,'RemConx2a');

```

```
% ---------------
% Step 9
% ---------------
% Insert a new node if necessary (based on lamdba).
% fprintf('Step 9. insertNewNode.\n');
insertNewNode;
% compareANN(A,NN,P,'InsertNode2');

% ---------------
% Step 10
% ---------------
% Decrease the error of all units.
% fprintf('Step 10. decreaseNodeError.\n');
% compareAC(A,C,P,'DecreaseE1');
decreaseNodeError;
% compareAC(A,C,P,'DecreaseE2');


% Get NN (and N in the future).
getNNHood;
```

A.2.2.13 insertNewNode.m

```
% ------------------------------------------------------------------------
% Script name:  insertNewNode
% Author:  Paul  Yanik
%
% Description:  This script inserts a new node into the GNG cloud as
% needed.  This is classically based on lambda, but other conditions are
% added for this implementaion.
% ------------------------------------------------------------------------

numNodes = size(A,1);
numEdges = size(C,1);

% compareAC(A,C,P,'Insert1');


numObservations = A(1,P.numObs) + 1;

if (((mod(numObservations,P.lambda)==0) && (numNodes<P.maxNodeCnt)) || ...
        (new_node_needed == 1))

    % fprintf('Adding a new node.\n');

    % Find nodeLabel for node with maxErr (q)
    % and it's neighbor with maxErr (f).
    findMaxErr;

    % Interpolate between nodes q and f to produce a new node.
    interpolateNodes;


end

% compareAC(A,C,P,'Insert2');
```

A.2.2.14 interpolateNodes.m

```
% ------------------------------------------------------------------------
% Script name:  interpolateNodes
% Author:  Paul Yanik
%
% Description: This script interpolates between nodes q and f (calculated in
% findMaxErr) in the GNG cloud to produce a new node, r.
% ------------------------------------------------------------------------

% compareAC(A,C,P,'Interpolate1');


[maxNodeLabel, row] = max(A(:,P.nodeLabel));

% for i=1:numNodes
%
%     if (A(i,nodeLabel) > maxNodeLabel)
%         maxNodeLabel = A(i,nodeLabel;
%     end
%
% end

% Calculate the new node's feature vector.
% fprintf('numNodes = %d, q = %d, f = %d\n', size(A,1),q,f);
wr = (A(q,P.fvec1:P.fvec2) + A(f,P.fvec1:P.fvec2))/2;
```

```
% Decrease the error on nodes q and f.
A(q,P.E) = A(q,P.E) - (P.alpha * A(q,P.E));
A(f,P.E) = A(f,P.E) - (P.alpha * A(f,P.E));

% Create the new node r, and update the necessary fields.
r = zeros(1,size(A(1,:),2));
r(1,P.nodeLabel) = maxNodeLabel + 1;
r(1,P.numConx) = 2;
r(1,P.reward) = -1;
r(1,P.ancestor) = r(1,P.nodeLabel);
r(1,P.Q) = norm(wr);
r(1,P.E) = (A(q,P.E) + A(f,P.E))/2;

% r(1,P.act1:P.act2) = (A(q,P.act1:P.act2) + A(f,P.act1:P.act2))/2;
% r(1,P.last1:P.last2) = (A(q,P.last1:P.last2) + A(f,P.last1:P.last2))/2;

% Set actions/last actions for new nodes to untrained status.
r(1,P.act1:P.act2) = [0,0,0];
r(1,P.last1:P.last2) = [0,0,0];


r(1,P.fvec1:P.fvec2) = wr;

A(end+1,:) = r;



%         node = struct('numObs', n(1,1), ...
%                       'nodeLabel', n(1,2), ...
%                       'numConx', n(1,3), ...
%                       'reward', n(1,4), ...
%                       'Q', n(1,5), ...
%                       'E', n(1,6), ...
%                       'action', n(1,7:9), ...
%                       'last', n(1,10:12), ...
%                       'featureVec', n(1,13:13+numFeatures-1) );



% Remove the defunct connection between q and f.
edges2Delete = [];
for i=1:numEdges

    % Generate a list of C indices to remove.
    % Note:  this list should only have 1 element.

    % The q-f connection.
    if ( (C(i,P.v1) == A(q,P.nodeLabel)) && ...
         (C(i,P.v2) == A(f,P.nodeLabel)) )

        edges2Delete(end+1) = i;

    elseif ( (C(i,P.v1) == A(f,P.nodeLabel)) && ...
             (C(i,P.v2) == A(q,P.nodeLabel)) )

        edges2Delete(end+1) = i;

    end

end
% size(C)
% edges2Delete

% Remove the connection from C.
C(edges2Delete,:) = [];

% size(C)

% Add the new connections {q,r} and {f,r}
C(end+1,:) = [A(f,P.nodeLabel), r(1,P.nodeLabel), 0, 1.0];
C(end+1,:) = [A(q,P.nodeLabel), r(1,P.nodeLabel), 0, 1.0];


% compareAC(A,C,P,'Intrpolate2');
```

A.2.2.15 print_N.m

```
% This script prints a neighborhood of GNG nodes.

function [] = print_N(hood, P);

hoodSize = size(hood,1)


```

```
for n = 1:hoodSize

    action = hood(n,P.act1:P.act2);
    last = hood(n,P.last1:P.last2);

    Q = norm(action);

    fprintf('[print_N]: Node %2d, actn=(%4.2f,%4.2f,%4.2f), last=(%4.2f,%4.2f,%4.2f), rwd=%d, anc=%d, Q=%5.3f\n',
      ...
        hood(n,P.nodeLabel), ...
        action(1), action(2), action(3), ...
        last(1), last(2), last(3), ...
        hood(n,P.reward), ...
        hood(n,P.ancestor), ...
        Q ...
        );


end
```

A.2.2.16 read_A.m

```
% ------------------------------------------------------------------------
% Script name:  read_A
% Author:  Paul Yanik
%
% Description:
% This script checks to see if the A matrix exists. If it does not exist
% then it checks for the file A.txt and reads it into the A matrix for the
% Growing Neural Gas algorithm.
% ------------------------------------------------------------------------

params;


% Check to see if the A matrix or A.txt file exists.
if (exist ('A', 'var'))

    % Do nothing.
    numNodes = size(A,1);

elseif (exist(A_file, 'file'))

    % Read in the A matrix from file A.txt.
    A = dlmread(A_file);

    % numNodes = number of rows of A.txt.
    numNodes = size(A,1);
    fprintf('Existing A.txt file contains %d nodes.\n', numNodes);


% Use of cells of structs was horribly slow.
%         node = struct('numObs', n(1,1), ...
%                       'nodeLabel', n(1,2), ...
%                       'numConx', n(1,3), ...
%                       'reward', n(1,4), ...
%                       'Q', n(1,5), ...
%                       'E', n(1,6), ...
%                       'action', n(1,7:9), ...
%                       'last', n(1,10:12), ...
%                       'featureVec', n(1,13:13+numFeatures-1) );
%
%         A{i} = node;


else ...

    fprintf('[A] does not exist.  Init with %d nodes.\n', ...
        num_initial_GNG_nodes);

    % Inialize A with a defined number of random nodes.
    A = zeros(num_initial_GNG_nodes,P.A_cols);

    for i=1:num_initial_GNG_nodes

        A(i,P.numObs) = 0;
        A(i,P.nodeLabel) = i;
        A(i,P.numConx) = 0;

        % Initialize rewards pessimistically - trigger random guess.
        A(i,P.reward) = -1;

        % New nodes are their own ancestor.
        A(i,P.ancestor) = A(i,P.nodeLabel);
```

```

        % Initial length and error are zero.
        A(i,P.Q) = 0;
        A(i,P.E) = 0;

        % Initialize actions to the origin of the arena (e.g. TurtleSim).
        A(i,P.act1:P.act2) = [0,0,0];
        A(i,P.last1:P.last2) = [0,0,0];

        % Feature vector
        for j=P.fvec1:P.fvec2
            A(i,j) = rand_in_range(0.0,0.5);
        end


    end

end
```

A.2.2.17 read_C.m

```
% This script reads C.txt for the Growing Neural Gas algorithm if no
% C matrix currently exists.

params;

% Check to see if the C matrix or C.txt file exists.
if (exist('C', 'var'))

    % Do nothing
    numEdges = size(C,1);

elseif (exist(C_file, 'file'))

    % Read in C.txt
    C = dlmread(C_file);

    numEdges = size(C,1);
    fprintf('Existing C.txt file contains %d edges.\n',numEdges);

else ...

    % fileNotFound(C_file);

    % Initialize C to an empty matrix.
    C = [];

end
```

A.2.2.18 read_descriptor_list.m

```
function[featureVecs, classNums, numSamples] = read_descriptor_list(fname)
% ---------------------------------------------------------------------------
% Function name:  read_descriptor_list
% Author:  Paul Yanik
%
% Description:  This function reads feature vectors from a specifid input
% file as input to the GNG algorithm.
% ---------------------------------------------------------------------------

params;


% Assume that the descriptor file is stored in dataDir.
fname = [dataDir,fname];


if (exist(fname, 'file'))

    % Do nothing
    % fprintf('Reading in %s.\n', fname);

    fileData = dlmread(fname);
    numSamples = size(fileData,1);

    % descriptor_list = cell(numSamples,1);

    featureVecs = zeros(numSamples, numFeatures);
    classNums = zeros(numSamples, 1);

    for i=1:numSamples

        s = fileData(i,:);
```

```

        featureVecs(i,1:numFeatures) = s(1,2:2+numFeatures-1);
        classNums(i,1) = s(1,1);

    end

else ...
    fileNotFound(fname);
end
```

A.2.2.19 removeOldConnections.m

```
% This script removes edges in the C matrix which have aged beyond
% a fixed limit (ageMax).

numEdges = size(C,1);
numNodes = size(A,1);

% compareANN(A,NN,P,'RemOldCnx1');



newC = [];
for i=1:numEdges

    % Look for connections that are too old.
    if (C(i,P.age) > P.ageMax)

        % Find the vertices.
        v1 = C(i,P.v1);
        v2 = C(i,P.v2);

        % Decrement connection counts for the vertex nodes.
        for j=1:numNodes

            thisNode = A(j,P.nodeLabel);

            if ((thisNode==v1) || (thisNode==v2))
                A(j,P.numConx) = A(j,P.numConx) - 1;

%                 if (NN(1,P.nodeLabel) == thisNode)
%                    NN(1,P.numConx) = NN(1,P.numConx) - 1;
%                 end


            end

        end


    else ...

        % The connection is young enough to keep.
        newC(end+1,:) = C(i,:);

    end
end
C = newC;

% compareANN(A,NN,P,'RemOldCnx1a');


% Remove nodes with zero connections from A.
newA = [];
for z=1:numNodes;

    if (A(z,P.numConx) > 0)
        newA(end+1,:) = A(z,:);
    end

end
A = newA;


% compareANN(A,NN,P,'RemOldCnx2');
```

A.2.2.20 write_A.m

```
% This function writes the A matrix to the file A.txt
% for the Growing Neural Gas algorithm.

params;

```

```
FID = fopen(A_file, 'w');

numNodes = size(A,1);

for i=1:numNodes

    fprintf(FID, '%d ', A(i,P.numObs));
    fprintf(FID, '%d ', A(i,P.nodeLabel));
    fprintf(FID, '%d ', A(i,P.numConx));
    fprintf(FID, '%d ', A(i,P.reward));
    fprintf(FID, '%d ', A(i,P.ancestor));
    fprintf(FID, '%f ', A(i,P.Q));
    fprintf(FID, '%f ', A(i,P.E));

    for j=P.act1:P.act2
        fprintf(FID, '%f ', A(i,j));
    end

    for j=P.last1:P.last2
        fprintf(FID, '%f ', A(i,j));
    end

    for j=P.fvec1:P.fvec2
        fprintf(FID, '%f ', A(i,j));
    end


    fprintf(FID, '\n');

end

fclose(FID);
```

A.2.2.21 write_C.m

```
% This function writes the C matrix to the file C.txt
% for the Growing Neural Gas algorithm.

params;

v1 = P.v1;
v2 = P.v2;
age = P.age;
len = P.len;

FID = fopen(C_file, 'w');

numEdges = size(C,1)

for i=1:numEdges

    fprintf(FID, '%d ', C(i,v1));
    fprintf(FID, '%d ', C(i,v2));
    fprintf(FID, '%d ', C(i,age));
    fprintf(FID, '%f ', C(i,len));
    fprintf(FID, '\n');

%     fprintf(FID, '%d ', C{i,1}.v1);
%     fprintf(FID, '%d ', C{i,1}.v2);
%     fprintf(FID, '%d ', C{i,1}.age);
%     fprintf(FID, '\n');

end

fclose(FID);
```

A.2.3 Graph Tools

A.2.3.1 admittance_matrix.m

```
function[Av] = admittance_matrix(C, v1_col, v2_col, edge_len_col)
% ---------------------------------------------------------------------------
% Function name:  admittance_matrix
%
% Description:
% This function calculates the admittance (Kirchhoff) matrix (Av) for an
% undirected graph (C) with node-pair vertices and edge lengths.  Each edge
% is defined as a row vector with a column # for vertex 1, a column # for
% vertext 2, and a column # for length.
%
% This matrix is used in calculation of a graph's resistance distance
```

```
% matrix.  The resistance distance only applies if the graph is connected.
% This function includes a warning for Inf admittance.
% ---------------------------------------------------------------------

% Column vectors of edge vertices.
v1_vec = C(:,v1_col);
v2_vec = C(:,v2_col);

% Graph parameters.
numNodes = max(max(v1_vec), max(v2_vec));
numEdges = size(C,1);

% Node Adjacency matrix (Av).
Av = zeros(numNodes);
bad_admittance = 0;
for i=1:numEdges

    vertex1 = C(i,v1_col);
    vertex2 = C(i,v2_col);

    % Add 1 to C(i,edge_len_col) for cases where the age of the link is 0.
    % This prevents a singular age matrix.

    admittance = 1/(C(i,edge_len_col) + 1); % 1

    if (admittance == Inf)
        bad_admittance = 1;
        C
        fprintf('Inf Y: Edge length between nodes %d and %d = %8.3f.\n', ...
            vertex1, vertex2, C(i,edge_len_col));
    end

    Av(vertex1, vertex2) = admittance; % G1(i,3);
    Av(vertex2, vertex1) = admittance; % G1(i,3);
end

end % function
```

A.2.3.2 centrality.m

```
% This function computes the centrality of a network.  Various centrality
% measures exist and may be (eventually) selectable by an input parameter.
%
% 9/23/2012:  Implement degree centrality and closeness centrality.



function[centrality_matrix] = centrality(centrality_type, Adj_matrix, C, Params)


v1 = Params.v1;
v2 = Params.v2;

numNodes = max(max(C(:,v1:v2)));



switch centrality_type


    case 1 % Degree centrality.

        % Estrada, eq. (7.1)
        centrality_matrix = sum(Adj_matrix);

    case 2 % Closeness centrality

        % This algorithm assumes that all links have a length < inf.

        D = floyd(C,Params)

        s = sum(D)

        % Estrada, eq. (7.49)
        centrality_matrix = (numNodes-1) ./ s

    otherwise

        fprintf('Invalid centrality type specified:  %d.\n', centrality_type);

end
```

A.2.3.3 clumpiness.m

```
function [Xi] = clumpiness(C,P,A)
% ---------------------------------------------------------------------
% Function name: clumpiness
% Author:  Paul Yanik
%
% Description:
% This function computes a matrix of clumpiness coefficients (Xi) for a
% graph based on a topology of connections (C), distances between nodes (D),
% and the degree matrix (K).
%
% This method is described in Estrada (2012), The Structure of Complex
% Networks.
% ---------------------------------------------------------------------
v1 = P.v1;
v2 = P.v2;
nodeLabel = P.nodeLabel;

numNodes = max(max(C(:,v1:v2)));

AdjMat = nodeAdjacency(C,P);

% Find the nodeDegree matrix (number of connections)
[k, K_matrix] = nodeDegree(AdjMat);

D = floyd(C,P,A);

Xi = zeros(numNodes);


% Compute the clumpiness coefficients
for i=1:numNodes
    for j=1:i

        if (i ~= j)

            clump_val = (k(i) * k(j)) / D(i,j)^2;

            Xi(i,j) = clump_val;
            Xi(j,i) = clump_val;


        end

    end
end


end % function
```

A.2.3.4 edgeLengths.m

```
% This function generates a square length matrix from an array of graph
% edge lengths:  [vertex1, vertex2, length].


function[lengths] = edgeLengths(C)


numEdges = size(C,1);
numNodes = max(max(C(:,1:2)));

lengths = inf(numNodes);


for i=1:numEdges

    v1 = C(i,1);
    v2 = C(i,2);

    lengths(v1,v2) = C(i,3);
    lengths(v2,v1) = C(i,3);


end
```

A.2.3.5 floyd.m

```
function [D] = floyd(C, P, A)
% ---------------------------------------------------------------------
% Function name:  floyd
% Author:  Paul Yanik
%
% Description:
```

```
% This function implements Floyd's algorithm for finding the minimum
% distance between all pairs of nodes in a network.
% Refer to A. Tucker, "Applied Combinatorics", p. 129.
%
% Currently (9/16/2012), it is assumed that a matrix consisting of rows of
% the form [vertex1, vertex2, age, length] is available.  Future
% implementations may use the adjacency matrix [Adj] to infer edge lengths.
% ---------------------------------------------------------------------------

numEdges = size(C,1);
numNodes = max(max(C(:,1:2)));
v1 = P.v1;
v2 = P.v2;
len = P.edgeLen_col;
nodeLabel = P.nodeLabel;
ancestor = P.ancestor;
reward = P.reward;
age = P.age;

% Initialize all interNode distances to infinity.
D = zeros(numNodes);
W = inf(numNodes);

D = D + triu(W,1); % Add upper triangle of inf.
D = D + tril(W,-1); % Add lower triange of inf.

for i=1:numEdges

    vertex1 = C(i,v1);
    vertex2 = C(i,v2);

    D(vertex1,vertex2) = C(i,len);
    D(vertex2,vertex1) = C(i,len);

end


% Find the minimum distance between all pairs of nodes
for k = 1:numNodes
    for i = 1:numNodes
        for j = 1:numNodes

            temp = D(i,k) + D(k,j);

            if (temp < D(i,j))

                D(i,j) = temp;

            end

        end
    end
end


% Lengthen distance between nodes where bad response guidance was given.
% This should reduce clumpiness of badly associated nodes.
Arows = size(A,1);
for i=1:Arows

    my_ancestor = A(i,ancestor);
    my_nodeLabel = A(i,nodeLabel);

    % if (((A(i,reward) == -1) && (my_ancestor ~= my_nodeLabel)) ...
    %     || (A(i,reward) == 0))

    if ((A(i,reward) == -1) && (my_ancestor ~= my_nodeLabel))

        D(my_nodeLabel,my_ancestor) = inf;
        D(my_ancestor,my_nodeLabel) = inf;


    end

end
```

A.2.3.6 laplacian.m

```
function[L] = laplacian(K, Av);
% ---------------------------------------------------------------------------
% Function name:  laplacian
% Author:  Paul Yanik
%
% Description:
% This script computes the Laplacian matrix given an adjacency
```

```
% (or admittance) matrix [Av] and a degree matrix [K].
%
% This formula is taken from E. Estrada, "The Structure of Complex
% Networks", p. 38, eq. 2.32.
% ---------------------------------------------------------------------

L = K - Av;
```

A.2.3.7 nodeAdjacency.m

```
function[adjMatrix] = nodeAdjacency(C,P)
% ---------------------------------------------------------------------
% Function name:  nodeAdjacency
% Author:  Paul Yanik
%
% Descripton:
% This function/script generates a node-adjacency matrix from the [C] matrix
% of a Growing Neural Gas network.  [C] is understood to be an undirected
% graph.
%
% Allocate a square matrix that is the size of the largest node label
% (which may be larger than the number of nodes).
% ---------------------------------------------------------------------

v1 = P.v1;
v2 = P.v2;

numNodes = max(max(C(:,v1:v2)));
numEdges = size(C,1);

adjMatrix = zeros(numNodes);


% Since [C] is an undirected graph, adjMatrix will be symmetric.
for i=1:numEdges

    vertex1 = C(i,v1);
    vertex2 = C(i,v2);

    adjMatrix(vertex1, vertex2) = 1;
    adjMatrix(vertex2, vertex1) = 1;

end




% % Check for accuracy (debug purposes).
% Arows = size(A,1);
% nodeLabel = P.nodeLabel;
% numConx = P.numConx;
% for i=1:numNodes
%
%     a = sum(adjMatrix(:,i));
%
%     for j=1:Arows
%
%         if ( (A(j,nodeLabel)==i) && (A(j,numConx)~=a))
%
%             fprintf('ERR: NodeLabel=%3d, adj=%2d, numConx=%2d.\n', ...
%                 i,a,A(j,numConx));
%
%         elseif ( (A(j,nodeLabel)==i) && (A(j,numConx)==a) )
%
%             fprintf('OK:  NodeLabel=%3d, adj=%2d, numConx=%2d.\n', ...
%                 i,a,A(j,numConx));
%
%         else ...
%                % Do nothing
%
%         end
%     end
%
% end
```

A.2.3.8 nodeDegree.m

```
function[k_vector,K_matrix] = nodeDegree(Av)
% ---------------------------------------------------------------------
% Function name:  nodeDegree
%
% Description:
% This function computes the vector [k] and matrix of node degrees [K] from
```

```
% the adjancency matrix or admittance matrix [Av].
% ----------------------------------------------------------------------

k_vector = sum(Av)';

K_matrix = diag(k_vector);

end % function
```

A.2.3.9 resDist.m

```
function[Omega] = resDist(C, v1_col, v2_col, edge_len_col)
% ----------------------------------------------------------------------
% Function name:  resDist
% Author:  Paul  Yanik
%
% Description:
% This function computes the resistance distance for a network based on the
% Laplacian matrix (based on the method of Klien and Randic, 1995).
% ----------------------------------------------------------------------

numEdges = size(C,1)
numNodes = max( max(C(:,v1_col)), max(C(:,v2_col)) )


% Admittance (Kirchhoff) matrix.
Av = admittance_matrix(C, v1_col, v2_col, edge_len_col)

% Degree matrix.
[k_vec, K] = nodeDegree(Av)


% Laplacian matrix (L)
L = laplacian(K,Av)


% Auxiliary Matrix (Phi) and
% inverse sumMatrix (non-singluar for connected graphs).
Phi = ones(numNodes);
sumMat = L + (Phi/numNodes);
sumMatInv = inv(sumMat);


% Resistance Distance Matrix (Omega) -- symmetric
for i=1:numNodes
    for j=1:i;

        resistance = sumMatInv(i,i) - 2*sumMatInv(i,j) + sumMatInv(j,j);

        Omega(i,j) = resistance;
        Omega(j,i) = resistance;

    end

    % This prevents a node from having the lowest R-distance to itself.
    Omega(i,i) = inf;
end


end % function
```

A.2.4 *k*NN Tools

A.2.4.1 gl_kNN2.m

```
% ======================================================================
% Filename:  gl_kNN2.m
% Author:  Paul Yanik
%
% Description:  This file contains code which emulates gestureLrnList.m
% using kNN from a DI training set as reference nodes instead of GNG.  The
% training data is read in.  Test data is then read in.  Near neighbors are
% found in the training data.  The longest action vector among the near
% neighbors is selected as the action to be taken.  The action vector for
% the training data is then lengthened by a learning step.
%
%
% Use model:  gl_knn(tstData, numEpochs, k)
%
% Where:
% descr_file is a file containing dynamic instant (DI) training data.
```

```
% ========================================================================

function[] = gl_kNN2(tstData, numEpochs, hoodRadius, kVal,kNN_buff)

tic


% This file contains runtime parameters for gestureLrn.
params;
oneShot = 0;


% Variables
neighbors_used = 0;
neighbors_used_successfully = 0;
hot_nodes = 0;


% Read in training DIs from file (e.g. DIs_300_real_trn.txt).
% [trnVecs, trnClassNums, numTrnSamples] = read_descriptor_list(trnData);


% Start with a raw 2-node A matrix.
read_A;

% Initialize a neighborhood structure.
kNN_hood = zeros(kVal,size(A,2));
fprintf('kVal = %d\n', kVal);

% Read in test DIs from file (e.g. DIs_450_real_tst.txt).
[tstVecs, tstClassNums, numSamples] = read_descriptor_list(tstData);


% Results array for one run of numEpochs.
% Format of results:  [classNum, Err].
results_array = zeros(numSamples*numEpochs, 2);



% Run numEpochs
index = 0;
for epoch = 1:numEpochs

    fprintf('Epoch = %3d\n', epoch);


    for sample = 1:numSamples

        index = index + 1;

        vec_in = tstVecs(sample,:);
        gestureClass = tstClassNums(sample,1);

        % Use the most recent samples (bottom of A).
        Arows = size(A,1);
        maxNeighbors = kNN_buff;
        if (Arows > maxNeighbors)
            A = A(Arows - maxNeighbors + 1:Arows,:);
        end

        [NN, kNN_hood] = kNN_A_noClassifier(kVal,A,vec_in);

        genAction_xyt;
        getResponse_warmerColder;


        % Put results in the results matrix.
        % index = ((epoch-1)*numSamples)+sample;
        results_array(index, 1:3) = [gestureClass, epoch, mag_dist2goal];

    end


end

write_results;



fprintf('Done.\n');


toc;
```

A.2.4.2 gl_kNN.m

```
% ======================================================================
% Filename:  gl_kNN.m
% Author:  Paul Yanik
%
% Description:  This file contains code which emulates gestureLrnList.m
% using kNN from a DI training set as reference nodes instead of GNG.  The
% training data is read in.  Test data is then read in.  Near neighbors are
% found in the training data.  The longest action vector among the near
% neighbors is selected as the action to be taken.  The action vector for
% the training data is then lengthened by a learning step.
%
%
% Use model:  gl_knn(tstData, numEpochs, k)
%
% Where:
% descr_file is a file containing dynamic instant (DI) training data.
% ======================================================================

function[] = gl_kNN(trnData, tstData, numEpochs, hoodRadius, kVal)

tic
kNN_buff = 0;


% This file contains runtime parameters for gestureLrn.
params;

% Variables
neighbors_used = 0;
neighbors_used_successfully = 0;
hot_nodes = 0;


% Read in training DIs from file (e.g. DIs_300_real_trn.txt).
[trnVecs, trnClassNums, numTrnSamples] = read_descriptor_list(trnData);

% Create an A matrix (similar to A for GNG) from training data.
A = read_A_kNN(trnVecs);


% Initialize a neighborhood structure.
kNN_hood = zeros(kVal,size(A,2));
fprintf('kVal = %d\n', kVal);

% Read in test DIs from file (e.g. DIs_450_real_tst.txt).
[tstVecs, tstClassNums, numSamples] = read_descriptor_list(tstData);


% Results array for one run of numEpochs.
% Format of results:  [classNum, Err].
results_array = zeros(numSamples*numEpochs, 2);



% Run numEpochs
index = 0;
for epoch = 1:numEpochs

    fprintf('Epoch = %3d\n', epoch);


    for sample = 1:numSamples

        index = index + 1;

        vec_in = tstVecs(sample,:);
        gestureClass = tstClassNums(sample,1);

        [NN, kNN_hood] = kNN_A_noClassifier(kVal,A,vec_in);

        genAction_xyt;
        getResponse_warmerColder;


        % Put results in the results matrix.
        % index = ((epoch-1)*numSamples)+sample;
        results_array(index, 1:3) = [gestureClass, epoch, mag_dist2goal];

    end


end

write_results;
```

```
fprintf('Done.\n');


toc;
```

A.2.4.3 kNN_A_noClassifier.m

```
% ========================================================================
% Filename:  kNN_A_noClassifier.m
% Author:  Paul Yanik
%
% Description:  This function finds the k nearest neighbors from a set of
% training data.  The original application for this code was in support of
% my research at Clemson University on gesture recognition with mapping to
% a robot configuration vector (x,y,theta).  As such, this function does
% not classify the output, but only returns the indices of the k nearest
% neighbors of an input vector to a set of sample vectors (A).
%
% Use model:  myNbrs = kNN_A_noClassifier(k,A,mySample)
% ========================================================================

function [NN, nbrs] = kNN_A_noClassifier(k, A, mySample)

params;


% Determine the number and dimension of training data vectors.
% The input trnData is assumed to contain input samples in each row.
[rows,cols] = size(A);


% Set up a results matrix to store distances and indices
distances = zeros(rows,2);
sorted_results = zeros(rows,2);

% Calculate distances
for n = 1:rows

    A_vec = A(n,P.fvec1:P.fvec2);

    distances(n,1) = A(n,P.nodeLabel);
    distances(n,2) = norm(A_vec - mySample);

end

% Sort the rows of the results matrix by distance.
% The top k entries are the nearest neighbors
sorted_results = sortrows(distances,2);
numNbrs = min(k, size(A,1)); % in case A has fewer than k entries.
nbr_labels = sorted_results(1:numNbrs,:);


% Pull the nbr_label rows out of A
nbrs = zeros(numNbrs,cols);
for i = 1:numNbrs
    for j = 1:rows

        if (A(j,P.nodeLabel) == nbr_labels(i,1))

            nbrs(i,:) = A(j,:);

        end
    end
end

% Return the nearest neighbor
NN = nbrs(1,:);



end % function
```

A.2.4.4 read_A_kNN.m

```
% This function generates an A matrix similar to the one used with GNG.  This
% affords access to all fields of A for generating actions and feedback
% when using kNN.

function [A] = read_A_kNN(vecs)

```

```
[numVecs,cols] = size(vecs);

params;

% Initialize A
A = zeros(numVecs,P.A_cols);

for i = 1:numVecs

    A(i,P.numObs) = 0;
    A(i,P.nodeLabel) = i;
    A(i,P.numConx) = 0;

    % Initialize rewards pessimistically - trigger random guess.
    A(i,P.reward) = -1;

    % Nodes are their own ancestor.
    A(i,P.ancestor) = A(i,P.nodeLabel);

    % Initial length and error are zero.
    A(i,P.Q) = 0;
    A(i,P.E) = 0;

    % Initialize actions to the origin of the TurtleSim arena.
    A(i,P.act1:P.act2) = [0,0,0];
    A(i,P.last1:P.last2) = [0,0,0];

    % Feature vector
    A(i,P.fvec1:P.fvec2) = vecs(i,:);

end

fprintf('A matrix contains %d nodes.\n', size(A,1));


end % function
```

A.2.5 SSM Tools

A.2.5.1 classifyHOGs.m

```
% ======================================================================
% Filename:  classifyHOGs.m
% Author:    Paul Yanik
% Date:      October, 2010
%
% Description:
% This program classifies three motions (reach, grab, press) using a mean
% Histograms of Oriented Gradients (HOG) as the descriptor for motion
% sensor data readings taken from various points on spherical surface
% surrounding the motions. Classification is performed using a Bayesian
% classifier (closest to the mean).  A classification is made and the
% closest actual HOG (using Frobenius norm) to the mean is taken to be the
% optimal vantage point for viewing that particular motion (reach, grab,
% or press).
% ======================================================================
function[dist2mean,closest2mean,stats] = classifyHOGs(sensorType, ...
    patchHeight,patchWidth,exemplarMethod,trnPercent,genViews,version, ...
    verbose);

% Read in runtime parameters.
HOGParams;

% Generate a view list for those views that will be used
% to calculate exemplars versus those whic will be used as
% views to be classified.
if (genViews == 1)
    [viewList,numTrnViews,numTstViews] = genRandViewList(trnPercent);
else
    load('viewList.mat');
end

% Statistics collection matrix:
% Each row is a class: [#actual, #found, P(error)]
% stats = zeros(numTstViews*numClasses,3);
stats = zeros(numClasses,3);

% Store the actual motion that is closest to mean for each class.
% Initialize 3x7 matrix: [theta,phi,FrobDist2mean,class,x,y,z]
closest2mean = zeros(numClasses,7);
closest2mean(:,3) = 999999;

% Store all distances to the mean exemplars.
```

```
% [1,2,3] = [dist2reach, dist2press, dist2grab];
dist2mean = zeros(numTstViews*numClasses,numClasses+8);
% The other (8) column labels (someday, make these variable
% so that we can add more classes).
actualClass = 4;
foundClass = 5;
rCol = 6;
tCol = 7;
pCol = 8;
xCol = 9;
yCol = 10;
zCol = 11;

% Consider all views from one exemplar at a time
for explrNum = 1:numClasses

    if (explrNum == reach)
        explrType = 'reach';
    elseif (explrNum == press)
        explrType = 'press';
    elseif (explrNum == grab)
        explrType = 'grab';
    else fprintf('BAD CLASS NUMBER\n');
    end

    % Calculate exemplar for the current class.
    if (exemplarMethod == 3)
        % Generate a mean HOG exemplar using the randomized view list.
        exemplar = genHOGexemplar(explrType,sensorType,viewList,3,version,0);
    else fprintf('Unrecognized exemplar generation method: %d.\n', ...
            exemplarMethod);
    end

    motNum = 0; % Motion number (1 of numPts)
    trainPtsCnt = 0;
    testPtsCnt = 0;
    viewListIndex = 0;
    for theta = thetaMin:thetaInterval:thetaMax
        for phi = phiMin:phiInterval:phiMax

            viewListIndex = viewListIndex + 1;

            if (viewList(viewListIndex,1) == 1)
                trainPtsCnt = trainPtsCnt + 1;
            else

                % Perform classification for 0s in viewlist.
                testPtsCnt = testPtsCnt + 1;

                % Compare each motion at this vantage point to exemplar.
                for c = 1:numClasses

                    motNum = motNum + 1;

                    if (c == reach)
                        motionType = 'reach';
                    elseif (c == press)
                        motionType = 'press';
                    elseif (c == grab)
                        motionType = 'grab';
                    else fprintf('BAD CLASS NUMBER\n');
                    end

                    mHOG = genArrayHOG(patchHeight,patchWidth, ...
                            motionType,sensorType,theta,phi,version,0);

                    dist2mean(motNum,explrNum) = ...
                        norm(exemplar - mHOG,'fro');
                    dist2mean(motNum,actualClass) = c;

                    dist2mean(motNum,rCol:pCol) = [r,theta,phi];
                    [x,y,z] = sph2cart(d2r*phi, ((pi/2)-(d2r*theta)), r);
                    dist2mean(motNum,xCol:zCol) = [x,y,z];

                end % for c (each motion at this view)

            end % if motNum in viewList

        end % for phi
    end % for theta
end % for explrNum

trainPtsCnt;
testPtsCnt;

% Find the nearest exemplar (by distance to Frob. mean) and classify.
for i = 1:(numTstViews*numClasses)
```

```
    [val,classFound] = min(dist2mean(i,1:3));
    dist2mean(i,foundClass) = classFound;
end

dist2mean(:,1:5);


% Calculate the statistics for each class.
% Row = [#actual, #found, %error]
stats = zeros(numClasses,3);

for i = 1:(numTstViews*numClasses)

    a = dist2mean(i,actualClass);
    f = dist2mean(i,foundClass);

    % Increment actual motion count.
    stats(a,1) = stats(a,1) + 1;

    % Increment found motion count.
    if (a == f)
        stats(a,2) = stats(a,2) + 1;
    end

end

% Calculate P(error) and avgPerr
avgPerr = 0;
for i = 1:numClasses
    stats(i,3) = 1 - (stats(i,2) / stats(i,1));
    avgPerr = avgPerr + stats(i,3);
end
avgPerr = avgPerr / i;

% Open an output file for writing.
% filename = 'ExhaustiveExemplarSearch.txt';
% FID = fopen(filename,'w');

if (verbose == 1)
fprintf('----------------------------------------------------\n');
fprintf('Classification stats:                                \n\n');
fprintf('          Actual     Found      P(error)     \n');
fprintf('Reach:       %3d       %3d        %6.3f\n',stats(reach,1),stats(reach,2),stats(reach,3));
fprintf('Press:       %3d       %3d        %6.3f\n',stats(press,1),stats(press,2),stats(press,3));
fprintf('Grab :       %3d       %3d        %6.3f\n\n',stats(grab ,1),stats(grab ,2),stats(grab ,3));
fprintf('Average P(error) = %6.3f\n',avgPerr);
fprintf('----------------------------------------------------\n');
end

% Find the closest vantage point to the mean.
for i = 1:(numTstViews*numClasses)
    a = dist2mean(i,actualClass);
    f = dist2mean(i,foundClass);
    d = dist2mean(i,a);

    if ((a == f) && (d < closest2mean(a,3)) )
        % Store a new nearest vantage point
        closest2mean(a,1) = dist2mean(i,tCol); % theta
        closest2mean(a,2) = dist2mean(i,pCol); % phi
        closest2mean(a,3) = d; % Frob. distance
        closest2mean(a,4) = a; % class - redundant
        closest2mean(a,5) = dist2mean(i,xCol); % x
        closest2mean(a,6) = dist2mean(i,yCol); % y
        closest2mean(a,7) = dist2mean(i,zCol); % z
    end
end

if (verbose == 1)
fprintf('Exemplars found:                                \n\n');
fprintf('Reach:   theta = %3d, phi = %3d, dist2mean = %6.3f \n', closest2mean(reach,1), closest2mean(reach,2),
      closest2mean(reach,3));
fprintf('Press:   theta = %3d, phi = %3d, dist2mean = %6.3f \n', closest2mean(press,1), closest2mean(press,2),
      closest2mean(press,3));
fprintf('Grab :   theta = %3d, phi = %3d, dist2mean = %6.3f \n', closest2mean(grab, 1), closest2mean(grab, 2),
      closest2mean(grab ,3));
fprintf('----------------------------------------------------\n');
end


% Graph the Frobenius distances of each HOG from the mean HOG
% versus the geometric distance of each vantage point from the vantage
% point that was found to be the best.
% [dist2mean, dist2view]
% geomDist = zeros(numViews,numClasses);
% frobDist = zeros(numViews,numClasses);
% viewPos = zeros(1:3);
% meanPos = zeros(1:3);
```

```
%
% for i = 1:numClasses
%
%     meanPos = closest2mean(i,5:7); % [x,y,z] of best vantage point.
%
%     k = 0;
%     for j = 1:numPts
%         if (dist2mean(j,actualClass) == i)
%             k = k + 1;
%             viewPos = dist2mean(j,9:11);
%             frobDist(k,i) = dist2mean(j,i);
%             geomDist(k,i) = norm(viewPos - meanPos,'fro');
%         end
%     end % for j
% end % for i


% plotting = 0;
% if (plotting == 1)
%     scatter(geomDist(:,reach),frobDist(:,reach),'filled'); hold on;
%     scatter(geomDist(:,press),frobDist(:,press),'filled'); hold on;
%     scatter(geomDist(:,grab ),frobDist(:,grab ),'filled');
%
%     % scatter(geomDist(:,reach),frobDist(:,reach)); hold on;
%     % scatter(geomDist(:,press),frobDist(:,press)); hold on;
%     % scatter(geomDist(:,grab ),frobDist(:,grab ));
%
%     title('Distances from mean view to mean HOG.');
%     xlabel('Distance to best vantage point');
%     ylabel('Dist to mean HOG');
%     legend('Reach','Press','Grab','Location','Northwest');
% end
```

A.2.5.2 classifyHOGsPlot.m

```
% ======================================================================
% Filename:  classifyHOGsPlot.m
% Author:    Paul Yanik
% Date:      October, 2010
%
% Description:
% ======================================================================
function[x,y,myZ,distances2mean] = classifyHOGsPlot(sensorType,motion, ...
    patchHeight,patchWidth,exemplarMethod,trnPercent,genViews,version);

% Read in runtime parameters.
HOGParams;

% Generate a view list for those views that will be used
% to calculate exemplars (training data) versus those which will be
% used as views to be classified (test data).
if (genViews == 1)
    [viewList,numTrnViews,numTstViews] = genRandViewList(trnPercent);
else
    load('viewList.mat');
end

% Store all distances to the mean exemplars.
% [1,2,3] = [dist2reach, dist2press, dist2grab];
distances2mean = zeros(numViews,numClasses+3);
% The other column labels:
rCol = 4;
tCol = 5;
pCol = 6;

z = zeros(thetaVals,phiVals,3);

% Consider all views from one exemplar at a time
for c = 1:numClasses

    if (c == reach)
        explrType = 'reach';
    elseif (c == press)
        explrType = 'press';
    elseif (c == grab)
        explrType = 'grab';
    else fprintf('BAD CLASS NUMBER\n');
    end

    % Calculate exemplar for the current class.
    if (exemplarMethod == 3)
        % Generate a mean HOG exemplar using the randomized view list.
        exemplar = genHOGexemplar(explrType,sensorType,viewList,3,version,0);
    else fprintf('Unrecognized exemplar generation method: %d.\n', ...
            exemplarMethod);
```

```
    end

    k = 0;
    t = 0;
    p = 0;

    for theta = thetaMin:thetaInterval:thetaMax
        t = t + 1;
        p = 0;
        for phi = phiMin:phiInterval:phiMax
            p = p + 1;

            k = k + 1;
            distances2mean(k,rCol) = r;
            distances2mean(k,pCol) = phi;
            distances2mean(k,tCol) = theta;

            if (c == reach)
                motionType = 'reach';
            elseif (c == press)
                motionType = 'press';
            elseif (c == grab)
                motionType = 'grab';
            else fprintf('BAD CLASS NUMBER\n');
            end

            mHOG = genArrayHOG(patchHeight,patchWidth, ...
                    motionType,sensorType,theta,phi,version,0);

            Q = norm(exemplar - mHOG,'fro');

            distances2mean(k,c) = Q;

            z(t,p,c) = Q;

        end % for phi
    end % for theta
end % for c

z;

x = thetaMin:thetaInterval:thetaMax;
y = phiMin:phiInterval:phiMax;
myZ = z;

if (motion == 1) mString = 'Reach'; end
if (motion == 2) mString = 'Press'; end
if (motion == 3) mString = 'Grab';  end


titleString = ['Dist:  HOG to class exemplar: ',mString,', Train = ',num2str(trnPercent),'%'];
surf(x,y,myZ(:,:,motion)');
title(titleString);
xlabel('Theta');
ylabel('Phi');
zlabel('Frob. Distance');
set(gca,'XTick',thetaMin:30:thetaMax);
set(gca,'YTick',phiMin:30:phiMax);
view(-45,75);
```

A.2.5.3 findOptVPs.m

```
% ---------------------------------------------------------------------
% Filename:  genMotSSMHybrid.m
% Author:    Paul Yanik
% Date:      January, 2011
%
% Description:
% This function runs classifyHOGs multiple times and plots the
% optimal vantage points to see if they cluster.  Also, the most common
% optimal vantage point found for each motoin through all trials is
% determined.
% ----------------------------------------------------------------------
function [dist2mean, optimalVPs] = findOptVPs(numTrials,version,displayMe)

HOGParams;

% Draw the spherical wire frame.
if (displayMe == 1)
    drawSphereFrame;
end

% Create a matrix to store optimal vantage point counts from
% each run of the classifier.
VPStats = zeros(numViews,6); % Vantage Point Stats
```

```
i = 0;
for t = thetaMin:thetaInterval:thetaMax
    for p = phiMin:phiInterval:phiMax
        i = i + 1;
        VPStats(i,:) = [30,t,p,0,0,0];
    end
end

% -------------------------------------------------------
% Run the classifier and plot the optimal vantage points.
% -------------------------------------------------------
reach = 1;
press = 2;
grab = 3;

xCol = 5;
yCol = 6;
zCol = 7;

% numTrials = 1;

reachXYZ = zeros(numTrials,3);
pressXYZ = zeros(numTrials,3);
grabXYZ  = zeros(numTrials,3);

for i=1:numTrials

    fprintf('Performing classification trial: %3d.\n',i);

    [dist2mean,closest2mean,stats] = ...
        classifyHOGs('Motion',1,1,3,50,1,version,0);

    % reach
    x = closest2mean(reach,xCol);
    y = closest2mean(reach,yCol);
    z = closest2mean(reach,zCol);
    reachXYZ(i,:) = [30,closest2mean(reach,1),closest2mean(reach,2)];
    r = 30;
    t = closest2mean(reach,1);
    p = closest2mean(reach,2);
    for g = 1:numViews
        if ((VPStats(g,1) == r) && (VPStats(g,2) == t) && (VPStats(g,3) == p))
            VPStats(g,reach+3) = VPStats(g,reach+3) + 1;
        end
    end

    if (displayMe == 1)
        scatter3(x,y,z,10,'ro','filled');
        hold on;
    end

    % press
    x = closest2mean(press,xCol);
    y = closest2mean(press,yCol);
    z = closest2mean(press,zCol);
    pressXYZ(i,:) = [30,closest2mean(press,1),closest2mean(press,2)];
    r = 30;
    t = closest2mean(press,1);
    p = closest2mean(press,2);
    for g = 1:numViews
        if ((VPStats(g,1) == r) && (VPStats(g,2) == t) && (VPStats(g,3) == p))
            VPStats(g,press+3) = VPStats(g,press+3) + 1;
        end
    end

    if (displayMe == 1)
        scatter3(x,y,z,10,'bo','filled');
        hold on;
    end

    % grab
    x = closest2mean(grab,xCol);
    y = closest2mean(grab,yCol);
    z = closest2mean(grab,zCol);
    grabXYZ(i,:) = [30,closest2mean(grab,1),closest2mean(grab,2)];
    r = 30;
    t = closest2mean(grab,1);
    p = closest2mean(grab,2);
    for g = 1:numViews
        if ((VPStats(g,1) == r) && (VPStats(g,2) == t) && (VPStats(g,3) == p))
            VPStats(g,grab+3) = VPStats(g,grab+3) + 1;
        end
    end

    if (displayMe == 1)
        scatter3(x,y,z,10,'ko','filled');
        hold on;
```

```
    end

end
% fprintf('Done.\n');

[bestReach,bestReachLoc] = max(VPStats(:,reach+3));
[bestPress,bestPressLoc] = max(VPStats(:,press+3));
[bestGrab,  bestGrabLoc] = max(VPStats(:,grab +3));

reachRTP = [VPStats(bestReachLoc,1),VPStats(bestReachLoc,2),VPStats(bestReachLoc,3)];
pressRTP = [VPStats(bestPressLoc,1),VPStats(bestPressLoc,2),VPStats(bestPressLoc,3)];
grabRTP  = [VPStats(bestGrabLoc, 1),VPStats(bestGrabLoc, 2),VPStats(bestGrabLoc, 3)];

optimalVPs = [reachRTP; pressRTP; grabRTP];

fprintf('Best reach at (%3d,%3d,%3d).  (%3d / %3d).\n', ...
    reachRTP(1,1),reachRTP(1,2),reachRTP(1,3),bestReach,numTrials);

fprintf('Best press at (%3d,%3d,%3d).  (%3d / %3d).\n', ...
    pressRTP(1,1),pressRTP(1,2),pressRTP(1,3),bestPress,numTrials);

fprintf('Best grab  at (%3d,%3d,%3d).  (%3d / %3d).\n', ...
    grabRTP(1,1),grabRTP(1,2),grabRTP(1,3),bestGrab,numTrials);
```

A.2.5.4 genArrayHOG.m

```
% ----------------------------------------------------------------------
% Filename:  genArrayHOG.m
% Author:    Paul Yanik
% Date:      September, 2010
%
% Description:
% This function returns the average of HOGs surrounding the vantage point
% (r,theta,phi) by 'rows' and 'cols'.
% ----------------------------------------------------------------------
function[arrayHOG] = genArrayHOG(rows,cols, ...
                motionType,sensorType,theta,phi,version,display)

HOGParams;

% Set up a matrix of thetas and phis over which to generate an
% average HOG.
thetas = zeros(1,rows);
phis = zeros(1,cols);

% Create a rows*cols "patch" of vantage points with the input
% vantage point (theta,phi) in the upper right corner

% Mirror any overSteps from the region (thetaMax, phiMax)
% before the angleMax
overage = 0;
for i = 1:cols
    phis(1,i) = phi + ((i-1) * phiInterval);
    if (phis(1,i) > phiMax)
        overage = phis(1,i) - phiMax;
        phis(1,i) = phiMax - overage;
    end
end

for j = 1:rows
    thetas(1,j) = theta + ((j-1) * thetaInterval);
    if (thetas(1,j) > thetaMax)
        overage = thetas(1,j) - thetaMax;
        thetas(1,j) = thetaMax - overage;
    end
end

phis;
thetas;

thisPhi = 0;
thisTheta = 0;
k = 0;
for i = 1:cols
    thisPhi = phis(1,i);

    for j = 1:rows
        k = k + 1;

        thisTheta = thetas(1,j);

        if (display == 1)
            fprintf('%5s ',motionType);
            fprintf('(r,t,p) = (%2d,%3d,%3d), ',r,theta,phi);
            fprintf('[Rows,Cols] = [%1d,%1d], ',rows,cols);
```

```
            fprintf('Local (t,p) = (%3d,%3d), ',thisTheta,thisPhi);
            fprintf('k = %d ',k);
            fprintf('\n');
        end

        inFileName = ['HOG_',motionType,'_',sensorType,'_r', ...
                    num2str(r),'t', num2str(thisTheta),'p', ...
                    num2str(thisPhi),'_v',num2str(version),'.mat'];

        load(inFileName);

        if (k == 1)
            totalHOG = hog;
        else totalHOG = totalHOG + hog;
        end

    end
end

arrayHOG = totalHOG / k;
```

A.2.5.5 genHOGexemplar.m

```
% ----------------------------------------------------------------------
% Filename:  genHOGexemplar.m
% Author:    Paul Yanik
% Date:      November, 2010
%
% Description:
% This program finds a mean HOG over selected views for a given motion
% type and sensor type.  This mean can be used as an exemplar for
% classification.
% ----------------------------------------------------------------------

function[meanHOG] = genHOGexemplar(motionType,sensorType, ...
    viewList,exemplarType,version,display);

HOGParams;

% Exemplar types (not finished implementing all these):
% 1 = one specific viewpoint
% 2 = uniform distribution
% 3 = randomized

% Calculate a cumulative exemplar over selected HOGs (from viewList).
k = 0;
q = 0;
for theta = thetaMin:thetaInterval:thetaMax
    for phi = phiMin:phiInterval:phiMax

        if (display == 1)
            fprintf('findHOGMean: ');
            fprintf('%s sensor(%s): ',sensorType,motionType);
            fprintf('(r,theta,phi) = (%2d,%3d,%3d).\n',r,theta,phi);
        end

        % Use k to index viewList.
        k = k + 1;

        if (viewList(k,1) == 1)

            % Use q to count HOGs factored into the exemplar.
            q = q + 1;

            inFileName = ['HOG_',motionType,'_',sensorType,'_r', ...
                num2str(r),'t',num2str(theta),'p',num2str(phi), ...
                '_v',num2str(version),'.mat'];

            load(inFileName);

            if (q == 1)
                meanHOG = hog;
            else
                meanHOG = meanHOG + hog;
            end

        end % checking viewList

    end % for phi
end % for theta

meanHOG = meanHOG / q;
```

A.2.5.6 genHOG.m

```
% ----------------------------------------------------------------------
% Filename:  genHOG.m
% Author:    Paul Yanik
% Date:      August, 2010
%
% Use model:
% genHOG(motionType,SensorType,r,theta,phi)
%
% Description:
% This function generates a HOG for points along the main diagonal of
% a Self Similarity Matrix (SSM).  The cells of the HOG are hardcoded in
% log-polar form for eleven regions as described in Junejo, et al. (2008).
%
% Reference:
% I. Junejo, E. Dexter, I. Laptev and P. Perez, "Cross-View Action
% Recognition from Temporal Self-Similarities", European Conference
% on Computer Vision -- ECCV 2008, 2008, pp. 293-306.
% ----------------------------------------------------------------------

function [hog] = genHOG(motionType,sensorType,r,theta,phi,version,display)

if (display == 1)
    fprintf('genHOG: %6s sensor (%5s), (r,theta,phi) = (%2d,%3d,%3d), ver %d, ', ...
        sensorType, motionType, r, theta, phi, version);
end

% Generate the fileName from which to read the SSM (called 'D').
inFileName = ['SSM_',motionType,'_',sensorType,'_r',num2str(r),'t', ...
    num2str(theta),'p',num2str(phi),'_v',num2str(version),'.mat'];

% Generate the fileName to which to write the HOG matrix.
outFileName = ['HOG_',motionType,'_',sensorType,'_r',num2str(r),'t', ...
    num2str(theta),'p',num2str(phi),'_v',num2str(version),'.mat'];

% Read in cell boundaries
getCellBoundsHOG;

% Test matrix:
% D = [0 1 2 3; 0 0 3 4; 0 0 0 5; 0 0 0 0];

% Read in the SSM: "D".
load(inFileName);
[numSamples, cols] = size(D);

% Set up angle quantization bins (in radians/bin).
numBins = 8;
twoPi = 2*pi;
binSize = twoPi/numBins;


% Calculate gradients at all points on or above the diagonal in D.
% Use Pruitt gradient calculation.  Compute gradient_x (Gx) and
% gradient_y (Gy) for each point.  Then convert each pair (Gx,Gy)
% to the equivalent (magnitude,theta).

% Number of points on or above the diagonal
numGradients = 0.5*(numSamples^2) + 0.5*numSamples;

% 'gradients' stores magnitude and quantized angle of gradient for
% each point on or above the diagonal of D.
gradients = zeros(numGradients,2);

k = 0;
% fprintf('Finding gradients.\n');
for i = 1:numSamples % rows of D
    % fprintf('i = %3d\n',i);
    for j = i:numSamples % columns of D

        k = k + 1;

        % Calculate Gx
        if (j == 1) % first column
            Gx = D(i,j+1) - D(i,j);
        elseif (j == numSamples) % last column
            Gx = D(i,j) - D(i,j-1);
        else % normal case
            Gx = D(i,j+1) - D(i,j-1);
        end

        % Calculate Gy
        if (i == 1) % first row
            Gy = D(i,j) - D(i+1,j); % replicates first row
        elseif (i == numSamples) % last row
            Gy = D(i-1,j) - D(i,j);
```

```
        else % normal case
            Gy = D(i-1,j) - D(i+1,j);
        end

        % Find the gradient magnitude

        % Manhattan distance
        gradients(k,1) = abs(Gx) + abs(Gy);

        % Euclidean distance:
%         gradients(k,1) = norm([Gx,0] + [0,Gy]);

        % Find gradient angle, quanitize it into bins

        theta = atan2(Gy,Gx);
        if (theta < 0)
            theta = theta + twoPi;
        end
         binNum = floor(theta/binSize) + 1;

        % Fix an apparent MatLab bug
        if (binNum > 8)
            binNum = binNum - 8;
        elseif (binNum < 0)
            binNum = binNum + 8;
        end

        gradients(k,2) = binNum;


    end % for j
end % for i

clear D;

% Starting from each point on the diagonal of D,
% calculate the cell number (1-11) of all other points
% and tally the histogram.
% fprintf('Finding HOG.\n');

hog = zeros(numSamples * 11, numBins);

k = 0;
for i = 1:numSamples % rows of D.
    % fprintf('i = %3d\n',i);
    for j = i:numSamples % columns of D.
        k = k + 1;

        if (i == j) % a point not on the diagonal

            % Calculate the descriptor for this point (cell HOGs).
            t = 0;
            for r = 1:numSamples
                for s = r:numSamples
                    t = t + 1;

                    % Find the distance to other points.

                    % Manhattan distance:
                    dist = abs(r - i) + abs(s - j);

                    % Euclidean distance:
                    % dist = norm([r,s] - [i,j]);
                    angle = atan2((-r+i),(s-j));
                    if (angle < 0)
                        angle = angle + twoPi;
                    end


                    % HOG is log-polar.
                    % Consider distances at 1, 10, 100 from the point.
                    % Consider angular regions centered at even multiples
                    % of pi/8.

                    getCellNumHOG;

                    % degAngle = angle * (360/twoPi);

                    row = (i-1)*11 + cellNum;

                    % Junejo does not appear to consider gradient
                    % magnitudes.  I tried it that way.  Then I tried
                    % it with the magnitudes as a histogram scalar of
                    % gradient in a given cell as below.

                    col = gradients(t,2);

```

```
                    % Without gradient magnitude:
                    % hog(row,col) = hog(row,col) + 1;

                    % With gradient magnitude:
                    hog(row,col) = hog(row,col) + (1 * gradients(t,1));

                end % for s
            end % for r
        end % if (i == j)

    end % for j
end % for i

% fprintf('Found HOG.\n');


% Normalize the HOG
totals = sum(hog,2);

for i = 1:(numSamples*11)
    rowTotal = totals(i,1);
    for j = 1:numBins
        if (rowTotal > 0)
            hog(i,j) = hog(i,j)/rowTotal;
        end
    end
end

if (display == 1)
    [Hrows,Hcols] = size(hog);
    fprintf('HOG is (%d x %d).\n', ...
        Hrows, Hcols);
end


% Store HOG matrix in a .mat file
fileWriteCommand = sprintf('save %s hog -mat', outFileName);
eval(fileWriteCommand);
```

A.2.5.7 genMotSSM.m

```
% ----------------------------------------------------------------------
% Filename:  genMotSSM.m
% Author:    Paul Yanik
% Date:      August, 2010
%
% Use model:
% genMotSSM(motionType,sensorType,r,theta,phi,plotSSM)
%
% Description:
% This function generates a Self-Similarity Matrix (SSM) from IR motion
% sensor data.
%
% Reference:
% I. Junejo, E. Dexter, I. Laptev and P. Perez, "Cross-View Action
% Recognition from Temporal Self-Similarities", European Conference
% on Computer Vision -- ECCV 2008, 2008, pp. 293-306.
% ----------------------------------------------------------------------
function [] = genMotSSM(motionType,sensorType,inSituPercentage, ...
    r,theta,phi,version,plotSSM)

fprintf('genMotSSM(%3d percent): %6s sensor (%5s), (r,theta,phi) = (%2d,%3d,%3d), v %d, ', ...
    inSituPercentage, sensorType, motionType, r, theta, phi, version);

% Generate the fileName to read from input parameters.
inFileName = [motionType,sensorType,'_theta_',num2str(theta)];

% Generate the fileName to write from input parameters.
outFileName = [motionType,'_',sensorType,'_r',num2str(r),'t', ...
    num2str(theta),'p',num2str(phi),'_v',num2str(version)];

% Read in motion data from the appropriate file.
motionDataFile = dlmread([inFileName,'.txt']);
[rows,cols] = size(motionDataFile);

ssmParams;

% Number of samples to read from the input file.
numDataPts = floor((inSituPercentage/100) * numSamples);

% Read the entire motionData file into an array.
motionData = zeros(numSamples,1);

k = 0;
m = 0;
```

```
for i = 1:rows
    if (motionDataFile(i,phiCol) == phi)

        % Count total data points in the input file.
        m = m + 1;

        if (k < numDataPts)
            k = k + 1;
            motionData(k,1) = motionDataFile(i,s2Col);
        end

    end
end



GK = genGaussKernel(0.5);

motionData = smooth1D(motionData,GK);
motionData = smooth1D(motionData,GK);
motionData = smooth1D(motionData,GK);
motionData = smooth1D(motionData,GK);
motionData = smooth1D(motionData,GK);

% x = 1:1:numDataPts;
% plot(x,motionData,'r.-');
% hold on;

% motionData = accel(motionData);
% plot(x,motionData,'b.-');


% Check the integrity of the input file.
if (m ~= numSamples)
    fprintf ('BAD INPUT FILE \n');
    fprintf ('Found %d samples for (theta,phi) = (%d,%d). \n', ...
        k,theta,phi);
end
clear motionDataFile;

% Calculate the Euclidean Distance Matrix (EDM), D.
% This matrix compares all pairs of points across a single sequence of
% data (1 motion from 1 view over some numDataPts/sampleInterval);
D = zeros(floor(numDataPts/sampleInterval), ...
    floor(numDataPts/sampleInterval));

m = 0;
for i = sampleInterval:sampleInterval:numDataPts
    m = m + 1;
    % fprintf('i = %3d, m = %3d\n',i,m);
  n = 0;
    for j = sampleInterval:sampleInterval:numDataPts
      n = n + 1;
        D(m,n) = norm(motionData(i,1) - motionData(j,1));
    end
end
[Drows,Dcols] = size(D);
fprintf('D is %d x %d.\n',Drows,Dcols);

% Store D matrix in a .mat file.
fileWriteCommand = sprintf('save %s D -mat', ['SSM_',outFileName,'.mat']);
eval(fileWriteCommand);

% Redefine numDataPts to those actually used.
numDataPts = m;
clear motionData;

maxDval = max(max(D)); % Used to normalize D

if (plotSSM == 1)
    k = 0;
    x = zeros(numDataPts*numDataPts, 1);
    y = zeros(numDataPts*numDataPts, 1);
    color = zeros(numDataPts*numDataPts, 1);
    axis([1,numDataPts,1,numDataPts]);
    for i = 1:numDataPts
       for j = 1:numDataPts
           k = k + 1;
          m = D(i,j)/maxDval;
            x(k) = i;
            y(k) = numDataPts - j;
            color(k) = m;
       end
    end

    scatter(x,y,5,color,'filled');

```

```
    titleString = [sensorType,' sensor (',motionType,'), r = ', ...
        num2str(r),', theta = ',num2str(theta),' phi = ', num2str(phi)];

    title(titleString);

    print ('-djpeg', outFileName)

end % plotSSM

clear all;
```

A.2.5.8 genVidSSM.m

```
% ----------------------------------------------------------------------
% Filename:  genVideoSSM.m
% Author:    Paul Yanik
% Date:      November, 2010
%
% Description:
% Generate a Self-Similarity Matrix (SSM) from video data.
% Video data was converted from still image sequences to SSMs using C++.
% This function converts the text form of the SSMs to Matlab m files.
% The format of the text data is a coordinate pair (x,y) and the SSM value
% for that pair of matrix coordinates:  x y val (one data point per line).
% ----------------------------------------------------------------------
function [] = genVidSSM(motionType,sensorType,r,theta,phi,plotSSM)

fprintf('genVidSSM: %6s sensor (%5s), (r,theta,phi) = (%2d,%3d,%3d)\n', ...
    sensorType, motionType, r, theta, phi);

% Generate the input fileName to read using the input parameters.
inFileName = [motionType,'theta',num2str(theta),'phi',num2str(phi)];

% Generate the output fileName to write using the input parameters.
outFileName = [motionType,'_',sensorType,'_r',num2str(r),'t', ...
    num2str(theta),'p',num2str(phi)];

% Read in video data from the appropriate file.
dataFile = dlmread([inFileName,'.txt']);
[rows,cols] = size(dataFile);

% Find the largest x coordinate.
numSamples = max(dataFile(:,1)) + 1;

% Find the largest val (for normalization).
maxDval = max(dataFile(:,1))^2;

% Initialize the SSM.
D = zeros(numSamples,numSamples);

for i=1:rows
    x = dataFile(i,1) + 1;
    y = dataFile(i,2) + 1;
    val = dataFile(i,3);
    D(x,y) = val/maxDval;
end

% Store SSM matrix in a .mat file.
fileWriteCommand = sprintf('save %s D -mat', ['SSM_',outFileName,'.mat']);
eval(fileWriteCommand);
clear dataFile;

markerSize = 5;
if (plotSSM == 1)
    k = 0;
    x = zeros(numSamples*numSamples, 1);
    y = zeros(numSamples*numSamples, 1);
    color = zeros(numSamples*numSamples, 1);
    axis([1,numSamples,1,numSamples]);
    for i = 1:numSamples
       for j = 1:numSamples
           k = k + 1;
         m = D(i,j);
           x(k) = i;
           y(k) = numSamples - j;
           color(k) = m;
        % scatter(numSamples - i,j,3,m,'filled');
        % hold on;
       end
    end

    scatter(x,y,markerSize,color,'filled');

    titleString = [sensorType,' sensor (',motionType,'), r = ', ...
        num2str(r),', theta = ',num2str(theta),' phi = ', num2str(phi)];
```

```

    title(titleString);

    print ('-djpeg', outFileName)
end % plotSSM

clear all;
```

A.2.5.9 getCellBoundsHOG.m

```
% Cell angle boundary angles for HOG cells

cellBounds = zeros(5,2);

cellBounds(1,1) = 14*pi/8; cellBounds(1,2) = 15*pi/8; % 315.0 - 347.5
cellBounds(2,1) = 15*pi/8; cellBounds(2,2) =    pi/8; % 347.5 -  22.5
cellBounds(3,1) =    pi/8; cellBounds(3,2) =  3*pi/8; %  22.5 -  67.5
cellBounds(4,1) =  3*pi/8; cellBounds(4,2) =  5*pi/8; %  67.5 - 112.5
cellBounds(5,1) =  5*pi/8; cellBounds(5,2) =  6*pi/8; % 112.5 - 135.0
```

A.2.5.10 getCellNumHOG.m

```
% Find the cell that a point belongs in.

if (dist < 1)
    cellNum = 1;
elseif ((dist >=1) && (dist < 10))
    if  ((angle >= cellBounds(1,1)) && (angle < cellBounds(1,2)))
        cellNum = 2;
    elseif ...
        (((angle >= cellBounds(2,1)) && (angle < twoPi         )) || ...
         ((angle >= 0              ) && (angle < cellBounds(2,2))) )
        cellNum = 3;
    elseif ...
        ((angle >= cellBounds(3,1)) && (angle < cellBounds(3,2)))
        cellNum = 4;
    elseif ...
        ((angle >= cellBounds(4,1)) && (angle < cellBounds(4,2)))
        cellNum = 5;
    elseif ...
        ((angle >= cellBounds(5,1)) && (angle <= cellBounds(5,2)))
        cellNum = 6;
    else
        cellNum = 99;
        % fprintf('BAD ANGLE: %f, distance = %f \n',angle, dist);
    end

else % distance > 10
    if  (((angle >= cellBounds(1,1)) && (angle < twoPi          )) || ...
         ((angle >= 0              ) && (angle < cellBounds(1,2))))
        cellNum = 7;
    elseif ...
        (((angle >= cellBounds(2,1)) && (angle < twoPi         )) || ...
         ((angle >= 0              ) && (angle < cellBounds(2,2))) )
        cellNum = 8;
    elseif ...
        ((angle >= cellBounds(3,1)) && (angle < cellBounds(3,2)))
        cellNum = 9;
    elseif ...
        ((angle >= cellBounds(4,1)) && (angle < cellBounds(4,2)))
        cellNum = 10;
    elseif ...
        ((angle >= cellBounds(5,1)) && (angle <= cellBounds(5,2)))
        cellNum = 11;
    else ...
        cellNum = 99;
        % fprintf('BAD ANGLE: %f, distance = %f \n',angle, dist);
    end
end
```

A.2.5.11 HOGParams.m

```
% Radius of the virtual sphere.
r = 30;

% Find number of views to compare.
% thetaMin = 0;
% thetaInterval = 30;
% thetaMax = 180;
% phiMin = 0;
% phiInterval = 30;
```

```
% phiMax = 240;

thetaMin = 0;
thetaInterval = 15;
thetaMax = 180;
phiMin = 0;
phiInterval = 15;
phiMax = 255;


thetaVals = 0;
phiVals = 0;
for localTheta1 = thetaMin:thetaInterval:thetaMax
    thetaVals = thetaVals + 1;
end
for localPhi1 = phiMin:phiInterval:phiMax
    phiVals = phiVals + 1;
end

% Motion classes
numClasses = 3;
reach = 1;
press = 2;
grab = 3;

numViews = phiVals * thetaVals;
numPts = numClasses * phiVals * thetaVals;

% Typical parameters - may pass these in the future
numBins = 8;
d2r = pi/180;
```

A.2.5.12 ssmParams.m

```
% This file contains runtime parameters related to motion sensor data
% analysis.

% Number of samples at a given vantage point
numSamples = 350;

% Downsampling rate
sampleInterval = 3;

% In situ percentage.  This is the percentage of data points to use
% for calculation of a sub-SSM before moving the sensors.
% inSituPercentage = 1;

% Number of samples to read from the input file.
% numDataPts = floor(inSituPercentage * numSamples);

% Motion data file column labels
rCol = 1; % radius value column
thetaCol = 2; % theta value column
phiCol = 3; % phi value column
s1Col = 4; % sensor 1 output column
s2Col = 5; % sensor 2 output column
classCol = 6; % class output column
```

A.2.6 General Tools

A.2.6.1 fileNotFound.m

```
% This function prints an error when a file does not exist.


function[] = fileNotFound(fileName)


fprintf('ERROR: File %s not found\n', fileName);
```

A.2.6.2 genGaussKernel.m

```
% void buildGaussKernel(float &mu, float &sigma, vector<float> &GKernel)
% // void buildGaussKernel(float &sigma, vector<float> &GKernel)
% {
%   // float f = 2.5;
%   // float mu = round(f*sigma - 0.5);
%   float w = 2 * mu + 1;
%   float sum = 0;
```

```
%
%   for (int i=0; i<w; i++) {
%     GKernel.push_back( exp(-(i-mu)*(i-mu) / 2*sigma*sigma) );
%     // sum = GKernel[i] * i;
%     sum += GKernel[i];
%   }
%   for (int i=0; i<w; i++) {
%     GKernel[i] = GKernel[i]/sum;
%   }
%
%   int s = GKernel.size();
%   printf("Gaussian Kernel = [");
%   for (int i=0; i<s; i++) {
%     printf(" %5.3f",GKernel[i]);
%   }
%   printf("] \n");
% }

function[g] = genGaussKernel(sigma)

% sigma = 0.5, mu = 1, GK = [ 0.319 0.362 0.319]
% sigma = 1.0, mu = 2, GK = [ 0.054 0.244 0.403 0.244 0.054]
% sigma = 1.5, mu = 3, GK = [ 0.000 0.007 0.194 0.598 0.194 0.007 0.000]

f = 2.5;
mu = floor(f * sigma + 0.5);
w = 2 * mu + 1;

g = zeros(1,w);

sum = 0;

for i = 0:w-1
    g(1,i+1) = exp( -(i-mu)*(i-mu)/2*sigma*sigma );
    sum = sum + g(1,i+1);
end

g;

for i = 1:w
    g(1,i) = g(1,i)/sum;
end
```

A.2.6.3 mEq.m

```
% This function returns a single bit (0 or 1) if two 2d matrices
% are equal.  The two input matrices must be the same size.

function [equality] = mEq(a,b);

equalityMatrix = (a == b);

c = sum(sum(equalityMatrix));

equality = (c == prod(size(a)));
```

A.2.6.4 rand_in_range.m

```
% This function generates a single random floating point number within
% a specified range.

function[r] = rand_in_range(a, b)


r = a + (b-a).*rand(1,1);
```

A.2.6.5 smooth1D.m

```
% Filename:  smooth1D
% Description:
% This function smooths a 1D curve by an input kernel.

function [smoothedCurve] = smooth1D(inCurve, kernel)

[numDataPts,cols] = size(inCurve);
smoothedCurve = zeros(numDataPts,1);

[kernelRows,kernelCols] = size(kernel);

% Assume an odd kernel size.
limit = floor(kernelCols/2);
```

```

for i = 1:numDataPts
    sum = 0;
    e = 0;
    for j = (i - limit):(i + limit)
        e = e + 1;
        element = kernel(1,e);

        if (j < 1)
            sum = element * inCurve(-j + 2,1);
        elseif (j > numDataPts)
            sum = element * inCurve(numDataPts - (j - numDataPts));
        else sum = element * inCurve(j);
        end

        smoothedCurve(i,1) = smoothedCurve(i,1) + sum;

    end
end
```

A.2.6.6 vecSimilarity.m

```
% This function computes the cosine of an angle between two n-dimensional
% column vectors.

function[similarity] = vecSimilarity(vec1, vec2)


similarity = dot(vec1,vec2) / (norm(vec1) * norm(vec2));
```

Appendix B

IRB-Approved Consent Forms

Information about Being in a Research Study
Clemson University

Using Gestures to Communicate with "Smart" Furniture

Description of the Study and Your Part in It
Dr. Ian Walker, Paul Yanik, and their research team are inviting you to take part in a research study. The purpose of this study is to collect gesture motion data in order to see if a gesture-based human-machine interface can be developed as a way for humans to communicate with "smart furniture." This research is a component in larger interdisciplinary project examining an Assistive Robotic Table (ART) designed to enable individuals to age-in-place.

If you choose to participate in this study, you may complete some or all of the following:

- Provide consent to participate in the study.
- Answer questions about your age, gender, height, and weight.
- Watch videos of up to 6 different hand/arm gestures being performed in American Sign Language.
- Stand in front of a Microsoft Kinect and practice the gestures at least 2 times each.
- Stand in front of a Microsoft Kinect and perform the gestures at least 50 times each.
- You may take as many breaks as needed throughout the study.

It will take you a maximum of three hours to complete the study. You will be one of approximately 10 volunteers in the study.

Risks and Discomforts
We do not know of any risks or discomforts to you in this research study. You may take breaks at any time if you feel tired.

Possible Benefits
We do not know of any way you would benefit directly from taking part in this study. However, this research may help us to design furniture to help the aging population to age-in-place.

Protection of Privacy and Confidentiality
We will do everything we can to protect your privacy and confidentiality. We will not tell anyone outside of the research team that you were in this study or what information we collected about you in particular. While we will use the Microsoft Kinect in this research, the camera will be covered so that no one can identify you. We will only have a single-color image of your overall shape with a "stick figure" overlaid to shown the gross motions of your arms.

This form is valid only if the Clemson University IRB stamp of approval is shown here:

CLEMSON UNIVERSITY
IRB CONSENT FORM
APPROVED 4/15/2013
EXPIRES 8/24/2013

Page 1 of 2

(a)

Figure B.1: IRB constent form for gesture recognition experimentation phase 1, page (a) 1 of 2.

The data we collect will consist of the moving 3D coordinates of your joints as shown by the stick figure.

We might be required to share the information we collect with the Clemson University Office of Research Compliance and the Federal Office for Human Research Protections. If this happens, the information would only be used to find out if we ran this study properly and protected your rights in the study.

Choosing to be in the Study
You do not have to be in this study. You may choose not to take part and you may choose to stop taking part at any time. You will not be punished in any way if you decide not to be in the study or to stop taking part in the study.

If you choose to stop taking part in this study, the information you have already provided will be used in a confidential manner.

Contact Information
If you have any questions or concerns about this study or if any problems arise, please contact Dr. Walker at Clemson University at 864-656-7209.

If you have any questions or concerns about your rights in this research study, please contact the Clemson University Office of Research Compliance (ORC) at 864-656-6460 or irb@clemson.edu. If you are outside of the Upstate South Carolina area, please call the ORC's toll-free number, 866-297-3071.

Consent

I have read this form and have been allowed to ask any questions I might have. I agree to take part in this study.

Participant's signature: _____ Date: _____

Print signature: _____

A copy of this form will be given to you.

This form is valid only if the Clemson University IRB stamp of approval is shown here:

CLEMSON UNIVERSITY
IRB CONSENT FORM
APPROVED 4/15/2013
EXPIRES 8/24/2013

Page 2 of 2

(b)

Figure B.1: IRB constent form for gesture recognition experimentation phase 1, page (b) 2 of 2.

Bibliography

[1] R. Alami, A. Clodic, V. Montreuil, E. Sisbot, and R. Chatila, "Toward human-aware robot task planning," in *Proc. of the 2006 AAAI Symposia*. AAAI Press, 2006, pp. 39–46.

[2] T. Alexander, H. Ahmed, and G. Anagnostopoulos, "An open source framework for real-time, incremental, static and dynamic hand gesture learning and recognition," *Human-Computer Interaction. Novel Interaction Methods and Techniques*, pp. 123–130, 2009.

[3] J. Alon, V. Athitsos, Q. Yuan, and S. Sclaroff, "A unified framework for gesture recognition and spatiotemporal gesture segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 31, no. 9, pp. 1685–1699, Sept. 2009.

[4] P. Althaus, H. Ishiguro, T. Kanda, T. Miyashita, and H. Christensen, "Navigation for human-robot interaction tasks," in *Proc. of the 2004 IEEE International Conference on Robotics and Automation (ICRA'04)*, vol. 2. IEEE, 2004, pp. 1894–1900.

[5] A. Angelopoulou, A. Psarrou, J. Garcia-Rodriguez, and G. Gupta, "Tracking gestures using a probabilistic self-organising network," in *Proc. of the 2010 International Joint Conference on Neural Networks (IJCNN)*, July 2010, pp. 1–7.

[6] ASL Pro Website, http://www.aslpro.com/cgi-bin/aslpro/aslpro.cgi.

[7] G. Barreto, A. Araújo, and H. Ritter, "Self-organizing feature maps for modeling and control of robotic manipulators," *Journal of Intelligent & Robotic Systems*, vol. 36, no. 4, pp. 407–450, 2003.

[8] A. Bauer, D. Wollherr, and M. Buss, "Human-robot collaboration: A survey," *International Journal of Humanoid Robotics*, vol. 5, no. 1, pp. 47–66, 2008.

[9] S. Beach, R. Schulz, J. Downs, J. Matthews, B. Barron, and K. Seelman, "Disability, age, and informational privacy attitudes in quality of life technology

applications: Results from a national Web survey," *ACM Transactions on Accessible Computing (TACCESS)*, vol. 2, no. 1, pp. 1–21, 2009.

[10] C. BenAbdelkader, R. Cutler, and L. Davis, "Gait recognition using image self-similarity," *EURASIP Journal on Applied Signal Processing*, vol. 2004, pp. 572–585, 2004.

[11] M. Bennewitz, W. Burgard, and S. Thrun, "Adapting navigation strategies using motions patterns of people," in *Proc. of the 2003 IEEE International Conference on Robotics and Automation (ICRA'03)*, vol. 2. IEEE, 2003, pp. 2000–2005.

[12] S. Berman and H. Stern, "Sensors for gesture recognition systems," *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, vol. 42, no. 3, pp. 277–290, 2012.

[13] C. Bishop, *Pattern Recognition and Machine Intelligence*. Springer, 2006.

[14] B. Blumberg, M. Downie, Y. Ivanov, M. Berlin, M. Johnson, and B. Tomlinson, "Integrated learning for interactive synthetic characters," in *ACM Transactions on Graphics (TOG)*, vol. 21, no. 3. ACM, 2002, pp. 417–426.

[15] A. Bobick, "Movement, Activity and Action: The Role of Knowledge in the Perception of Motion," *Philosophical Transactions of the Royal Society B: Biological Sciences*, vol. 352, no. 1358, pp. 1257–1266, 1997.

[16] A. Bobick and J. Davis, "The recognition of human movement using temporal templates," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 23, no. 3, pp. 257–267, 2001.

[17] A. Bobick and A. Wilson, "A state-based approach to the representation and recognition of gesture," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 19, no. 12, pp. 1325–1337, Dec. 1997.

[18] J. Brooks, L. Smolentzov, A. DeArment, W. Logan, K. Green, I. Walker, J. Honchar, C. Guirl, R. Beeco, C. Blakeney, A. Boggs, C. Carroll, K. Duckworth, L. Goller, S. Ham, S. Healy, C. Heaps, C. Hayden, J. Manganelli, L. Mayweather, H. Mixon, K. Price, A. Reis, and P. Yanik, "Towards a 'smart' nightstand prototype: An examination of nightstand table contents and preferences," *Health Environments Research and Design*, vol. 4, no. 2, pp. 91–108, 2011.

[19] J. Brooks, I. Walker, K. Green, J. Manganelli, J. Merino, L. Smolentzov, T. Threatt, P. Yanik, S. Ficht, R. Kriener, M. Mossey, A. Mutlu, D. Salvi, G. Schafer, P. Srikanth, and P. Xu, "Robotic alternatives for bedside environments in healthcare," *International Journal of Systems Applications, Engineering and Development*, vol. 6, no. 3, pp. 308–316, 2012.

[20] J. Burke, R. Murphy, D. Riddle, and T. Fincannon, "Task performance metrics in human-robot interaction: Taking a systems approach," DTIC Document, Tech. Rep., 2004.

[21] E. Carmeli, H. Patish, and R. Coleman, "The Aging Hand," *The Journals of Gerontology Series A: Biological Sciences and Medical Sciences*, vol. 58, no. 2, p. M146, 2003.

[22] G. Carpenter and S. Grossberg, "The ART of adaptive pattern recognition by a self-organizing neural network," *Computer*, vol. 21, no. 3, pp. 77–88, 1988.

[23] H. Cheng, A. Chen, A. Razdan, and E. Buller, "Contactless gesture recognition system using proximity sensors," in *Proc. of the 2011 IEEE International Conference on Consumer Electronics (ICCE)*, Jan. 2011, pp. 149–150.

[24] D. Cook, M. Youngblood, E. Heierman, K. Gopalratnam, S. Rao, A. Litvin, and F. Khawaja, "MavHome: An Agent-Based Smart Home," in *Proc. of the IEEE International Conference on Pervasive Computing and Communications*. IEEE, 2003, pp. 521–524.

[25] K. Covinsky, R. Palmer, R. Fortinsky, S. Counsell, A. Stewart, D. Kresevic, C. Burant, and C. Landefeld, "Loss of Independence in Activities of Daily Living in Older Adults Hospitalized with Medical Illnesses: Increased Vulnerability with Age," *Journal of the American Geriatrics Society*, vol. 51, no. 4, pp. 451–458, 2003.

[26] J. Crandall and M. Goodrich, "Measuring the intelligence of a robot and its interface," in *Proc. of the Performance Metrics for Intelligent Systems Workshop*, 2003.

[27] R. Cutler and L. Davis, "Robust real-time periodic motion detection, analysis, and applications," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 8, pp. 781–796, 2002.

[28] N. Dalal, B. Triggs, I. Rhone-Alps, and F. Montbonnot, "Histograms of Oriented Gradients for Human Detection," in *Proc. of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2005)*, 2005, pp. 886–893.

[29] K. Dautenhahn, S. Woods, C. Kaouri, M. Walters, K. Koay, and I. Werry, "What is a robot companion - friend, assistant or butler?" in *Proc. of the 2005 IEEE/RJS International Conference on Intelligent Robots and Systems (IROS 2005)*. IEEE, 2005, pp. 1192–1197.

[30] G. Demeris, B. Hensel, M. Skubic, and M. Rantz, "Senior residents' perceived need of and preferences for "smart home" sensor technologies," *International Journal of Technology Assessment in Health Care*, pp. 120–124, 2008.

[31] B. DeRuyter and E. Pelgrim, "Ambient Assisted-Living Research in CareLab," *Interactions*, vol. XIV, no. 4, pp. 30–34, 2007.

[32] R. Dongseok, U. Dugan, P. Tanofsky, H. Do, S. Young, and K. Sungchul, "T-less: A novel touchless human-machine interface based on infrared proximity sensing," in *Proc. of the 2010 IEEE/RJS International Conference on Intelligent Robots and Systems (IROS)*, Oct. 2010, pp. 5220–5225.

[33] E. Estrada, *The Structure of Complex Networks: Theory and Applications.* Oxford, 2012.

[34] R. Fiebrink, P. Cook, and D. Trueman, "Human model evaluation in interactive supervised learning," in *Proc. of the 2011 Annual Conference on Human Factors in Computing Systems.* ACM, 2011, pp. 147–156.

[35] R. Fiebrink, D. Trueman, and P. Cook, "A metainstrument for interactive, on-the-fly machine learning," in *Proc. New Interfaces for Musical Expression*, 2009.

[36] T. Fong, I. Nourbakhsh, and K. Dautenhahn, "A survey of socially interactive robots," *Robotics and autonomous systems*, vol. 42, no. 3, pp. 143–166, 2003.

[37] K. Förster, S. Monteleone, A. Calatroni, D. Roggen, and G. Tröster, "Incremental kNN classifier exploiting correct-error teacher for activity recognition," in *Proc. of the 2010 Ninth International Conference on Machine Learning and Applications (ICMLA)*, Dec. 2010, pp. 445–450.

[38] A. Friedman, *The Adaptable House: Designing Homes for Change.* McGraw-Hill Professional, 2002.

[39] B. Fritzke, "A Growing Neural Gas Network Learns Topologies," *Advances in Neural Information Processing Systems 7*, vol. 7, pp. 625–632, 1995.

[40] B. Fritzke, "A self-organizing network that can follow non-stationary distributions," in *Proc. of the 1997 International Conference on Artificial Neural Networks (ICANN'97).* Springer, 1997, pp. 613–618.

[41] D. Frolova, H. Stern, and S. Berman, "Most probable longest common subsequence for recognition of gesture character input," *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics.*

[42] S. Furao and O. Hasegawa, "An incremental network for on-line unsupervised classification and topology learning," *Neural Networks*, vol. 19, no. 1, pp. 90–106, 2006.

[43] H. Gross, V. Stephan, and H. Boehme, "Sensory-based robot navigation using self-organizing networks and q-learning," in *Proc. of the 1996 World Congress on Neural Networks*. Lawrence Erlbaum, 1996, pp. 94–99.

[44] S. Grossberg, "Adaptive pattern classification and universal recoding: I. Parallel development and coding of neural feature detectors," *Biological Cybernetics*, vol. 23, no. 3, pp. 121–134, 1976.

[45] S. Grossberg, "Nonlinear neural networks: Principles, mechanisms, and architectures," *Neural Networks*, vol. 1, no. 1, pp. 17–61, 1988.

[46] Y. Gu, A. Lo, and I. Niemegeers, "A survey of indoor positioning systems for wireless personal networks," *IEEE Communications Surveys & Tutorials*, vol. 11, no. 1, pp. 13–32, 2009.

[47] M. Hackel, G. Wolfe, S. Bang, and J. Canfield, "Changes in hand function in the aging adult as determined by the Jebsen Test of Hand Function," *Physical Therapy*, vol. 72, no. 5, p. 373, 1992.

[48] F. Hamker, "Life-long learning Cell Structures-continuously learning without catastrophic interference," *Neural Networks*, vol. 14, no. 4, pp. 551–573, 2001.

[49] J. Hertzberg and R. Chatila, "AI reasoning methods for robotics," in *Springer Handbook of Robotics*, B. Siciliano and O. Khatib, Eds. Springer, 2008, pp. 207–223.

[50] S. Hoggar, *Mathematics of Digital Images: Creation, Compression, Restoration, Recognition*. Cambridge University Press, 2006.

[51] S. Intille, K. Larson, and E. Tapia, "Designing and Evaluating Technology for Independent Aging in the Home," in *International Conference on Aging, Disability and Independence*, 2003.

[52] S. Iwarsson and A. Stahl, "Accessibility, Usability and Universal Design - Positioning and Definition of Concepts Describing Person-Environment Relationships," *Disability & Rehabilitation*, vol. 25, no. 2, pp. 57–66, 2003.

[53] H. J., "Growing Neural Gas: Experiments with GNG, GNG with Utility and Supervised GNG," Master's thesis, Uppsala University, 2002.

[54] S. Jin, Y. Li, G. Lu, J. Luo, W. Chen, and X. Zheng, "Som-based hand gesture recognition for virtual interactions," in *Proc. of the 2011 IEEE International Symposium on VR Innovation (ISVRI)*, Mar. 2011, pp. 317–322.

[55] G. Johansson, "Visual perception of biological motion and a model for its analysis," *Perception and Psychophysics*, vol. 14, no. 2, pp. 201–211, 1973.

[56] I. Junejo, E. Dexter, I. Laptev, and P. Pérez, "Cross-View Action Recognition from Temporal Self-Similarities," *European Conference on Computer Vision–ECCV 2008*, pp. 293–306, 2008.

[57] A. Kapadia, I. Walker, D. Dawson, and E. Tatlicioglu, "A model-based sliding mode controller for extensible continuum robots," in *Proc. of the 9th WSEAS International Conference on Signal Processing, Robotics and Automation*. WSEAS, 2010, pp. 113–120.

[58] F. Kaplan, P. Oudeyer, E. Kubinyi, and A. Miklósi, "Robotic clicker training," *Robotics and Autonomous Systems*, vol. 38, no. 3, pp. 197–206, 2002.

[59] A. Karahoca and M. Nurullahoglu, "Human motion analysis and action recognition," in *Proc. of the 1st WSEAS International Conference on Multivariate Analysis and its Application in Science and Engineering*. World Scientific and Engineering Academy and Society (WSEAS), 2008, pp. 156–161.

[60] U. Kartoun, H. Stern, and Y. Edan, "A human-robot collaborative reinforcement learning algorithm," *Journal of Intelligent & Robotic Systems*, vol. 60, no. 2, pp. 217–239, 2010.

[61] C. Kidd, R. Orr, G. Abowd, C. Atkeson, I. Essa, B. MacIntyre, E. Mynatt, T. Starner, and W. Newstetter, "The Aware Home: A Living Laboratory for Ubiquitous Computing Research." Springer, 1999, pp. 191–198.

[62] D. Klein and M. Randić, "Resistance Distance," *Journal of Mathematical Chemistry*, vol. 12, pp. 81–95, 1993.

[63] W. Knox, "Learning from human-generated reward," Ph.D. dissertation, The University of Texas at Austin, 2012.

[64] W. Knox, I. Fasel, and P. Stone, "Design principles for creating human-shapable agents," in *AAAI Spring 2009 Symposium on Agents that Learn from Human Teachers*, 2009.

[65] W. Knox and P. Stone, "Tamer: Training an agent manually via evaluative reinforcement," in *Proc. of the 7th IEEE International Conference on Development and Learning (ICDL 2008)*. IEEE, 2008, pp. 292–297.

[66] T. Kohonen, "The self-organizing map," *Proc. of the IEEE*, vol. 78, no. 9, pp. 1464–1480, 1990.

[67] Y. Kuno, T. Murashima, N. Shimada, and Y. Shirai, "Interactive gesture interface for intelligent wheelchairs," in *Multimedia and Expo, 2000. ICME 2000. 2000 IEEE International Conference on*, vol. 2. IEEE, 2000, pp. 789–792.

[68] S. Lee, "Automatic gesture recognition for intelligent human-robot interaction," in *Proc. of the 2006 7th International Conference on Automatic Face and Gesture Recognition (FGR 2006)*. IEEE, 2006, pp. 645–650.

[69] J. Lementec and P. Bajcsy, "Recognition of arm gestures using multiple orientation sensors: gesture classification," in *Proc. of the 7th IEEE International Conference on Intelligent Transportation Systems, 2004*, Oct. 2004, pp. 965–970.

[70] D. Leyzberg, E. Avrunin, J. Liu, and B. Scassellati, "Robots that express emotion elicit better human teaching," in *Proc. of the 2011 6th ACM/IEEE International Conference on Human-Robot Interaction (HRI 2011)*. IEEE, 2011, pp. 347–354.

[71] T. Martinetz and K. Schulten, *A "Neural-Gas" Network Learns Topologies*, T. Kohonen, K. Makisara, O. Simula, and J. Kangas, Eds. Amsterdam, The Netherlands: Elsevier Science Publishers, 1991.

[72] T. Martinetz, H. Ritter, and K. Schulten, "Three-dimensional neural net for learning visuomotor coordination of a robot arm," *IEEE Transactions on Neural Networks*, vol. 1, no. 1, pp. 131–136, 1990.

[73] MATLAB, *Version 7.10.0 (R2010a)*. Natick, Massachusetts: The MathWorks Inc., 2010.

[74] J. Merino, A. Threatt, I. Walker, and K. Green, "Forward Kinematic Model for Continuum Robotic Surfaces," in *Proc. of the 2012 IEEE/RJS International Conference on Intelligent Robots and Systems (IROS 2012)*. IEEE, 2012.

[75] Microsoft Xbox 360 + Kinect Website, http://www.xbox.com/en-US/kinect.

[76] S. Mitra and T. Acharya, "Gesture recognition: A survey," *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, vol. 37, no. 3, pp. 311–324, 2007.

[77] M. Moni and A. Ali, "HMM based hand gesture recognition: A review on techniques and approaches," in *Proc. of the IEEE International Conference on Computer Science and Information Technology*, 2009, pp. 433–437.

[78] T. Mori, A. Fujii, M. Shimosaka, H. Noguchi, and T. Sato, "Typical behavior patterns extraction and anomaly detection algorithm based on accumulated home sensor data," in *Future Generation Communication and Networking (FGCN 2007)*, vol. 2. IEEE, 2007, pp. 12–18.

[79] H. Olafsdottir, V. Zatsiorsky, and M. Latash, "The effects of strength training on finger strength and hand dexterity in healthy elderly individuals," *Journal of Applied Physiology*, vol. 105, no. 4, p. 1166, 2008.

[80] D. Olsen and M. Goodrich, "Metrics for evaluating human-robot interactions," in *Proc. of PERMIS*, vol. 2003, 2003.

[81] J. Prasad and G. Nandi, "Clustering method evaluation for hidden markov model based real-time gesture recognition," in *Proc. of the International Conference on Advances in Recent Technologies in Communication and Computing, 2009 (ARTCom'09)*, Oct. 2009, pp. 419–423.

[82] W. Preiser and E. Ostroff, *Universal Design Handbook*. McGraw-Hill Professional, 2001.

[83] PrimeSense Website, http://www.primesense.com.

[84] C. Rao, A. Yilmaz, and M. Shah, "View-Invariant Representation and Recognition of Actions," *International Journal of Computer Vision*, vol. 50, no. 2, pp. 203–226, 2002.

[85] H. Ritter, T. Martinetz, and K. Schulten, "Topology-conserving maps for learning visuo-motor-coordination," *Neural networks*, vol. 2, no. 3, pp. 159–168, 1989.

[86] F. Rivera-Illingworth, V. Callaghan, and H. Hagras, "Towards the Detection of Temporal Behavioural Patterns Patterns in Intelligent Environments," in *Proc. of the 2nd IET International Conference on Intelligent Environments, 2006 (IE 06)*, vol. 1, 2006.

[87] ROS OpenNI Tracker Website, http://ros.org/wiki/openni_tracker.

[88] ROS Website, http://www.ros.org.

[89] T. Schlömer, B. Poppinga, N. Henze, and S. Boll, "Gesture recognition with a wii controller," in *Proc. of the 2nd International Conference on Tangible and Embedded Interaction*, 2008, pp. 11–14.

[90] A. Steinfeld, T. Fong, D. Kaber, M. Lewis, J. Scholtz, A. Schultz, and M. Goodrich, "Common metrics for human-robot interaction," in *Proc. of the 2006 1st ACM SIGCHI/SIGART conference on Human-robot interaction*. ACM, 2006, pp. 33–40.

[91] E. Stergiopoulou and N. Papamarkos, "A new technique for hand gesture recognition," in *Proc. of the International Conference on Image Processing*, Oct. 2006, pp. 2657–2660.

[92] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*. Cambridge Univ Press, 1998, vol. 1, no. 1.

[93] L. Takayama, W. Ju, and C. Nass, "Beyond dirty, dangerous and dull: what everyday people think robots should do," in *Proc. of the 3rd ACM/IEEE International Conference on Human Robot Interaction*. ACM, 2008, pp. 25–32.

[94] A. Thomaz and C. Breazeal, "Teachable robots: Understanding human teaching behavior to build more effective robot learners," *Artificial Intelligence*, vol. 172, no. 6-7, pp. 716–737, 2008.

[95] A. Thomaz, G. Hoffman, and C. Breazeal, "Experiments in socially guided machine learning: understanding how humans teach," in *Proc. of the 1st ACM SIGCHI/SIGART Conference on Human-Robot Interaction (HRI 2006)*. ACM, 2006, pp. 359–360.

[96] A. Threatt, K. Green, J. Brooks, J. Merino, I. Walker, and P. Yanik, "Design and Evaluation of an Artificial Sub-linguistic Language for Non-Humanoid, Assistive Robot Interaction," in *Proc. of the 2013 15th International Conference on Human-computer Interaction (HCI 2013)*, Las Vegas, NV, 2013.

[97] A. Threatt, J. Merino, K. Green, I. Walker, J. Brooks, S. Ficht, R. Kriener, M. Mossey, A. Mutlu, D. Salvi, G. Schafer, S. Pallavi, P. Xu, J. Manganelli, and P. Yanik, "A Vision of the Patient Room as an Architectural-Robotic Ecosystem," in *Proc. of the 2012 IEEE/RSJ International Conference on Robots and Systems (IROS 2012)*, Vila Moura, Algarve, Portugal, 2012.

[98] S. Thrun, "Toward a framework for human-robot interaction," *Human–Computer Interaction*, vol. 19, no. 1-2, pp. 9–24, 2004.

[99] C. Touzet, "Q-Learning for Robots," in *The Handbook of Brain Theory and Neural Networks*. MIT Press, 2003, pp. 934–937.

[100] C. Touzet, "Neural reinforcement learning for behaviour synthesis," *Robotics and Autonomous Systems*, vol. 22, no. 3, pp. 251–281, 1997.

[101] A. Tucker, *Applied combinatorics*, 6th ed. Wiley, 2007.

[102] A. Varkonyi-Koczy and B. Tusor, "Human–computer interaction for smart environment applications using fuzzy hand posture and gesture models," *IEEE Transactions on Instrumentation and Measurement*, vol. 60, no. 5, pp. 1505–1514, 2011.

[103] J. Wachs, U. Kartoun, H. Stern, and Y. Edan, "Real-time hand gesture telerobotic system using fuzzy c-means clustering," in *Proc. of the 5th Biannual World Automation Congress, 2002*, vol. 13. IEEE, 2002, pp. 403–409.

[104] I. Walker, J. Brooks, K. Green, J. Manganelli, L. Smolentzov, A. Threatt, P. Yanik, and J. Merino, "Interactive Robotic Environments in Healthcare," in *Workshop on Interactive Systems in Healthcare (Poster Paper)*, 2011.

[105] I. Walker and K. Green, "Architectural Robotics: Unpacking the Humanoid," in *Proc. of the ARCHIBOTS Workshop on Architectural Robotics (UBICOMP '09)*, Orlando, FL, 2009.

[106] J. Walter and K. Schulten, "Implementation of self-organizing neural networks for visuo-motor control of an industrial robot," *IEEE Transactions on Neural Networks*, vol. 4, no. 1, pp. 86–96, 1993.

[107] C. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3, pp. 279–292, 1992.

[108] D. Weinland, R. Ronfard, and E. Boyer, "A survey of vision-based methods for action representation, segmentation and recognition," *Computer Vision and Image Understanding*, vol. 115, no. 2, pp. 224–241, 2011.

[109] J. Wiener, R. Hanley, R. Clark, and J. Van Nostrand, "Measuring the Activities of Daily Living: Comparisons Across National Surveys," *The Journal of Gerontology*, vol. 45, no. 6, p. S229, 1990.

[110] A. Wilson and A. Bobick, "Realtime online adaptive gesture recognition," in *Proc. of the 15th International Conference on Pattern Recognition*, vol. 1. IEEE, 2000, pp. 270–275.

[111] D. Wilson and C. Atkeson, "Simultaneous Tracking and Activity Recognition (STAR) Using Many Anonymous Binary Sensors," in *The Third International Conference on Pervasive Computing*. Springer, 2005, pp. 62–79.

[112] J. Yamato, J. Ohya, and K. Ishii, "Recognizing human action in time-sequential images using hidden markov model," in *Proc. of the 1992 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'92)*, 1992, pp. 379–385.

[113] Y. Yamazaki, H. Vu, P. Le, Z. Liu, C. Fatichah, M. Dai, H. Oikawa, D. Masano, O. Thet, Y. Tang *et al.*, "Gesture recognition using combination of acceleration sensor and images for casual communication between robots and humans," in *Proc. of the 2010 IEEE Congress on Evolutionary Computation (CEC)*, July 2010, pp. 1–7.

[114] R. Yan, K. Tee, Y. Chua, and H. Tang, "A gesture recognition system using localist attractor networks for human-robot interaction," in *Proc. of the 2010 IEEE International Conference on Robotics and Biomimetics (ROBIO)*. IEEE, 2010, pp. 1217–1222.

[115] M. Yang and N. Ahuja, "Recognizing hand gesture using motion trajectories," in *Proc. of the 1999 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 1, June 1999, pp. 466–472.

[116] P. Yanik, J. Manganelli, J. Merino, A. Threatt, J. Brooks, K. Green, and I. Walker, "Use of Kinect Depth Data and Growing Neural Gas for Gesture Based Robot Control," in *Proc. of the 6th International Conference on Pervasive Computing Technologies for Healthcare (PervasiveHealth 2012)*, La Jolla, CA, 2012, pp. 283–290.

[117] P. Yanik, J. Manganelli, L. Smolentzov, J. Merino, I. Walker, J. Brooks, and K. Green, "Toward active sensor placement for activity recognition," in *Proc. of the 10th WSEAS International Conference on Signal Processing, Robotics and Automation (ISPRA '11)*, Cambridge, UK, 2011, pp. 231–236.

[118] P. Yanik, J. Merino, J. Manganelli, L. Smolentzov, I. Walker, J. Brooks, and K. Green, "Sensor placement for activity recognition: comparing video data with motion sensor data," *International Journal of Circuits, Systems and Signal Processing*, no. 5, pp. 279–286, 2011.

[119] P. Yanik, J. Merino, A. Threatt, J. Manganelli, J. Brooks, K. Green, and I. Walker, "A Gesture Learning Interface for Simulated Robot Path Shaping Using a Human Teacher," *IEEE Transactions on Systems Man and Cybernetics, Part A - Systems and Humans (under review)*, 2013.

[120] P. Yanik, I. Walker, J. Merino, K. Green, A. Threatt, J. Manganelli, and J. Brooks, "A Gesture Learning Interface for Assistive Robotics Using a Growing Neural Gas Approach," in *Proc. of the 2013 IUI Machine Learning Workshop (IUIMLW '13)(Poster Paper)*, Santa Monica, CA, 2013.

[121] S. Zhou, Q. Shan, F. Fei, W. Li, C. Kwong, P. Wu, B. Meng, C. Chan, and J. Liou, "Gesture recognition for interactive controllers using mems motion sensors," in *Proc. of the 4th IEEE International Conference on Nano/Micro Engineered and Molecular Systems, 2009 (NEMS 2009)*, Jan. 2009, pp. 935–940.

[122] C. Zhu and W. Sheng, "Wearable sensor-based hand gesture and daily activity recognition for robot-assisted living," *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, vol. 41, no. 3, pp. 569–573, May 2011.