A METHOD FOR GENERATING ROBOT CONTROL SYSTEMS

by

Russell C. Bishop

Submitted in partial fulfillment of the requirements

for the degree of

Master of Science in Engineering

in the

Electrical and Computer Engineering

Program

YOUNGSTOWN STATE UNIVERSITY

Youngstown, Ohio

August 2008

A METHOD FOR GENERATING ROBOT CONTROL SYSTEMS

Signature:

Russell C. Bishop Date

Approvals:

Dr. Jalal Jalali, Thesis Advisor Date

Dr. Philip Munro, Committee Member Date

Dr. Frank Li, Committee Member Date

Dr. Peter Kasvinsky, Dean of Graduate Studies and Research Date

ABSTRACT

This thesis presents a method of generating neural-network based control systems for walking robots. A genetic learning rule is combined with a physics simulation and scoring system in order to find appropriate weights for these networks. This approach produces highly robust neural-network control mechanisms that are capable of handling a wide variety of conditions, such as rough terrain and randomly varying robot proportions. In each of two test runs, the system was able to make the robot walk approximately 1.75 meters (5.8 body lengths) in the physics simulation, over very rough terrain, in 14 seconds of simulation-world time.

ACKNOWLEDGEMENTS

I would like to thank all of the faculty members in the Department of Electrical and Computer Engineering at YSU, for helping me to develop the skills necessary to solve complex engineering problems, and for all of the interesting things that they have shown me during the past several years. In addition, I would like to thank Dr. Russell Smith, the other contributors to the ODE project, and the open-source software community as a whole for providing so many of the tools and building blocks that helped to make this research possible. Finally, I would like to thank my friends and family, who helped and supported me throughout my graduate (and undergraduate) studies.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

This thesis describes a method for automatically generating complex control systems for walking robots. One of the most interesting research fields today is the development of robots that are able to perform complex and somewhat arbitrary actions with some degree of reliability. While robotics as a field of engineering has existed for quite some time now, and robots have been created which are capable of performing many tasks, it is still very difficult to create a robot which can effectively navigate complex terrain, or inside buildings. This is mostly due to the fact that the simple forms of mechanical movement, such as wheels, are only effective over a narrow range of conditions. A wheeled robot, for example, may be able to navigate a single floor of a building, or a landscaped outdoor area, but would normally be incapable of dealing with anything that its wheels cannot roll over, such as stairs, or rough terrain. For this reason, an effective walking-robot technology would be very useful.

Designing an effective walking robot is a difficult problem for two distinct reasons. First, it is actually quite challenging for engineers to design mechanical systems that exhibit anything close to the combination of speed, strength, size and weight that exist in biological organisms. This problem tends to either introduce severe limits on what can be done, or alternatively, cause the cost to construct a robot to be extremely high. Secondly, and somewhat relatedly, the control system for an effective walking robot is by necessity very complicated. This is because of the wide variety of conditions under which such a robot must be able to operate; a simple pre-programmed sequence of movements is not sufficient to provide reliable walking.

There are many different methods which have been used to provide intelligent control of walking robots. One approach is the use of Central Pattern Generators (CPGs), which have been used to control biped robots [1, 2]. Like the biological systems that inspired this method, a robot using CPG motion control has a very small neural network in which groups of individual perceptrons behave like schmidt trigger oscillators. The currently-active perceptrons inihibit the others until their responses to the input vector override the inhibition. At this point, when the system begins to switch states, a positive-feedback condition is created which strongly attracts the system into its next state. These neuronal oscillators can be connected in a purely feed-forward layout, in which the neurons use only each other's outputs as inputs, or they can use feedback, in which the inputs to the neurons are sensor outputs from the controlled system[3]. The behavior of this system is normally hard-coded, and tends to suffer from most of the same drawbacks as a pre-programmed gait — it requires a human programmer to consider each possible situation that it may encounter.

Genetic algorithms have also been used to develop control systems in walking robots. Luk, Galt and Chen [4] use a genetic algorithm to develop feed-forward walking patterns for an octopod robot, while Lewis, Fagg and Bekey [5] combine a genetic algorithm with a CPG to produce walking behavior in a hexapod robot.

In this thesis, a new method is developed which works in a similar way to [5], in that a neuronal oscillator controller is trained with a genetic learning rule, but with several key differences. First, the new method uses a relatively large neural network, of the type proposed by Auer, Burgsteiner and Maass [6]. The network used in this thesis has dozens to hundreds of perceptrons and, in some cases, upwards of a half-million weights (see test runs in Chapter 5). These perceptrons are not connected together directly as they are in the CPG, but do have feedback from the aggregate (system) output. In addition, the system has some internal memory which stores a certain number of past inputs and outputs. Thus, the control system can not only "see" the current state of the robot, but also remembers what has been happening with the physical robot and what it has been doing. The length of this memory is a user-entered variable, which has been set at 150 and 250 in the test runs performed for this thesis (see Chapter 5). Finally, the scoring and selection algorithms used in this thesis are based only on walking performance; the first training steps used in [5] to initially produce oscillatory behavior is not present.

For purposes of training the neural network, software is created which combines a physics simulation with a scoring algorithm. Candidate control systems are scored on how far they can make a simulated robot walk over randomly-generated terrain in a given amount of time, and this information is passed back to the genetic algorithm. After each neural network has had a turn, and received a score, the software ranks them and replaces the lower scorers with new networks that are created by combining pairs of high-scorers and applying random mutations. These steps are then repeated until the user decides that a sufficiently effective one has been produced, based on observation of the 3D-rendered simulation or the figures of merit introduced in Chapter 5, and terminates the program.

When the program is first started, all of the neural network weights are random and the simulated robots are only able to move a very short distance. As time progresses, however, the robots begin to develop the ability to produce continuous motion in one direction. In the test runs, the robots began to show some walking ability within about two days, and were becoming quite effective at walking after about a week.

While this method still requires some forethought on what types of situation the robot will encounter, in order to create effective training simulations, it does not need any hard-coding to be performed. All that is necessary is to create a 3D “world” with any terrain that the robot might have to navigate, as the software will randomly place robots in the world and score the control systems on how well they perform. In addition, the neural networks produced by this software are not limited to a single type of walking — multiple methods of movement have been observed in individual networks — which simplifies their integration into a complete robot.

This thesis is organized as follows: In Chapter 2, the neural network topology is described, as is the method for generating its input vector. There is a discussion on why it was chosen in section, and why it was expected to be effective, and its software implementation is described in detail. In Chapter 3, we discuss the genetic learning rule that is used with the neural network. The scoring rules that are used in the physics simulation are defined, as are the rules used for selection, crossover and mutation. Then, the software implementation of the genetic algorithm is described. In Chapter 4, the physics simulation in which the neural networks are trained is described, starting with the simulation “world”. Then, we discuss the quadruped robot body that is used in the simulations, its physics-engine implementation, and the geometrical parameters that describe individual robots. Finally, we describe the simulation loop in which the physics engine, the robot model, the neural network and the genetic algorithm come together. In Chapter 5, the performance of the software is evaluated. Figures of merit, collected from two test runs, are presented, and the results are discussed. In Chapter 6, we discuss our conclusions from this work, and propose some ideas for further research, as well as some potential applications.

2. NEURAL NETWORK

2.1 OVERVIEW

The neural network used in this project consists of a single layer of parallel perceptrons, similar to that described by Auer, Burgsteiner and Maass [6], but with an outboard genetic learning rule rather than the one described in that work. Each perceptron has a set of input weights that determines its response to a given set of inputs, an activation function which, in this thesis, is a unit-step function, and a set of output weights, which are multiplied by the output of the activation function (1 or 0) and added to the system output vector. This neural network operates in discrete time, evaluating sampled inputs and producing outputs at fixed time intervals. A block diagram of the neural network, and its associated memory stacks, is shown in Figure 2.1.

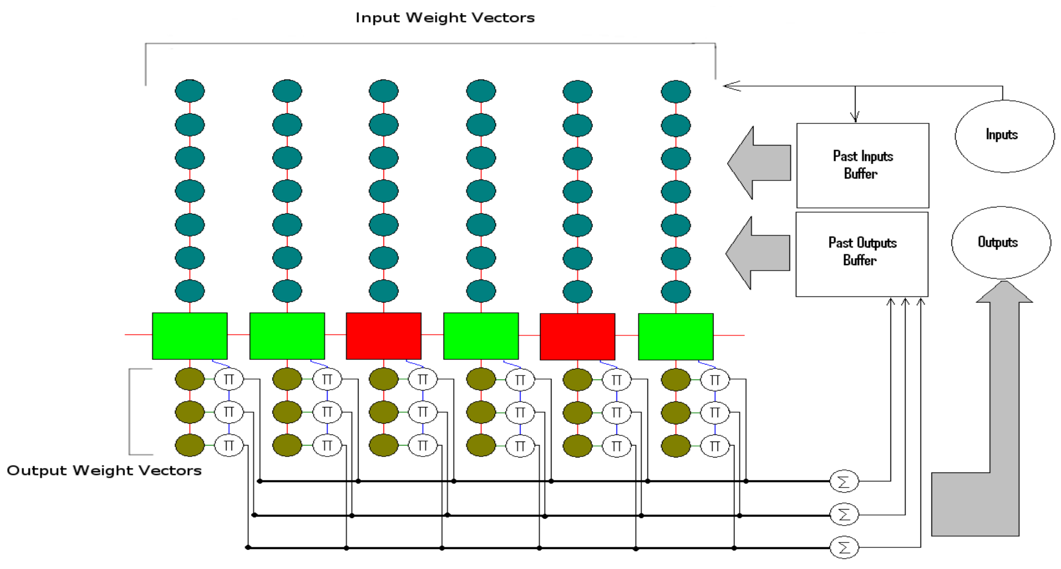

Figure 2.1: Neural network block diagram

2.2 INPUT VECTOR GENERATION

Inputs to the neural network come from three sources: body sensors, command and control signals, and previous inputs and system outputs. Past inputs and outputs come from a type of stack buffer where data travels down the stack and is discarded when it passes the last level. These historical data are used for two purposes: as inputs for the neural network, and as training data for a second learning rule that is implemented in the software, but not currently being used. The organization of this stack is shown in Figure 2.2.

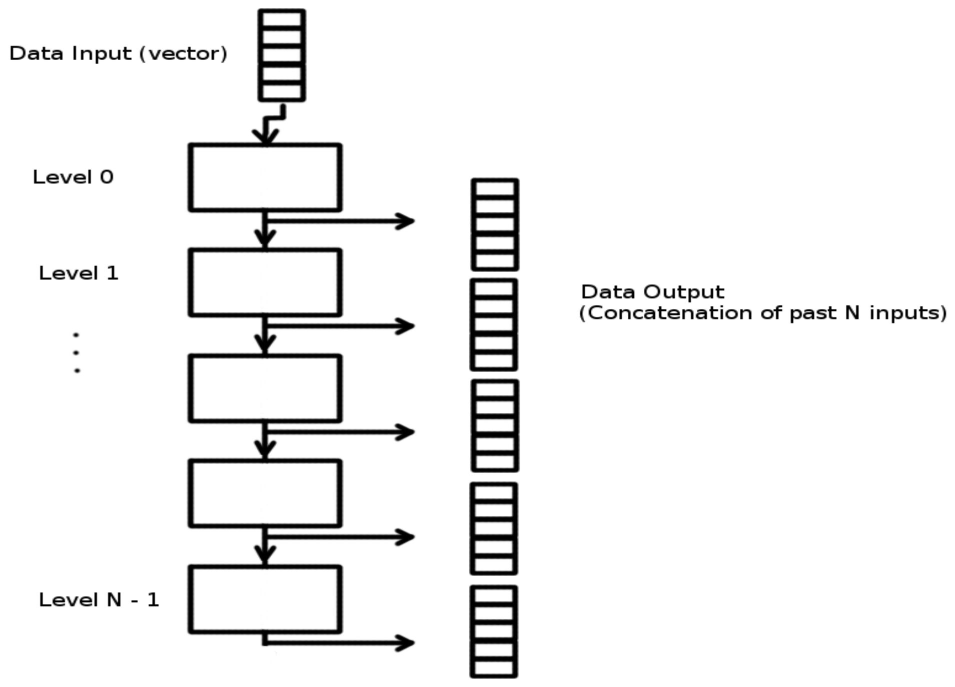

Figure 2.2: Block Diagram of History Buffer Object

2.3 OUTPUT VECTOR GENERATION

On each time step, the input vector to the neural network is generated by concatenating the body sensor and command inputs with the past inputs and outputs from the history buffer objects. This vector, I_sys, is multiplied (dot product) with each perceptron's input weight vector, W, to give the postsynaptic potential (PSP). The output of the perceptron is the unit step function of the PSP, multiplied piecewise by the perceptron's output weights to give its contribution, Rn, to the system output vector, Rs. This may be expressed as follows:

$$\mathbf{Rn = u(\, I\_sys \bullet W\,)} \tag{2.1}$$

$$\mathbf{Rs = \Sigma(\, Rn\,)} \tag{2.2}$$

2.4 WHY THIS ALGORITHM

At this point, some information is given regarding why this system can work. First, due to the fact that the number of perceptrons is much larger than the number of outputs, this algorithm is a universal function approximator[6]. This means that it can implement an arbitrary bounded function given the correct weights, even when the network has only a single hidden layer. Because the outputs of this neural network determine the rate of change in the actuator positions on the robot, the result is a system of nonlinear partial differential equations which, depending on the weight vectors and the physical properties of the robot, are capable of producing an extremely wide variety of behaviors (although not all behavior is technically possible, as there are physical limits on speed, force, and acceleration). Due to the way the data propagate through the history buffers, and thus constantly change position with respect to the input weights, it is relatively difficult for the system to reach a stable state where the robot does not move. Instead, this tends to encourage strange attractors, which produce repetitive, but not necessarily periodic, motion.

2.5 SOFTWARE IMPLEMENTATION

This neural network is implemented in C++ as the mcNeuron object class (in which the "mc" is short for "Motion Control"). It is organized in a linked list, where each instance represents one perceptron, and holds a pointer to the next perceptron in the chain. The advantage to this type of organization is that the source code can be kept short, as a large portion of the compiled machine code is automatically generated by the compiler itself. This also helps prevent errors by making the source code more readable, and relying on the very mature code-generation algorithms used in the compiler. The source code for this object class is given in Appendix A, and its member functions are described below:

- void rnNet(float* inputs, historyBuffer* iHistory, historyBuffer* oHistory, float* outputs)

  This function multiplies the input weights of the perceptron (dot product) by the concatenation of *inputs, iHistory,* and *oHistory,* and if the result is positive, add its output weights to *outputs.* If there are more perceptrons in the chain, as indicated by a non-null "next" pointer, then this function is called in the next node, with the same parameters. Thus, one call to the first perceptron in the chain propagates to all of them.

- void updateNet(float scale, historyBuffer* iHistory, historyBuffer* oHistory)

  This function implements a second learning rule, which is not used in this project. It was replaced by the genetic algorithm very early in development. When called, it multiplies *scale* by values from *iHistory* and *oHistory*, and adds this to its input weights. Like rnNet, it propagates through all perceptrons in the chain.

- void iW_preset(float * newWeights)

  This Function sets the input weights to the values stored in *newWeights.* This function is recursive, and if the perceptron has a non-null "next" pointer, will call the same function in the next perceptron. In this case, the pointer is advanced by the number of input weights, so that one large array can be used to set all of the input weights in a chain.

- void iW_preset_justOne(float * newWeights)

  This function is the same as iW_preset(), but is not recursive.

- void oW_preset_justOne(float * newWeights)

  This is the same as iW_preset_justOne(), but acts on the output weights instead of the input weights.

- mcNeuron *getNext()

  This function returns a pointer to the next perceptron in the chain, or NULL if a next node does not exist.

- mcNeuron *cutNth(int index)

  This function cuts the chain at the Nth node, and returns a pointer to the removed segment. It works by recursively propagating down the chain while decrementing *index,* until *index* = 1. When this condition is true, the node sets its “next” pointer to NULL, and returns the value that was in that pointer. The returned pointer propagates back up the chain as the CPU falls down through the call stack, until the first called node finally returns it to the calling function.

- void setNext(mcNeuron * newNext)

  This function sets the “next” pointer in the called node to *newNext.*

- void appendChain(mcNeuron * newSegment)

  This function appends the chain specified by *newSegment* to the end of the called chain. It works by recursively propagating down the chain until it is called on a node whose “next” pointer is null, and setting that pointer to *newSegment.*

- float *getIWeights()

  This function returns a pointer to the input weights for the called perceptron.

- float *getOWeights()

  This function returns a pointer to the output weights for the called perceptron.

- void setRandomOWeights(float maxValue)

This function sets the output weights of the perceptron to random numbers, varying from -*maxValue* to +*maxValue.* It is recursive, and operates on each node in the chain until a null “next” pointer is reached.

- void setRandomIWeights(float maxValue)

  This function is the same as setRandomOWeights(), but operates on the input weights.

- void setCascadingOWeights(float weight, int oIndex)

  This function sets the output weight specified by *oIndex* to *weight,* and sets all others to zero. If the “next” pointer is not null, it calls the same function on the next node, with the parameters set by the following two rules:

  - If *oIndex* is less than the number of output weights, increment oIndex.
  - If *oIndex* is equal to the number of output weights, then the next *oIndex* is zero, and the next weight is -*weight.*

  Note that this function is not called in the final build of the software.

- void shakeIptWeights(float maxValue)

  This function adds a random number, which varies from -*maxValue* to *maxValue*, to each of the input weights. It is recursive, and operates on all perceptrons in the chain. After the random values are added, the weight vector is normalized.

- void shakeOptWeights(float)

  This function is the same as shakeIptWeights(), but operates on the output weights.

- void mutateIptWeights(float maxValue)

  This function selects a random, continuous segment of the input weights and replaces them with random numbers, which vary from *–maxValue* to *maxValue*. It is not recursive (it operates on only one perceptron), and is called by the much more extensive mutation function in the genetic algorithm class.

- void mutateOptWeights(float)

  This is the same as mutateIptWeights(), but operates on the output weights.

- void svNet(ofstream * saveFile)

  This function saves the input and output weights of a perceptron to the fstream object pointed to by saveFile. It is recursive, so the entire network will be saved when it is called on the first element in the chain. Note that the fstream object has an internal index that counts up as data are saved, so the function can be called on multiple chains with one open file, and they will all be saved in order.

- void ldNet(ifstream * loadFile)

  This function loads the input and output weights stored in the fstream object pointed to by loadFile into the input and output weights. It is also recursive, and operates in the same way as svNet.

3. GENETIC ALGORITHM

3.1 INTRODUCTION

The neural network described in Chapter 2 is trained using an outboard genetic search algorithm, which operates on the entire network, rather than individual perceptrons. Each candidate neural network is given a turn to control a randomly generated robot in a physics simulation, and scored based on its effectiveness at making the robot walk. Like all genetic algorithms, this one combines randomness, selection, crossover, and mutation to search the space of all possible input and output weight vectors. Due to the extremely large search space, and the fact that there are large clusters of viable solutions (different types of walking) with fitness functions that tend to be somewhat continuous, this problem should be particularly well-matched to the properties of a genetic algorithm [7].

Selection is based on a floating-point score that is generated by evaluating the network's efficacy in controlling a simulated robot. In order to function, a genetic algorithm must find a region in the search space where there exists a score gradient before it can begin to function as a genetic algorithm; before this happens it implements only a random search. As a result, the search must happen upon a region with a fitness gradient, by chance. If these regions fill too small a portion of the total search space, it can take a very long time for the search to locate one of them. For this reason, points must initially be awarded for results that are not directly useful, but which are likely to be connected to a useful region by a “bridge” of scores that are high for their particular region[7].

3.2 SCORING

At the start of a turn, the software drops a robot into the “world” at a random position and begins stepping its neural network along with the physics engine. In order to reduce noise in the score due to a random bounce when the robot falls a short distance to the ground, and reduce the tendency for the system to waste time early on by simply making the robots lean forward, there is a delay of approximately two seconds in simulation time before the software records the robot's “start” position. At the end of the turn, the start position is subtracted from the ending position, and points are awarded according to the following five rules:

1. Score is awarded for any movement that occurs, regardless of direction. Early in the process, this causes the system to select the neural networks that cause the system to exhibit those attractors that produce constant motion. This causes oscillatory behavior to be learned early in the evolutionary process, and is what replaces the initial learning step used in [5], where fitness functions were assigned to per-leg oscillations.
2. The population member receives points a second time for movement in the desired direction, as determined by a dot product, but only if that number is positive — a negative score here is counted as zero. As a result, it is possible for an individual to receive up to two points per meter for moving in the correct direction.
3. A two-point penalty is assessed if the robot is upside-down at the end of the turn, which can occur quite easily due to the physical characteristics of this

particular robot design. The purpose of this penalty is to avoid behavior that emerged in some of the earliest tests, where the robot would roll forward, and then hop along upside-down by kicking its legs.

4. A user-configurable penalty is assigned each time the robot chassis comes into contact with the ground. There is a delay of approximately 1 second in simulation time after a ground impact is registered, before the counter can be incremented again. This prevents large penalties from accruing quickly if the chassis remains in contact with the ground for a period of time. From the test runs that have been performed, it was found that this penalty needs to be very small at the beginning. In the tests discussed in Chapter 5, a penalty of 0.05 was used. It may be effective to increase this penalty slowly after the system has learned to walk, but this has not yet been tested.
5. The population member retains half of the score it received in the previous generation, so that a single weak performance is not likely to “kill” a high-scoring neural network. While this last rule can sometimes prevent a more-fit individual from displacing a less-fit one, the effect quickly fades away when an individual performs poorly for two or more generations. It also is not typically enough to prevent displacement in the case of a very low, or negative, score. For this reason, several replacements still occur in most generations.

3.3 SELECTION

At the end of a generation, all members of the population are sorted by a ranking algorithm, so that those with the highest score appear in the earliest positions. In order to select each parent for the next generation, a random floating-point number in the range [0, 1] is generated, and squared, so that the new probability distribution will tend toward zero. This new number still falls within the same range, but has an average value of ¼ instead of ½ — thus selecting higher-scoring individuals more often than low-scoring ones. This number is then multiplied by the size of the population, cast to an integer, and used to index a neural network that will be the “parent” of a new population member. Note that the random number could also be raised to any other positive power, or another function could be used to provide a different probability distribution, although these options have not been investigated. A second method which has been tested is to instead multiply the square by the maximum score in the population, and then take the weakest member above that score, but it appears to be too aggressive for the small populations that are feasible on a current PC, and was found to cause problems with early convergence. This cause of this problem is that the highest score in a generation tends to be much higher than the average score, or even the average of the top 5 scores, as shown in Chapter 5. The top scoring population member thus tends to be chosen as a parent very often by this rule, which causes the diversity in the population to disappear rapidly, leading to the early convergence problems that were observed.

3.4 CROSSOVER AND MUTATION

After the two parent networks are selected, a new neural network is created by combining them. Each perceptron in the child is created by randomly selecting the perceptron at the same position from one of the parents, and occasionally introducing a random mutation. These mutations can take any of the forms outlined below:

- A random, continuous, segment of the perceptron's input weights is chosen, and replaced with a string of random numbers. This permits behavior to drift over time at the individual perceptron level.
- A perceptron's output weights are rotated, so that all of its effects are “mirrored” to the opposite side of the body (either side-side or front-back can occur). At the same time, the perceptron's response is time-delayed by a random amount by doing a circular shift on its input weights by an integer multiple of the number of inputs. The purpose of this mutation is to encourage symmetry in the robot's motion, and allow effective behavior that evolves in one leg to eventually propagate to the other legs.
- At the population-member level, the software randomly selects a continuous group of perceptrons, and moves them to a new position in the list. This has no direct effect, but makes it possible for a new child to be created with multiple perceptrons that originally occurred at the same position. For example, the child could contain four nodes that were all at position 25 in its grandparents.
- After the new perceptron is generated, all of its weights (both input and

output) are randomly adjusted by a small amount, and the input weights vector is normalized.

3.5 SOFTWARE IMPLEMENTATION

The genetic algorithm is implemented by the mcEVO object class, which manages the population, and two helper functions, rankNodes() and breedNets(), which perform the genetic operations.

The mcEVO class encapsulates the neural network and its associated history buffers in such a way that the entire population can be accessed through one pointer. It also stores the geometry for the randomly generated robots. The source code for this class is given in Appendix B, and its member functions are described below:

- mcEVO(int popSize, mcEVO * previous, dReal * geomMin, dReal * geomMax)

  This is a chain constructor which builds a population of *popSize.* It does not generate the neural networks (this is done in a separate call), but it does generate a random set of robot-body proportions for each element. The input variable *geomMin* should point to an array containing the lower limits for each body dimension, while *geomMax* should contain the upper limits. These parameters are described in detail in the simulation section of this thesis. *Previous* is used internally to this chain constructor, and should be set to NULL when it is called from outside.

- ~mcEVO()

  This destructor operates on the entire chain, deleting all nodes and any perceptron chains that were attached to them.

- mcEVO * getMax(mcEVO * curBest, float curMax)

  This function returns a pointer to the node in the chain with the highest score value. The input variables *curBest* and *curMax* are used internally as the function recurses through the chain; it should thus be called with *curBest* = NULL and *curMax* set to a large negative number (-10 is sufficient in this case).

- void setPrevious(mcEVO * newPrevious)

  This function sets the “previous” pointer for the called node to *newPrevious.*

- void setNext(mcEVO *)

  This function sets the “next” pointer for the called node to *newNext.*

- void detach()

  This function detaches the called node from the chain, calls previous->setNext(next) and next->setPrevious(previous), and sets its own previous and next pointers to NULL. Thus, the node is removed from the chain, and the chain is spliced back together.

- mcEVO *getNext()

  This function returns the value in the “next” pointer of the called node.

- mcEVO *getPrevious()

  This function returns the value in the “previous” pointer of the called node.

- mcEVO *getFirst()

  This recursive function can be called on any node in the chain. It calls previous->getfirst() until previous = NULL, then returns a pointer to that node.

- mcEVO *getLast()

  This function works in the same way as getFirst(), but recurses down the chain instead of up, and returns a pointer to the last node.

- float getScore()

  This function returns the score stored by the called node.

- mcEVO *getLastAbove(float minScore)

  This function recurses up the chain until it reaches a node whose score is higher than *minScore.* It then returns a pointer to that node. Note that this function is called on the last node in the chain (rather than the first), and is intended to be used after the ranking operation is complete. See the section on the rankNodes() helper function below.

- mcEVO *getNth(int N)

  This recursive function extracts a pointer to the Nth node in the chain. It works by calling itself on the next node in the chain, while decrementing N, until N = 0. It then returns a pointer to the node where this occurred.

- void insBefore(mcEVO * newNode)

  This function inserts the node pointed to by newNode into the position preceding the called node. It sets its own “previous” pointer to newNode, and calls setPrevious() and setNext() on the new node, and setNext() on the current previous node, so that the chain is still continuous in both directions.

- void dumpScores()

  This recursive debug function causes all nodes in the chain to send their scores to stdout.

- void dumpWeights()

  This debug function causes all nodes in the chain to send their weights to stdout. Note that there can be many millions of weights, which can cause problems depending on the terminal program from which the software is run.

- void setScore(float newScore)

  This function sets the score stored by the called node to *newScore.*

- dReal *getParams()

  This function returns a pointer to the robot-body geometry parameters stored by the node.

- void appendChain(mcEVO * newSegment)

  This function causes the chain starting at newSegment to be appended to the end of the chain holding the called node. It recurses down the chain until next = NULL, then sets next = newSegment and calls newSegment->setNext(this).

- int killLast(int numDeleted)

  This function deletes the last numDeleted nodes in the chain. It works by recursively calling itself on the next node until next = NULL, then returning numDeleted. As the CPU falls back up through the call stack, each recursion subtracts one from the returned number and returns that, thus counting down toward zero. When the return value is zero, the node calls delete next, and sets next = NULL. All nodes below this point are then deleted by the chain destructor, as described above.

- void svBrains(ofstream * saveFile)

  This recursive function saves all of the neural networks being managed by a mcEVO chain into *saveFile.* It works by calling svNet() on the mcNeuron chain pointed to by each node in the chain, and then calling itself on the next mcEVO node. Note that the fstream object class counts and records the current position within the file, which greatly simplifies this implementation.

- void ldBrains(ifstream * loadFile)

  This function works in a similar way to svBrains(), but loads the neural network weights from a file into all of the mcNeuron objects being managed by the called mcEVO chain.

- void mkBrains(int numPerceptrons, int RHL, int THL)

  This recursive function causes all nodes in the mcEVO chain to generate neural networks and history buffer lists using the chain constructor for the mcNeuron class. The neural networks thus created have *numPerceptrons* perceptrons, and both history buffers (one for input variables, and one for output variables) have RHL + THL nodes. Note that this function, in its current implementation, assumes that each neural network has 34 inputs and 16 outputs. This will change when the class is adapted away from this project for general-purpose use.

- void mkBrains_random(int numPerceptrons, int RHL, int THL, float * array)

  This function works in the same way as mkBrains, but fills the input and output weight arrays with random numbers rather than leaving the memory uninitialized. Array points to an array of type float that is large enough to hold all input and output weights, which was used internally in a different version of this function. It has not been removed, because that version has not yet been fully evaluated at the time of this writing. For the version of the function used in this thesis, *array* can be set to NULL.

- mcNeuron *getBrain()

  This function returns a pointer to the first node in the mcNeuron chain being managed by the called mcEVO node.

- historyBuffer *getIHist()

  This function returns a pointer to the first node in the input history buffer chain being managed by the called mcEVO node.

- historyBuffer *getOHist()

  This function returns a pointer to the first node in the output history buffer chain being managed by the called mcEVO node.

- void setIHist(historyBuffer *)

  This function sets the input history buffer chain to be used by the called node.

- void setOHist(historyBuffer *)

  This function sets the output history buffer chain to be used by the called node.

The core features of the genetic algorithm, including selection, crossover, and mutation, are implemented in two helper functions that are written to operate on a mcEVO chain. These functions are:

- rankNodes(mcEVO * target)

  This function performs a sorting operation on the mcEVO chain beginning at *target*. The nodes are ranked in order of descending score. Note that, after the ranking is complete, target is no longer the first node in the chain. However, the member function getFirst() can be called on target, and the first node will be returned.

- breedNets(mcEVO *thePopulation, int popSize, int nReplaced, dReal *pMin, dReal *pMax, int nNeurons, int RHL, int THL, float mutProb, float maxMut, float iRnd, float oRnd)

This function implements almost all of the actual genetic algorithm, and is called after rankNodes(). Its arguments are as follows:

- *thePopulation* is a pointer to the mcEVO chain on which the function will operate.
- *popSize* is the size of the population.
- *nReplaced* is the number of population members that be replaced with newly created candidates.
- *pMin* is a pointer to the array containing the lower limits for the robot body parameters (see sections 4.6 and 4.7, as well as Tables 4.1 and 4.2).
- *pMax* is a pointer to an array containing the upper limits for the robot body parameters.
- *nNeurons* is the number of perceptrons in each population member.
- *RHL* is the length of the history stack used by the neural networks as inputs.
- *THL* is the length of the history buffer used for an additional learning rule that is not used in this thesis, but is implemented in the mcNeuron class. Note that the total length of the stacks is equal to *RHL* + *THL*.
- mutProb is the probability that a mutation will occur in any given perceptron.
- *maxMut* is the maximum magnitude of the random numbers that a segment of a perceptron's input weights will be replaced with, when this type of mutation occurs (see section 3.4). The newly generated weights will thus

vary from -*maxMut* to *maxMut*. Note that this value should be chosen so that its average magnitude is approximately equal to the average magnitude in the input weight vector, so that the newly created weights do not swamp the other weights. Because the input weights vector is normalized, the value of *maxMut* used in this thesis is set to 2 * sqrt(1 / number_of_input_weights).

- *iRnd* is the maximum magnitude of the random numbers that are added to each input weight, after the perceptron is created and all mutations are applied, and before the input weight vector is normalized.
- *oRnd* is the maximum magnitude of the random numbers that are added to the output weights. Note that the output weights are never normalized.

4. SIMULATION ENVIRONMENT

4.1 OVERVIEW

The software in which the robot controllers are trained is based on a free and open-source rigid body physics engine called OpenDE or ODE [8], which is short for “Open Dynamics Engine”. This engine was orignally created by Russell Smith, and is currently being maintained and extended by a community of volunteers. It is distributed under two separate licenses — the GNU LGPL and a BSD-style license — such that a user can choose either of them. Thus, it may be used in free or commercial software, with very few restrictions. The most significant restriction in the BSD-style license is that the original work must be cited. This physics engine provides general-purpose simulation of articulated bodies, in addition to collision detection, and is primarily intended for use in video games. It has become popular enough in robot simulations, however, that there have been robot-simulation software packages[9] created and even a book[10] written about modeling robots in ODE.

4.2 SIMULATION WORLD

The simulation “world” consists of two parts — a randomly generated height map (the “ground”), and a randomly proportioned robot model. The height map is arranged on a 256 x 256 grid that spans 50 x 50 meters in simulation space. At each grid point, the height is set to a random number so that all heights fall within a 0.13m range.

The robot body is generated and inserted into the world by the spiderBody object class (see section 4.4). A majority of the code in this class, about 1500 lines, comprises the constructor function, which performs the following steps:

- Create the core body of the robot, which consists of three ODE primitives, set up its mass and inertia matrix, add its collision detection geometry, and insert it into the world.
- Repeat the previous step for the upper legs and lower legs.
- Calculate the starting positions / rotations for the legs, and move them to those locations.
- Attach the legs with the appropriate ODE joints (ball joints at the hips and hinge joints at the knees).
- Calculate the base / tip positions of the actuators, and call genActuator() on each one.

4.3 QUADRUPED ROBOT BODY

The robot body used in these simulations is shown in Figure 4.1. This robot has four legs, each with four degrees of freedom, for a total of 16 DoF. The linear servos controlling a single leg are shown in Figure 4.2; their effects are as follows:

1. Works with Actuator 2 to control the direction of the axis of the upper leg.
2. Works with Actuator 1 to control the direction of the axis of the upper leg.
3. Controls the rotation of the upper leg about its axis. The effect of this actuator is interdependent with Actuators 1 and 2.
4. Controls the bending angle of the knee joint.

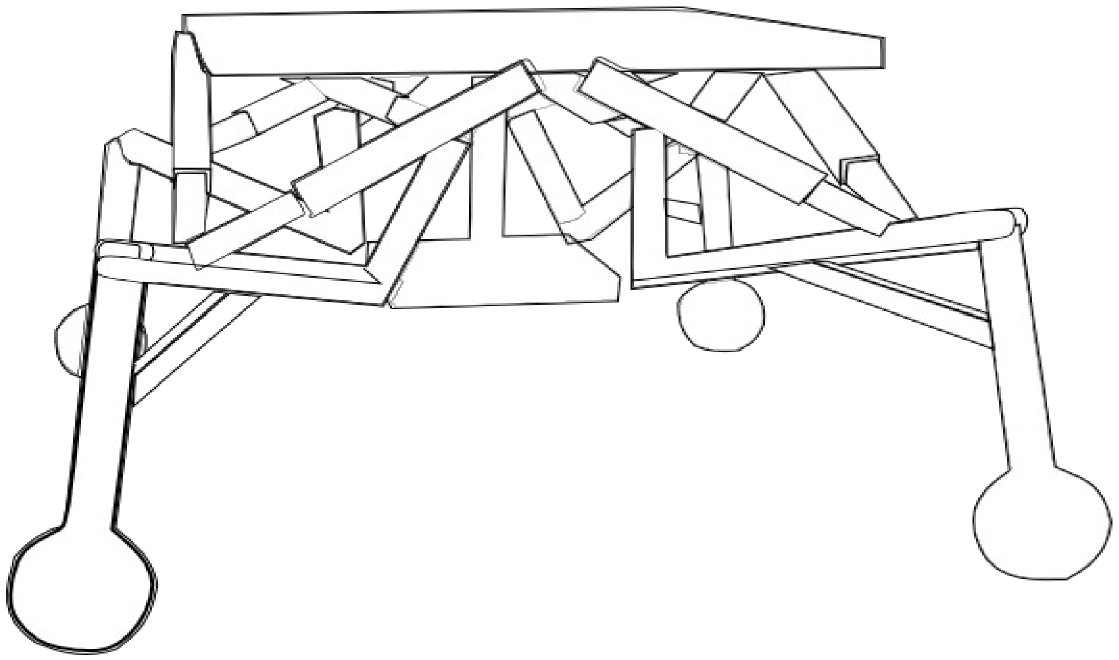

Figure 4.1: Quadruped Robot

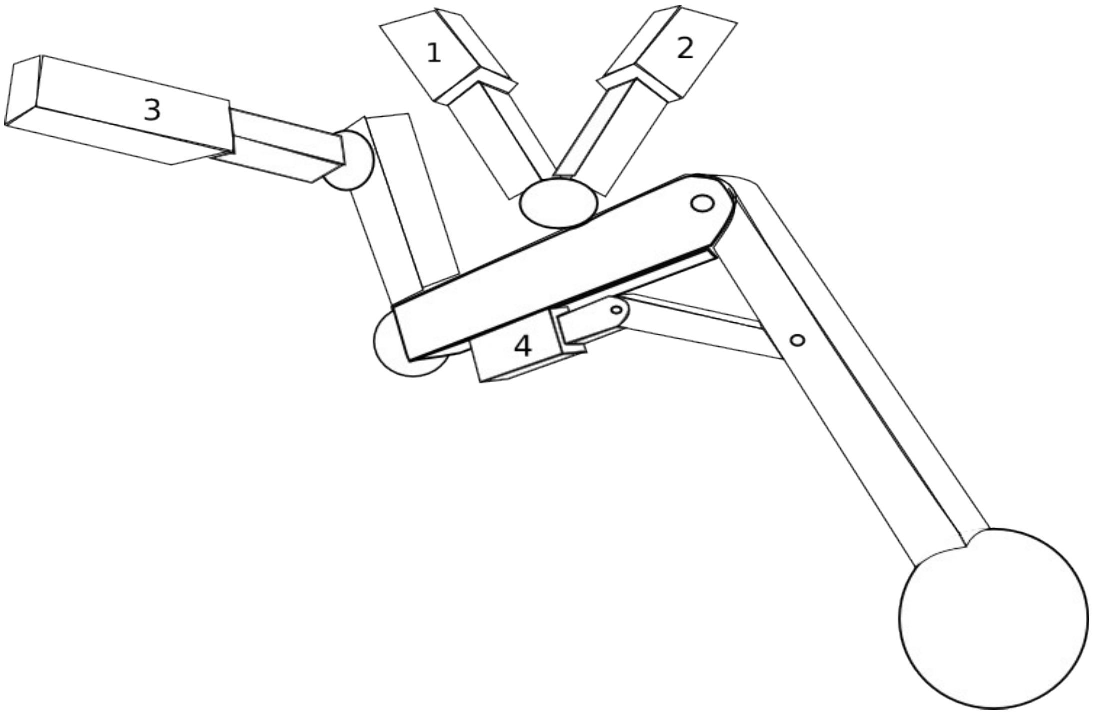

Figure 4.2: Diagram of a Single Leg Showing Actuator Indices

The major dimensions of the robot are shown in Figures 4.3, 4.4 and 4.5. These dimensions correspond to those shown in Table 4.1, and the upper and lower limits given in Table 4.2.

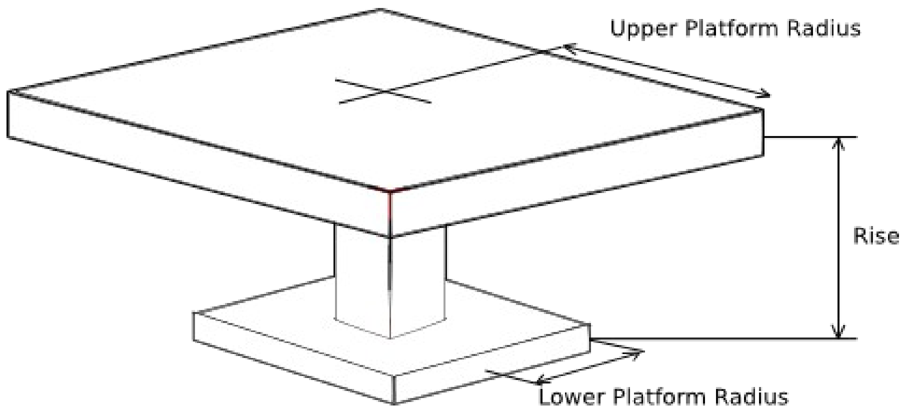

Figure 4.3: Robot Body Core (isometric view), Showing Dimensions

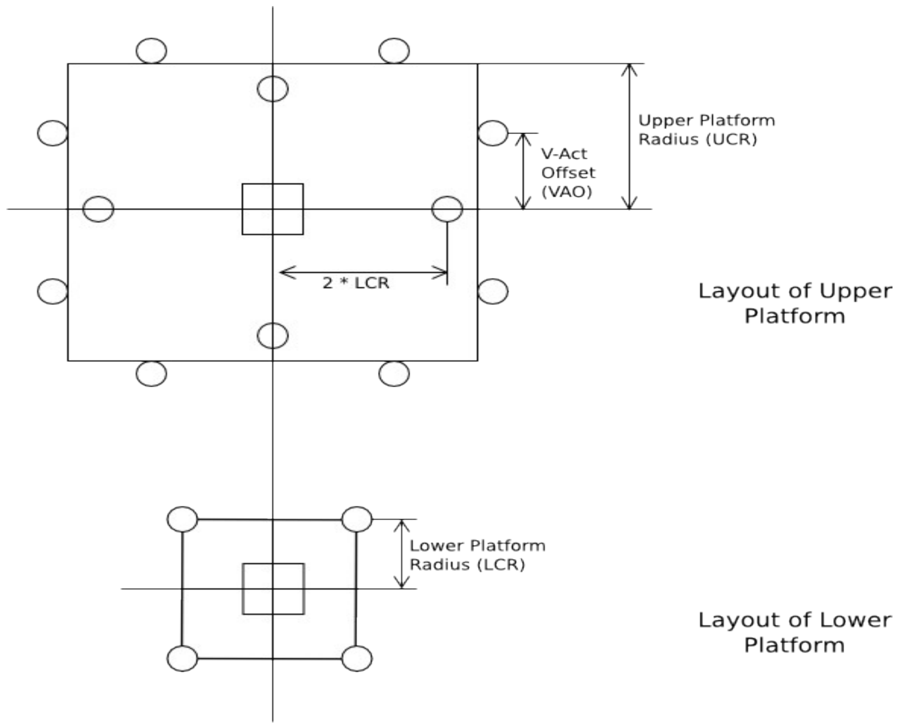

Figure 4.4: Diagram of Upper and Lower Chassis Platforms

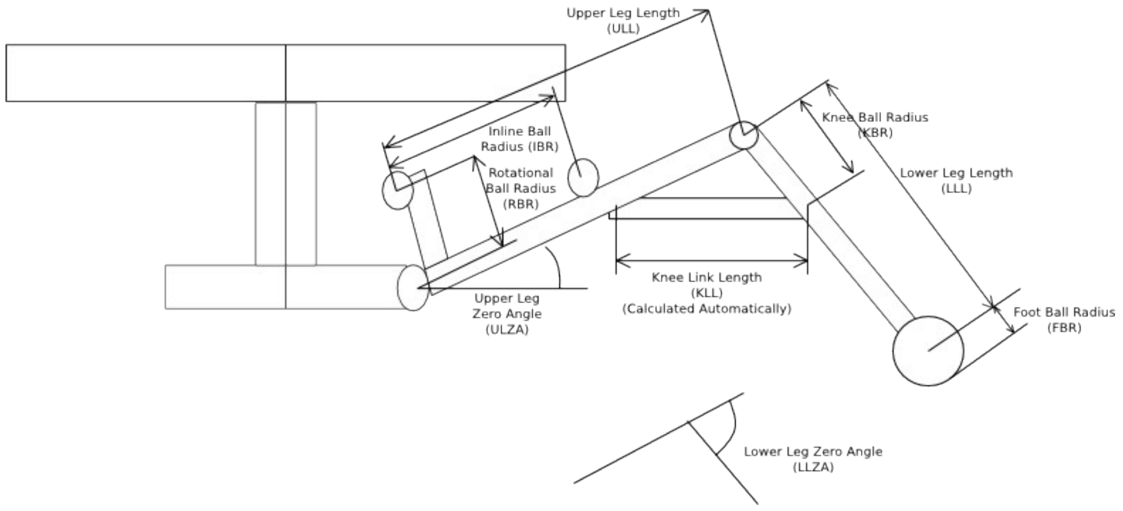

Figure 4.5: Diagram of a Leg, Showing Dimensions

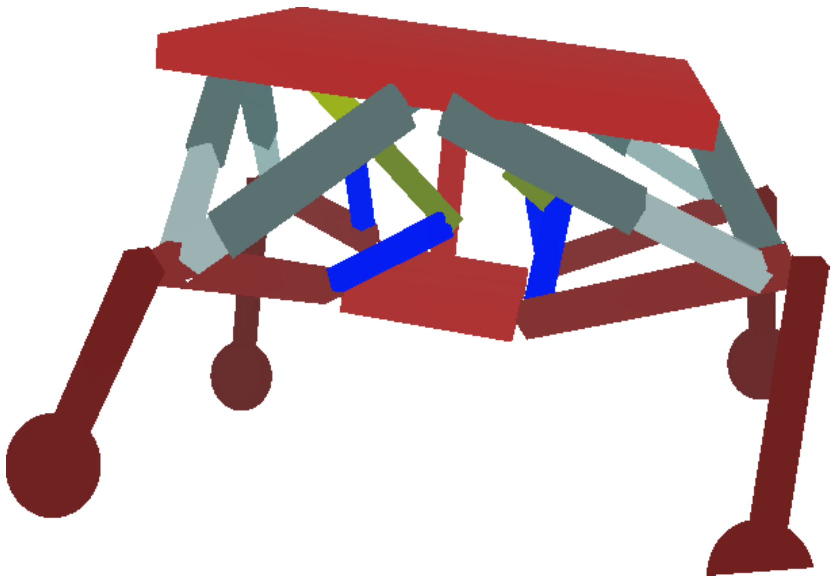

Figure 4.6: 3D Rendering of the Robot Walking in the Simulation Environment

Figure 4.6 shows a 3D-rendered example of the robot. This image was made from a screenshot of the robot walking in the simulation software. The gray actuators correspond to Actuators 1 and 2 in Figure 4.2. The yellow actuators correspond to Actuator 3, while Actuator 4 is not shown in this picture because it is handled outside ODE, in order to increase the speed of the software, and not drawn when the scene is rendered.

4.4 ROBOT BODY OBJECT CLASS

The ODE objects which model the robot body are created and manipulated through the spiderBody object class. The source code for this class is given in Appendix C. Aside from the constructor and destructor, the robot body class implements the following member functions:

- dReal getPos(int index)

  Returns the current length, in meters, of the linear actuator specified by *index,* with respect to its starting length. Negative numbers indicate that the actuator has retracted, while positive numbers indicate that it has extended.

- dReal getVel(int index)

  Returns the linear speed, in meters per second, of the actuator specified by *index*, where negative numbers indicate that the actuator is retracting and positive numbers indicate that it is extending.

- void addForce(int index, dReal force)

  Adds a 3rd law pair of forces of magnitude *force* to the two ends of the actuator specified by *index,* which are directed along its axis. This is the source of all of the driven motion in the physics simulation, except for the four knee joints.

- void addKneeTorque(int index, dReal torque)

  Adds a 3rd law pair of torques, of magnitude *torque,* to the upper and lower leg specified by *index.* This is the source of all driven motion at the knee joints.

- dReal getKneeAngle(int index)

  Returns the current angle, in radians, of the knee specified by *index.* This angle is measured from the direction of the upper leg (if the knee is straight, the angle is zero), and increases as the lower leg bends downward.

- dReal getKneeOmega(int index)

  Returns the current angular speed, in radians per second, of the knee specified by *index.*

- dBodyID getCore()

  Returns the ODE body ID of the robot chassis. This is used in the collision detection callback to count collisions between the chassis and ground (which incurs a small score penalty).

4.5 HELPER FUNCTIONS

In addition, there are three helper functions that are not members of the robot body class, but are used with it. All three of these functions relate to the actuator that drives each knee, but is external to the ODE world in order to increase processing speed. The source code for these helper functions is given in Appendix C, and they are described below:

- dReal calcKneeActOffset(dReal angle, dReal KBR, dReal KLL)

  Calculates the position of the knee actuator tip, in meters, with respect to the knee joint. This position ranges from zero to the length of the upper leg. Angle specifies the angle of the knee joint, in radians, as returned by spiderBody::getKneeAngle(int), KBR is the distance between the knee joint and the link attachment point on the lower leg, and KLL is the length of the linkage itself.

- dReal calcKneeTorque(dReal Angle, dReal slidePos, dReal KBR, dReal F)

  Returns the torque applied to the knee joint by a force F in the knee actuator. The input variable, *slidePos,* specifies the position of the knee actuator, as defined above, while *F* is the linear force in the actuator. *Angle* and *KBR* are the same variables described above.

- dReal calcKneeActVel(dReal Angle, dReal slidePos, dReal KBR, dReal w)

  Returns the linear speed of the knee actuator, in meters per second, given the angular speed of the knee joint, in radians per second. The input variable *w* is the angular speed; other inputs are the same as described above.

4.6 BODY GEOMETRY PARAMETERS

The body parameters, which are set at random by the software and passed to the robot body constructor in a parameter array are listed in Table 4.1. These parameters correspond to the dimensions in Figures 4.3, 4.4 and 4.5. The Index column specifies the position in the array, while the Macro column gives the three- or four-letter macro by which the variables are referenced in the source code (see section 4.4 and Appendix C). Note that all linear dimensions are in meters, while all mass parameters are in kilograms.

Table 4.1: Robot Body Parameters Array

Index	Variable	Macro
0	Upper platform (chassis) radius	UCR
1	V actuator upper mount offset (from centers of UP)	VAO
2	Distance between upper and lower platforms	RISE
3	Lower platform radius	LCR
4	Upper leg length	ULL
5	Lower Leg Length	LLL
6	Distance hip -> V ball on upper leg	IBR
7	Hip rotation linkage length	RBR
8	Knee link length *(Obsolete; now set automatically)*	KLL
9	Distance knee -> knee link attachment	KBR
10	Upper platform mass	UPM
11	Lower platform mass	LPM
12	Square tubing density (mass / unit length)	LINDENS
13	Platform and Leg thickness	THICK
14	Starting Position X	POSX
15	Starting Position Y	POSY
16	Starting Position Z	POSZ
17	Upper leg zero angle	ULZA
18	Leg rotation zero angle	LRZA
19	Lower leg zero angle	LLZA
20	Foot ball radius	FBR
21	Foot ball mass	FBM
22	V Actuator base mass	VABM
23	V Actuator tip mass	VATM
24	Rotational Actuator base mass	RABM
25	Rotational Actuator tip mass	RATM
26	Upper leg mass	ULM

4.7 BODY PARAMETER LIMITS

These body-geometry parameters listed in Table 4.1 vary randomly within a set of upper and lower limits defined by two limit arrays. The purpose of this variation is to train the neural networks to control a range of robots, rather than just a single example, to increase their resistance to the effects of small changes when going from the simulated robots to a physical one. The values used in the lower and upper limit arrays are given in Table 4.2.

Table 4.2: Upper and Lower Robot Parameter Limits

Index	Macro	Variable Description	Lower Limit	Upper Limit
0	UCR	Upper Platform Radius	0.22	0.27
1	VAO	V-Actuator Offset	0.018	0.022
2	RISE	Distance between upper / lower platforms	0.18	0.22
3	LCR	Lower Platform Radius	0.085	0.12
4	ULL	Upper Leg Length	0.27	0.32
5	LLL	Lower Leg Length	0.22	0.27
6	IBR	Inline Ball Radius	0.22	0.27
7	RBR	Rotational Ball Radius	0.14	0.15
8	KLL	Knee Link Length (OBSOLETE)	0.18	0.22
9	KBR	Distance between knee and link attachment	0.09	0.11
10	UPM	Upper Platform Mass	1.8	2.2
11	LPM	Lower Platform Mass	0.9	1.1
12	LINDENS	Linear Density of Square Tubing	0.18	0.22
13	THICK	Thickness of Square Tubing	0.025	0.028
14	POSX	Starting X Position	-5.00	5.0
15	POSY	Starting Y Position	-5.00	5.0
16	POSZ	Starting Z Position	0.39	0.4
17	ULZA	Upper Leg Zero Angle	0.25	0.3
18	LRZA	Leg Rotation Zero Angle	0.37	0.42
19	LLZA	Lower Leg Zero Angle	1.3	1.7
20	FBR	Foot Ball Radius	0.035	0.055
21	FBM	Foot Ball Mass	0.17	0.22
22	VABM	V-Actuator Base Mass	0.4	0.52
23	VATM	V-Actuator Tip Mass	0.09	0.12
24	RABM	Rotational Actuator Base Mass	0.38	0.42
25	RATM	Rotational Actuator Tip Mass	0.077	0.1
26	ULM	Upper Leg Mass	0.46	0.52

4.8 SIMULATION LOOP

On each step through the simulation loop, the inputs to the control system are updated with the force and position values for all of the actuators. The position values for the 12 upper leg actuators are obtained from ODE, using the getPos() member function of the robot body class, while the motion speeds for these actuators are obtained using getVel(). The knee actuator positions and speeds are calculated from the knee angles and angular velocities, which are obtained from ODE using the getKneeAngle() and getKneeOmega().

For all actuators, including the ones for the knees which are handled externally to ODE, the position is zero as seen by its control-system input at whatever position the actuators are created in. These zero positions are also used to define the actuator position variables which are modified by the outputs of the control system. The difference between these "set" position variables, and those returned by ODE, or calculated from angular values, in the case of the knees, are used to calculate the force in each actuator using a simple damped-spring equation:

F = -ks * (actual position – set position) – kd * (actuator speed)

where ks is a spring constant, and kd is a damping coefficient.

The spring constant for knee actuators is 1500N/m; for other actuators it is 1100N/m, and the damping coefficient is 30N*s/m. These values are based on measurements taken from a prototype linear actuator.

The calculated forces for all actuators except those in the knees are sent back to ODE through the robot body class using the addForce(index, force) member function, as well as to the control system as force-sensor inputs. The forces for the knees are converted to torque values, and sent to ODE using the addKneeTorque(index, torque) member function.

The actuator set positions are produced by the control system outputs through a double integral. The control system is able to set acceleration values for the actuators, up to a certain maximum acceleration, and these values change the speed of the actuators (the rate of change of the set value), up to a certain maximum. The maximum acceleration is set to be 2.9m/s^2 and the maximum speed is 0.35m/s, both of which are based on measurements taken from a prototype actuator.

In addition to position and force measurements, the control system also has two other inputs that describe the desired direction of travel with respect to the robot. These two values are dot products of a unit vector pointing in the desired direction with the robot's local X and Y vectors. These are treated exactly the same as the sensor inputs, and propagate through the history stack in the same way.

5. PERFORMANCE EVALUATION

5.1 OVERVIEW AND QUALITATIVE ANALYSIS

For a system such as this, the most definitive performance criterion is whether the robots begin walking in an effective way within a reasonable amount of time, while operating on a computer which is economically feasible to the user. During and after the development of this software, many test runs were performed, using an Intel E4300 CPU, a very inexpensive processor used in consumer PCs. In eac test, the AI always either learned to walk, or found a way to work around the rules and “cheat”, within a few days.

In the earliest runs, there was no penalty for being upside-down, which resulted in the robots' bouncing and rolling forward as far as they could upon dropping into the world, then kicking their legs and hopping forward while upside-down. Some of them also managed to tilt 90 degrees to the side and roll a good distance, effectively doing cartwheels, before falling down. When the penalty was added and the software re-run, a population of robots was produced fairly quickly that would hop forward, like frogs. At this point, a bug in the physics simulation code was found and fixed, and the first population of actual walkers was produced on the following run. For this test, the software was allowed to run for a period of approximately three weeks in real-time, in which time the it became very good at making the robots walk—at the end of this run, the robots were moving about 16 body lengths in 14 seconds of simulation time, which is quite fast given the physical characteristics of the robot and the limits that were in place on how fast the actuators were allowed to move and accelerate (see Chapter 4).

5.2 QUANTITATIVE ANALYSIS

In order to obtain a quantitative analysis of the performance of this system, a pair of test runs was done, with different parameters for the neural network. A special version of the software was created for these runs, which has the added feature of creating the log files that are used in the analyses below. These log files are formatted as plain text, with one line for each population member evaluated. The entries on each line are as follows:

- The index of the current population member. This ranges from 0 – 39, as a population size of 40 was used for all of the runs that used a log file.
- The score that the population member retained from the last generation, according to scoring rule #5 (see section 3.2).
- The number of times the chassis came into contact with the ground, as described in rule #4.
- The score given for any movement at all, as described in rule #1.
- The movement of the robot in the X direction.
- The movment of the robot in the Y direction.
- The final score passed back to the mcEVO node.

Results from two of these logged runs are included in this section. In these runs, each neural network is given a turn of 2000 time steps in which to control its robot. The starting positions are recorded after a delay of 250 time steps, which gives an effective turn length of 1750 time steps. Each time step for the neural network represents 0.012 seconds of simulation time, so there is a period of approximately 21 seconds in simulation time for which movement is recorded. Both tests are identical in all respects, except that one uses a neural network of 30 perceptrons, with a memory of 250 time-steps while the other uses 150 perceptrons, with a memory of 150 time-steps. Note that 250 time-steps is equivalent to approximately 3 seconds of simulation time, while 150 time-steps is equivalent to about 1.8 seconds. For these runs, the desired direction is always along the X axis, and the ground impact penalty is very small (0.05). Changes to these rules can be implemented slowly through a modification to the software — the desired direction will take random values that slowly drift away from the X axis, while the ground-impact penalty will slowly increase. This is not done here due to the length of time the software has to run before a new adaptation is made.

The results from the log files were post-processed using a second program, which was written to parse the data from the logs and extract the following data sets for each generation:

- The maximum score attained by any population member during the generation, excluding any score carried over from the previous generations.
- The top 5 scores from the generation.
- The average value of the top five scores from the generation.
- The maximum score ever achieved, in the current or any previous generation.

- The total movement in the X and Y directions for the top 5 scorers in the generation.

Figure 5.1 shows the top score results vs. generation from the 30-perceptron test. There are three data sets on this plot: the top score attained during the generation (orange), the average of the top five scores (purple), and the running maximum score (black). These scores are a figure of merit which represents the performance of the neural networks with respect to all of the scoring rules that are discussed in Chapter 3. A plot of the total movement in the X direction (orange) and the Y direction (purple) for the top scoring neural network in each generation is given in Figure 5.2. Unlike the scores shown in Figure 5.1, these movement figures provide concrete values that are relevant outside the context of the genetic algorithm — they represent the actual distance that the simulated robots were able to walk during the time allotted.

Figures 5.3 and 5.4 are the same plots as those in 5.1 and 5.2, respectively, but are taken from the 150-perceptron run. They show data taken from a smaller number of generations, but the same amount of real-world run time. This is because the software runs more slowly when a larger neural network is used.

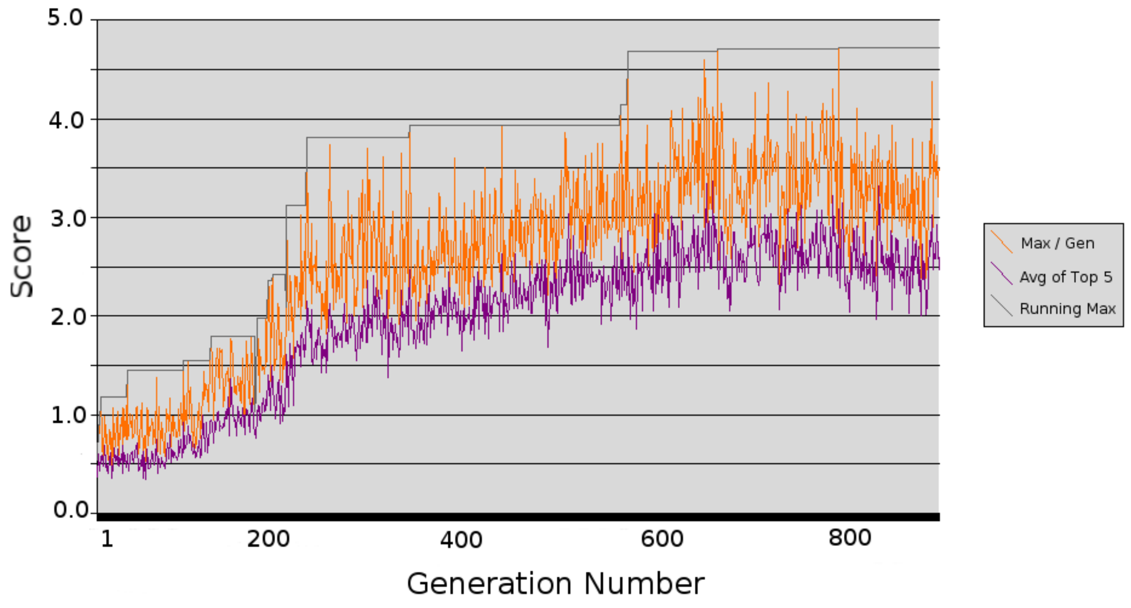

Figure 5.1: Scores Per-Generation for the 30-Perceptron Run

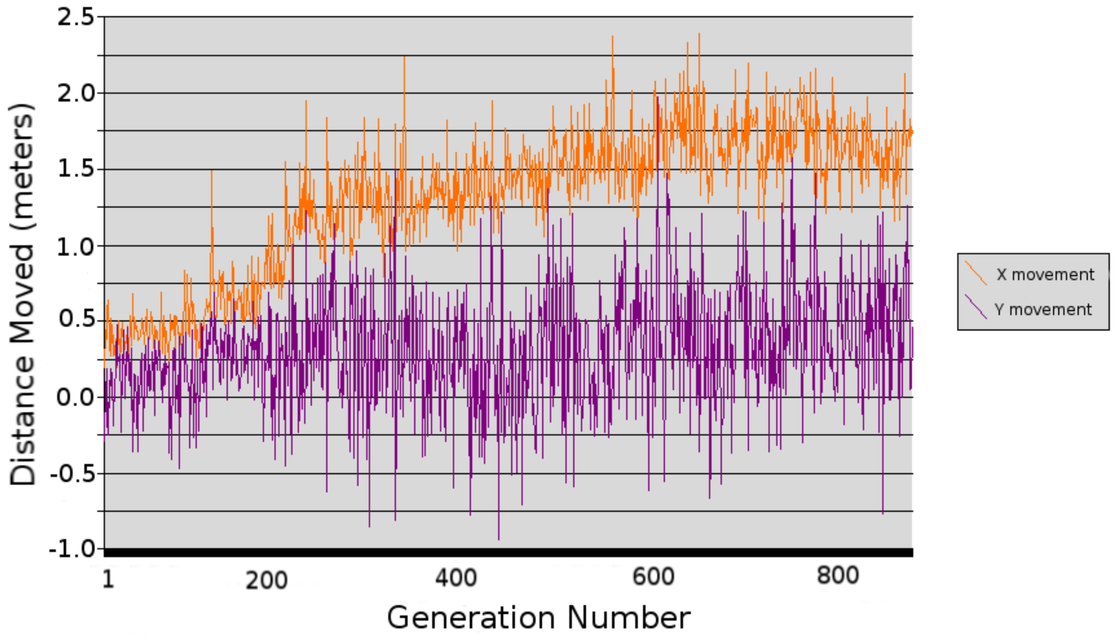

Figure 5.2: X and Y Displacement for the 30-Perceptron Run

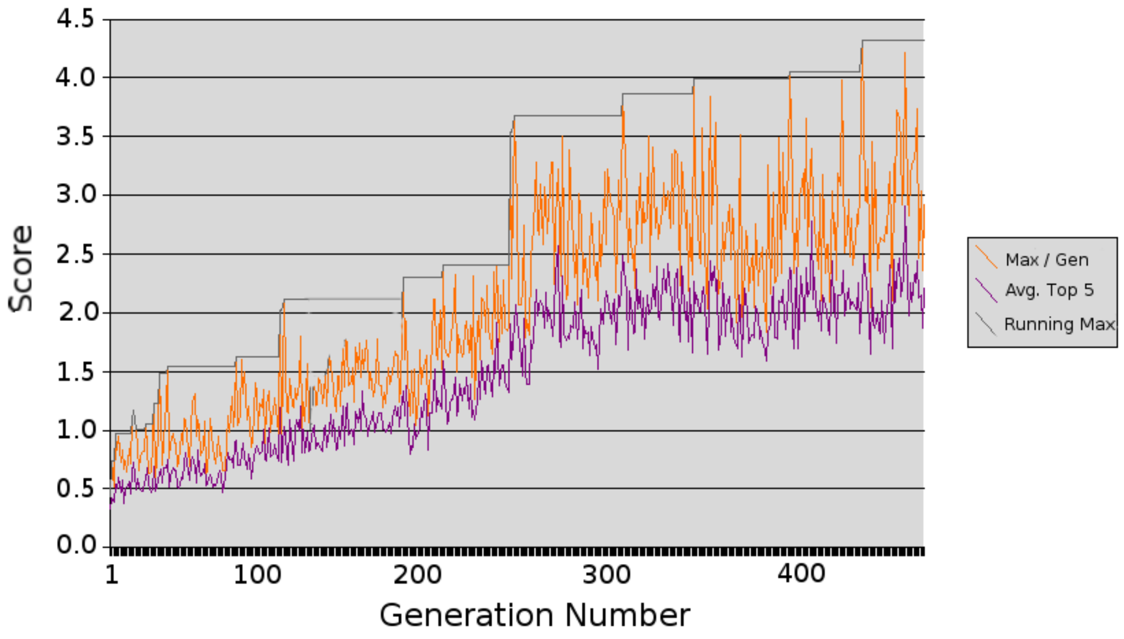

Figure 5.3: Scores Per-Generation From the 150-Perceptron Run

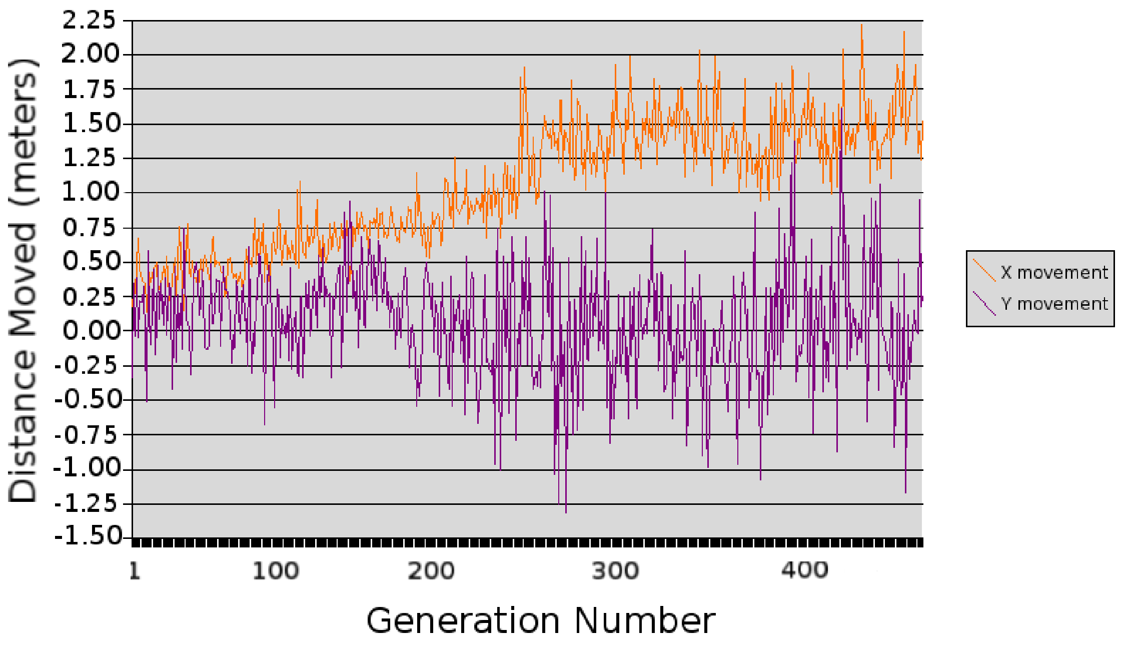

Figure 5.4: X and Y Displacement From 150-Perceptron Run

5.3 DISCUSSION OF RESULTS

Note that the first run (30-perceptrons) went for 405 generations, while the second (150-perceptrons) run was only 240 generations. Both tests ran for approximately 11 days in real-world time, each running on one core of the same CPU, but the larger neural network slowed down the software considerably on the second run. This is to be expected, as the neural networks from the first run consume only 59MB of RAM, while those from the second run consume 179MB —and all of these weights need to be processed 2,000 times per turn, and 160,000 times per generation.

Several other things are apparent from Figures 5.1-5.4. First, the data has quite a bit of randomness in it—there is a large amount of inconsistency between generations in both the scores and displacements. Secondly, while the scores are generally rising as the generations progress, they do so in a very chaotic way, with relatively flat periods and periods of rapid increase. There is even what appears to be a period of decrease in the scores in Figure 5.1. Third, Figures 5.2 and 5.4 show the X component of motion increasing with the score, while the Y component remains approximately centered at zero, but with steadily increasing random variation.

The first observation can be explained by the fact that the robots the system is being asked to control are randomly generated. Thus, a neural network that performs well in one generation may be do poorly with the robot it is given in the next generation. This is intentional, as the goal is to evolve a control system which is effective in a wide variety of robots (thus increasing the chance that it will work well with a physical robot in the real world). In addition, it is possible for an otherwise strong-performing control system to flip its robot upside-down, obtaining a very low (or negative) score in the process. This tends to be especially likely with the very high scoring individuals in any generation, as they tend to be the “risk takers”. This issue can be exacerbated by the randomness in the robot parameters, as a behavior that is only slightly risky in one robot may be fatal in another.

The chaotic nature of the increases in score over time can be explained by the properties of the genetic algorithm. The software is continually recombining the same characteristics into new population members, only occasionally happening upon a new adaptation that results in significantly higher scores. It takes time, however, for this adaptation to propagate through the population, and be optimized to work in a consistent way. Thus, there can be a very large jump in the running maximum, creating a “high score” that holds for quite some time. The apparent decrease in score in the 30-perceptron run (Figure 5.1) could be due to the “deaths” of several population members which, while high-scoring, were also highly inconsistent. This is backed up by the fact that the randomness in the plot drops off very quickly during the same few generations, and remains smaller than before as the scores recover.

The movement in the X direction (which is always the “desired” direction in these two runs, as explained above) behaves as one would expect; it appears to increase along with the scores. The Y movement, however, remains approximately centered at zero, but has a random noise in it that increases through the generations. This can be explained by the fact that the control system is becoming more effective at moving the robots in general, and because the population members still receive points for moving along the Y axis. In later generations, this movement is small compared to the motion in the X direction, as the control system improves at directing the robot in the direction of maximum score. This side movement could also be suppressed by slowly introducing a penalty for movement in the Y direction, especially if an additional input was added to the control system for current (absolute) position.

Finally, it is worth pointing out that the 30- and 150- perceptron tests were only allowed to run for 860 and 485 generations, respectively, due to time limitations. Previous runs that were much longer, including one that went into the thousands of generations, showed a continued increase in performance, with the longest run producing several scores between 8 and 9 on each generation. The plots here are, however, sufficient to show that the ability of the AI to control a robot is generally rising with time, and to show some of its characteristics.

6. CONCLUSIONS AND FURTHER RESEARCH

6.1 CONCLUSIONS

From the results given in section 5.2, as well as direct observation of the simulated robots in the software, it is clear that this system is capable of generating effective walking movement. In addition, the robot design used in this thesis is particularly difficult to control, as its wide body does not permit the center of mass to remain in a stable position. In quadruped animals, the body is long and narrow, so that diagonal pairs of feet that are on the ground form a straight line that is always beneath the center of mass. With a hexapod or octopod, the problem would be even easier, as the feet on the ground at any given time form a triangle or a trapezoid, respectively, that can always enclose the center of mass on the horizontal plane. Thus, this method can be expected to produce better results than those given here for these other body types.

6.2 CONTINUED WORK WITH THIS BUILD

The first step that should be taken in order to learn more about this system is to perform more extensive testing than what was done for this thesis in order to maximize the efficiency of the system with respect to CPU load and memory usage. This will require a large number of test runs to be performed with many different configurations, in order to optimize the following variables:

- Population size
- Number of perceptrons

- Memory length
- Probability of each type of mutation
- Scoring with respect to different criteria
- Selection rules

In order to perform a large number of tests in a reasonable amount of time, it would be best to use a computer with a large number of processor cores, as this software does not parallelize easily in its current form. Alternatively, the physics engine could be replaced with one that runs on a stream processor, such as PhysX from Nvidia, which runs on their GeForce 8 and newer graphics cards, and the neural network could be rewritten to run on a GPU.

6.3 EXTENSION OF CONTROL SYSTEM

It would also be good to extend the scope of the control systems that are produced in a few different ways. First, multiple neural networks can be used, with each trained to perform a different task. While individual networks have been observed to produce multiple behaviors in this system, this would be a good way to separate the desired behaviors. Also, it might be effective to have “nested” learning rules, such that the neural network continues to learn on its own after it is produced by the genetic algorithm. This could be done by adding some form of short-term reinforcement learning, or by adding a classifier network to the inputs of the control system that predicts the result of current behavior on the score and adjusts the weights of the network, perhaps using the P-Delta learning rule[6] that originally went with the parallel perceptron network that is used here. Another option may be to add some outputs that do not control anything, but still act as feedback loops. This would create a form of memory that permits state-space orbits that last much longer than the history-buffer length, which the system would use in whatever way happens to produce the highest scores.

6.4 POTENTIAL APPLICATIONS

In terms of applications, there are two things that would be very interesting to do. One such idea is to create a CAD-style robot “editor” in which robots can be designed in a quick and convenient way, instead of writing a 1500+ line constructor, as was done with the spiderBody class used in this research. This editor would allow one to create a robot using a library of predefined parts such as the linear servos seen on the robot that this thesis deals with, and automatically generate a bill of materials for its physical construction. After the robot is designed, the software can then be used to create parts of its control system.

The second possibility is to modify the simulation and genetic algorithm software to operate as a P2P application, in a similar way to the BitTorrent network. A large number of users who want the same robot could download a task file that specifies the robot that is to be controlled and points to an online “tracker”. Having connected to the tracker, a user's client would join the “swarm” of other users, and begin receiving population members to evaluate. Each user's PC processes a small population, similar to the ones that were used in the two test runs here, but downloads a few new neural networks from other users and transmits a few on each generation. Depending on the number of users who want a particular robot, this could permit effective population sizes in the tens of thousands. Like the other possibilities mentioned above, this has not been evaluated at this point, and it is unknown whether it would be an effective design. It would, however, be very interesting to see what might come out of it.

APPENDICES

APPENDIX A: NEURAL NETWORK SOURCE CODE

This Appendix shows the source code that implements the neural network used in this thesis. There are two sections to this source code: the mcNeuron object class, and the historyBuffer object class.

The mcNeuron class implements the neural network itself. This class functions as a linked list, where each instance manages a single perceptron, and contains a pointer to the memory address of the next perceptron. Thus, the perceptrons are organized in a chain structure, so that the software using this class need only interact with the first instance in the chain. The member functions of this class, and their calling conventions, are described in detail in section 2.5.

The historyBuffer object class implements the memory stack discussed in Chapter 2. The source code for this class begins on the second page of this appendix. The historyBuffer class is structured as a linked list, where each instance of the class acts as one stack layer (see Figure 2.2). When a new vector is to be added to the stack, a new historyBuffer object is created, and the previous top layer is passed as an argument. To avoid creating a memory leak, the recursive killOldest() member function is called on the top stack layer, which causes the last layer in the stack to be deleted.

Note that this stack is managed externally to the mcNeuron class, so the memory address of the top layer must be passed as an argument to several of the mcNeuron member functions.

```
///  ░▒▓█▓▒░▒▓█▓▒░▒▓█▓▒░▒▓█▓▒░▒▓█▓▒░▒▓█▓▒░▒▓█▓▒░▒▓█▓▒░▒▓█▓▒░▒▓█▓▒░▒▓█▓▒░▒▓█▓▒░
///  █▓▒░▒▓█▓▒░▒▓█▓▒░▒▓█▓▒░▒▓ historyBuffer class █▓▒░▒▓█▓▒░▒▓█▓▒░▒▓█▓▒░▒▓█▓▒░
///  ░▒▓█▓▒░▒▓█▓▒░▒▓█▓▒░▒▓█▓▒░▒▓█▓▒░▒▓█▓▒░▒▓█▓▒░▒▓█▓▒░▒▓█▓▒░▒▓█▓▒░▒▓█▓▒░▒▓█▓▒░

///                                                          ______________________________//
///_________________________________________________________/ @@  CLASS DECLARATION  @@     //
//                                                                                          //
class historyBuffer{                                                                        //
                                                                                            //
private:                                                                                    //
        historyBuffer *next;                                                                //
        float *data;                                                                        //
        int dataSize;                                                                       //
                                                                                            //
public:                                                                                     //
        historyBuffer( int, historyBuffer*, float* );   // size, parent, data               //
        historyBuffer( int, historyBuffer* );           // size, parent                     //
        ~historyBuffer();                                                                   //
        float dotValues( float *, int, int );           // dot with the float *             //
        bool killOldest();                              // kills last entry in list         //
        void rollToWeights( float, float*, int, int );  // Adds a scaled copy of HB to weights //
        void testList();                                                                    //
};                                                                                          //
//__________________________________________________________________________________________//


///                                                          ______________________________//
///_________________________________________________________/  Constructor                 //
//                                                                                          //
historyBuffer::historyBuffer( int size, historyBuffer* nextHB, float* myData ) {            //
                                                                                            //
next = nextHB;                                                                              //
                                                                                            //
dataSize = size;                                                                            //
data = new float[ size ];                                                                   //
                                                                                            //
for( int i = 0; i < size; i++ ) {                                                           //
        data[ i ] = myData[ i ];                                                            //
        }                                                                                   //
}                                                                                           //
//__________________________________________________________________________________________//


///                                                          ______________________________//
///_________________________________________________________/  Constructor  (blank buffer) //
//                                                                                          //
historyBuffer::historyBuffer( int size, historyBuffer* nextHB ) {                           //
                                                                                            //
next = nextHB;                                                                              //
                                                                                            //
dataSize = size;                                                                            //
data = new float[ size ];                                                                   //
                                                                                            //
for( int i = 0; i < size; i++ ) {                                                           //
        data[ i ] = 0.0;                                                                    //
        }                                                                                   //
}                                                                                           //
//__________________________________________________________________________________________//


///                                                          ______________________________//
///_________________________________________________________/  Destructor                  //
//                                                                                          //
historyBuffer::~historyBuffer() {                                                           //
                                                /* You can't get much simpler than this... */ //
delete[] data;                                                                              //
                                                                                            //
}                                                                                           //
//__________________________________________________________________________________________//
```

```
///                                                              _______________________________//
///____________________________________________________________/  dotValues()                  //
//                                                                                             //
float historyBuffer::dotValues( float *iptData, int startFrame, int runFrames ) {             //
                                                                                               //
float c_accum = 0.0;                                                                           //
        /*  Count startFrame down to zero before accum  */                                     //
if( startFrame > 0 ) {                                                                         //
        return( next->dotValues( iptData, startFrame - 1, runFrames ) );                       //
        }                                                                                      //
                                                                                               //
                                                                                               //
                /*  Accumulate on own data  */                                                 //
for( int i = 0; i < dataSize; i++ ) {                                                          //
        c_accum += iptData[ i ] * data[ i ];                                                   //
        }                                                                                      //
                                                                                               //
                                                                                               //
if( runFrames > 1 ) {                                                                          //
        c_accum += next->dotValues( &iptData[ dataSize ], 0, runFrames - 1 );                  //
        }                                                                                      //
                                                                                               //
return( c_accum );                                                                             //
                                                                                               //
}                                                                                              //
//_____________________________________________________________________________________________//


///                                                              _______________________________//
///____________________________________________________________/  killOldest()                 //
//                                                                                             //
bool historyBuffer::killOldest() {                                                             //
bool imLast;                                                                                   //
                                                                                               //
if( next == NULL ) { return( true ); }                                                         //
                                                                                               //
imLast = next->killOldest();                                                                   //
                                                                                               //
if( imLast ) {                                                                                 //
        delete next;                                                                           //
        next = NULL;                                                                           //
        }                                                                                      //
                                                                                               //
return( false );                                                                               //
                                                                                               //
}                                                                                              //
//_____________________________________________________________________________________________//


///                                                              _______________________________//
///____________________________________________________________/  rollToWeights()              //
//                                                                                             //
void historyBuffer::rollToWeights( float score, float* target, int sFrame, int nFrames ) {    //
                                                        /* Adds a scaled copy of HB to the   */ //
if( sFrame > 0 ) {                                      /* given weights matrix, starting at */ //
next->rollToWeights(score,target,sFrame-1,nFrames);     /* startFrame and going for nFrames  */ //
        return;     /* <---------------------+   */                                            //
        }                               /*   |    */                                           //
        /* <---------------------------------|------------ From here down, sFrame equalled 0 */ //
for( int i = 0; i < dataSize; i++ ) {   /*   +------------ due to this bad boy right here    */ //
        target[ i ] += score * data[ i ];                                                      //
        }                                                                                      //
                                                                                               //
if( nFrames > 1 ) {                                                                            //
        next->rollToWeights( score, &target[ dataSize ], 0, nFrames - 1 );                     //
        }                                                                                      //
                                                                                               //
}                                                                                              //
```

```
//_______________________________________________________________________________________________//


//      TEST CODE       TEST CODE       TEST CODE
//______________________________________________________
void historyBuffer::testList() {                        //
                                                        //
for( int i = 0; i < dataSize; i++ ) {                   //
        cout << data[ i ] << "\n";                      //
        }                                               //
                                                        //      Dumps the list to the screen
if( next != NULL ) {                                    //      to verify that the class works
        next->testList();                               //
        }                                               //
                                                        //
}                                                       //
                                                        //
                                                        //
//______________________________________________________


///  ░▓▓▓░▓▓▓░▓▓▓░▓▓▓░▓▓▓░▓▓▓░▓▓▓░▓▓▓░▓▓▓░▓▓▓░▓▓▓░▓▓▓
///  ▓▓▓░▓▓▓░▓▓▓░▓▓▓░▓▓▓░ mcNeuron class ░▓▓▓░▓▓▓░▓▓▓░▓▓▓░▓▓▓░▓▓▓
///  ░▓▓▓░▓▓▓░▓▓▓░▓▓▓░▓▓▓░▓▓▓░▓▓▓░▓▓▓░▓▓▓░▓▓▓░▓▓▓░▓▓▓


///                                                               ______________________________//
///____________________________________________________________/ @@  CLASS DECLARATION  @@     //
//_____________________________________________________________________________________________//
class mcNeuron {                                                                               //
                                                                                               //
private:                                                                                       //
        mcNeuron *next;    /* This uses the same list architecture as historyBuffer     */      //
        float *iWeights;   /* Input weights (used to determine the neuron's next state) */      //
        float *oWeights;   /* Output weights (the effects of this neuron's action)      */      //
        int numInputs;                                                                         //
        int numWeights;                                                                        //
        int numOutputs;                                                                        //
        int runHL;         /* Run history length (shorter than training history length  */      //
        int trainHL;       /* Training history length                                   */      //
                           /* NOTE: total HL = trainHL + runHL                          */      //
public:                                                                                        //
        mcNeuron( int, int, int, int, int );   /*numNeurons, numInputs, numOutputs, RHL, THL */ //
        ~mcNeuron();                                                                           //
                                                                                               //
                /*    IPTS     IHISTORY        OHISTORY       OPTS       */                    //
        void rnNet( float*, historyBuffer*, historyBuffer*, float* );                          //
                /*       SCORE     IHISTORY        OHISTORY              */                    //
        void updateNet( float, historyBuffer*, historyBuffer* );                               //
                /*    number of dimensions                               */                    //
                                                                                               //
        void iW_preset( float * );                                                             //
        void iW_preset_justOne( float * );                                                     //
        void oW_preset_justOne( float * );                                                     //
        mcNeuron *getNext();                                                                   //
        mcNeuron *cutNth( int );
        void setNext( mcNeuron * );                                                            //
        void appendChain( mcNeuron * );                                                        //
        float *getIWeights();                                                                  //
        float *getOWeights();                                                                  //
                                                                                               //
        void setRandomOWeights( float );                                                       //
        void setRandomIWeights( float );                                                       //
        void setCascadingOWeights( float, int );                                               //
        void shakeIptWeights( float );                                                         //
        void shakeOptWeights( float );                                                         //
        void mutateIptWeights( float );                                                        //
        void mutateOptWeights( float );                                                        //
        void svNet( ofstream * );                                                              //
        void ldNet( ifstream * );                                                              //
```

```
                                                                                        //
        /*  TEST / DEBUG FUNCS:  */                                                     //
        void iW_dump();                                                                 //
        void oW_dump();                                                                 //
        void oW_ortho( int );                                                           //
};                                                                                      //
//______________________________________________________________________________________//


///                                                             _______________________//
///____________________________________________________________/  Constructor          //
// ____________________________________________________________________________________//
mcNeuron::mcNeuron( int nNeurons, int nInputs, int nOutputs, int RHL, int THL ) {       //
                                                                                        //
numWeights = nInputs * ( RHL + 1 ) + nOutputs * RHL;                                    //
                                                                                        //
iWeights = new float[ numWeights ];                                                     //
oWeights = new float[ nOutputs ];                                                       //
numInputs = nInputs;                                                                    //
numOutputs = nOutputs;                                                                  //
runHL = RHL;                                                                            //
trainHL = THL;                                                                          //
                                                                                        //
if( nNeurons > 1 ) { next = new mcNeuron( nNeurons - 1, nInputs, nOutputs, RHL, THL ); }   //
else{ next = NULL; }                                                                    //
                                                                                        //
}                                                                                       //
//______________________________________________________________________________________//


///                                                             _______________________//
///____________________________________________________________/  Destructor           //
// ____________________________________________________________________________________//
mcNeuron::~mcNeuron()   {                                                               //
                                                                                        //
delete[] iWeights;                                                                      //
delete[] oWeights;                                                                      //
if( next ) { delete next; }                                                             //
                                                                                        //
}                                                                                       //
//______________________________________________________________________________________//


///                                                             _______________________//
///____________________________________________________________/  rnNet()              //
// ____________________________________________________________________________________//
void mcNeuron::rnNet( float *iBuf, historyBuffer *iHist, historyBuffer *oHist, float *oBuf ) {  //
                                                                                        //
float c_Accum = 0.0;                                                                    //
                                                                                        //
        /*   Accumulate input weights   */                                              //
for( int i = 0; i < numInputs; i++ ) {                                                  //
        c_Accum += iBuf[ i ] * iWeights[ i ];                                           //
        }                                                                               //
                                                                                        //
                        /*  dotValues( input, startFrame, runFrames )   */              //
c_Accum += iHist->dotValues( &iWeights[ numInputs ], 0, runHL );                        //
c_Accum += oHist->dotValues( &iWeights[ numInputs * ( runHL + 1 ) ], 0, runHL );        //
                                                                                        //
if( c_Accum > 0.0 ) {   /* unit step activation function -- any other could be used also */   //
        for( int i = 0; i < numOutputs; i++ ) { oBuf[ i ] += oWeights[ i ]; }           //
        }                                                                               //
                                                                                        //
if( next ) { next->rnNet( iBuf, iHist, oHist, oBuf ); }                                 //
                                                                                        //
}                                                                                       //
//______________________________________________________________________________________//


///                                                             _______________________//
```

```
///_____________________________________________________________/  updateNet()            //
// ______________________________________________________________________________________//
void mcNeuron::updateNet( float myScore, historyBuffer *iHist, historyBuffer *oHist ) {   //
/* It starts a frame newer with the ipt weights than the output */                        //
/* since each rnNet sees iWeights • <ipts; iptHist; optHist>    */                        //
                                                                                          //
float delta = myScore * oHist->dotValues( oWeights, 0, 1 );                              //
                                                                                          //
for( int i = 0; i < trainHL; i++ ) {                                                      //
        iHist->rollToWeights( delta, iWeights, i, runHL + 1 );                            //
        oHist->rollToWeights( delta, &iWeights[ ( runHL + 1 ) * numInputs ], i + 1, runHL );   //
        }                                                                                 //
                                                                                          //
if( next ) { next->updateNet( myScore, iHist, oHist ); }                                  //
                                                                                          //
}                                                                                         //
//_______________________________________________________________________________________//


///                                                              ________________________//
///_____________________________________________________________/  iW_preset()            //
// ______________________________________________________________________________________//
// Preloads neurons with weights from an array of floats                                  //
void mcNeuron::iW_preset( float *myNewWeights ) {                                         //
                                                                                          //
for( int i = 0; i < numWeights; i++ ) {                                                   //
        iWeights[ i ] = myNewWeights[ i ];                                                //
        }                                                                                 //
                                                                                          //
if( next ) { next->iW_preset( &myNewWeights[ numWeights ] ); }                            //
                                                                                          //
}                                                                                         //
//_______________________________________________________________________________________//


///                                                              ________________________//
///_____________________________________________________________/  iW_preset_justOne()    //
// ______________________________________________________________________________________//
// Preloads just this neuron with input weights from an array of floats                   //
void mcNeuron::iW_preset_justOne( float *myNewWeights ) {                                 //
                                                                                          //
for( int i = 0; i < numWeights; i++ ) {                                                   //
        iWeights[ i ] = myNewWeights[ i ];                                                //
        }                                                                                 //
                                                                                          //
}                                                                                         //
//_______________________________________________________________________________________//


///                                                              ________________________//
///_____________________________________________________________/  oW_preset_justOne()    //
// ______________________________________________________________________________________//
// Preloads just this neuron with output weights from an array of floats                  //
void mcNeuron::oW_preset_justOne( float *myNewWeights ) {                                 //
                                                                                          //
for( int i = 0; i < numOutputs; i++ ) {                                                   //
        oWeights[ i ] = myNewWeights[ i ];                                                //
        }                                                                                 //
                                                                                          //
}                                                                                         //
//_______________________________________________________________________________________//


///                                                              ________________________//
///_____________________________________________________________/  getNext()              //
// ______________________________________________________________________________________//
mcNeuron *mcNeuron::getNext() {                                                           //
                                                                                          //
return( next );                                                                           //
                                                                                          //
}                                                                          //
//_________________________________________________________________________//


///                                                 _______________________//
///______________________________________________/  cutNth()              //
//                                                                         //
mcNeuron *mcNeuron::cutNth( int N ) {                                      //
                                                                           //
mcNeuron * retVal;                                                         //
if( N > 1 ) { return( next->cutNth( N - 1 ) ); }                           //

else if( N == 1 ) {
        retVal = next->cutNth( N - 1 );
        next = NULL;
        return( retVal );
        }

else{ return( this ); }
                                                                           //
                                                                           //
}                                                                          //
//_________________________________________________________________________//


///                                                 _______________________//
///______________________________________________/  setNext()             //
//                                                                         //
void mcNeuron::setNext( mcNeuron *newNext ) {                              //
                                                                           //
next = newNext;                                                            //
                                                                           //
}                                                                          //
//_________________________________________________________________________//


///                                                 _______________________//
///______________________________________________/  appendChain()         //
//                                                                         //
void mcNeuron::appendChain( mcNeuron *newChunk ) {     // Appends newChunk to the end of the   //
                                                       // current chain    //
if( next ) { next->appendChain( newChunk ); }                              //
else { next = newChunk; }                                                  //
                                                                           //
}                                                                          //
//_________________________________________________________________________//


///                                                 _______________________//
///______________________________________________/  getIWeights()         //
//                                                                         //
float *mcNeuron::getIWeights() {                                           //
                                                                           //
return( iWeights );                                                        //
                                                                           //
}                                                                          //
//_________________________________________________________________________//


///                                                 _______________________//
///______________________________________________/  getOWeights()         //
//                                                                         //
float *mcNeuron::getOWeights() {                                           //
                                                                           //
return( oWeights );                                                        //
                                                                           //
}                                                                          //
//_________________________________________________________________________//
```

```
///                                                           _______________________________//
///_________________________________________________________/  setRandomIWeights()          //
//                                                                                          //
void mcNeuron::setRandomIWeights( float maxVal ) {                                          //
                                                                                            //
for( int i = 0; i < numWeights; i++ ) { iWeights[ i ] = rollFloat( -maxVal, maxVal ); }     //
if( next ) { next->setRandomIWeights( maxVal ); }                                           //
                                                                                            //
}                                                                                           //
//__________________________________________________________________________________________//


///                                                           _______________________________//
///_________________________________________________________/  setRandomOWeights()          //
//                                                                                          //
void mcNeuron::setRandomOWeights( float maxVal ) {                                          //
                                                                                            //
for( int i = 0; i < numOutputs; i++ ) { oWeights[ i ] = rollFloat( -maxVal, maxVal ); }     //
if( next ) { next->setRandomOWeights( maxVal ); }                                           //
                                                                                            //
}                                                                                           //
//__________________________________________________________________________________________//


///                                                           _______________________________//
///_________________________________________________________/  setCascadingOWeights()       //
//                                                                                          //
void mcNeuron::setCascadingOWeights( float amount, int oIndex ) {                           //
                                        /* Clicks through neurons, setting one output weight */ //
oWeights[ oIndex ] = amount;            /* to (amount). When the last output is passed, the  */ //
                                        /* function starts back at zero, but with -(amount)  */ //
if( next && oIndex < numOutputs ) { next->setCascadingOWeights( amount, oIndex + 1 ); }     //
else if( next ) { next->setCascadingOWeights( -amount, 0 ); }                               //
                                                                                            //
}                                                                                           //
//__________________________________________________________________________________________//


///                                                           _______________________________//
///_________________________________________________________/  shakeIptWeights()            //
//                                                                                          //
void mcNeuron::shakeIptWeights( float maxAmount ) {                                         //
                                                                                            //
float curSum;                                                                               //
for( int i = 0; i < numWeights; i++ ) {                                                     //
        iWeights[ i ] += rollFloat( -maxAmount, maxAmount );                                //
        curSum += pow( iWeights[ i ], 2.0 );                                                //
        }                                                                                   //
curSum = 1.0 / sqrt( curSum );                                                              //
                                                                                            //
for( int i = 0; i < numWeights; i++ ) {                                                     //
        iWeights[ i ] *= curSum;                                                            //
        }                                                                                   //
                                                                                            //
}                                                                                           //
//__________________________________________________________________________________________//


///                                                           _______________________________//
///_________________________________________________________/  shakeOptWeights()            //
//                                                                                          //
void mcNeuron::shakeOptWeights( float maxAmount ) {                                         //
                                                                                            //
for( int i = 0; i < numOutputs; i++ ) {                                                     //
        oWeights[ i ] += rollFloat( -maxAmount, maxAmount );                                //
        }                                                                                   //
                                                                                            //
}                                                                                           //
//__________________________________________________________________________________________//
```

```
///                                                              _______________________________//
///_____________________________________________________________/  mutateIptWeights()           //
//                                                                                              //
void mcNeuron::mutateIptWeights( float wMax ) {                                                 //
                                                                                                //
int startIndex = (int)rollFloat( 0.0, (float)numWeights );                                      //
int stopIndex = (int)rollFloat( (float)startIndex, (float)numWeights );                         //
                                                                                                //
for( int i = startIndex; i < stopIndex; i++ ) {                                                 //
        iWeights[ i ] = rollFloat( -wMax, wMax );                                               //
        }                                                                                       //
                                                                                                //
}                                                                                               //
//______________________________________________________________________________________________//


///                                                              _______________________________//
///_____________________________________________________________/  mutateOptWeights()           //
//                                                                                              //
void mcNeuron::mutateOptWeights( float wMax ) {                                                 //
                                                                                                //
int startIndex = (int)rollFloat( 0.0, (float)numOutputs );                                      //
int stopIndex = (int)rollFloat( (float)startIndex, (float)numOutputs );                         //
                                                                                                //
for( int i = startIndex; i < stopIndex; i++ ) {                                                 //
        oWeights[ i ] = rollFloat( -wMax, wMax );                                               //
        }                                                                                       //
                                                                                                //
}                                                                                               //
//______________________________________________________________________________________________//


///                                                              _______________________________//
///_____________________________________________________________/  svNet()                      //
//                                                                                              //
void mcNeuron::svNet( ofstream * saveFile ) {                                                   //
                                                                                                //
        saveFile->write( (char *)iWeights, numWeights * sizeof( float ) );                      //
        saveFile->write( (char *)oWeights, numOutputs * sizeof( float ) );                      //
        if( next != NULL ) {                                                                    //
                next->svNet( saveFile );                                                        //
                }                                                                               //
}                                                                                               //
//______________________________________________________________________________________________//


///                                                              _______________________________//
///_____________________________________________________________/  ldNet()                      //
//                                                                                              //
void mcNeuron::ldNet( ifstream * loadFile ) {                                                   //
                                                                                                //
        loadFile->read( (char *)iWeights, numWeights * sizeof( float ) );                       //
        loadFile->read( (char *)oWeights, numOutputs * sizeof( float ) );                       //
        if( next != NULL ) {                                                                    //
                next->ldNet( loadFile );                                                        //
                }                                                                               //
}                                                                                               //
//______________________________________________________________________________________________//


/*

rnNet order:

Ipt     iHist   oHist   Opt
  1      NULL    NULL     1
  2         1       1     2
  3         2       2     3
```

```
history save order:

iHist   oHist
________________
 +-----+
 | n   |  n
 |     +------+
 |n-1    n-1  |
 | ...   ...  |
 |n-RHL n-RHL |
 +------------+
   ●       ●
   ●       ●
   ●       ●
________________



                              */

/// VERIFICATION / DEBUG CODE BELOW THIS LINE     VERIFICATION / DEBUG CODE BELOW THIS LINE
/// VERIFICATION / DEBUG CODE BELOW THIS LINE     VERIFICATION / DEBUG CODE BELOW THIS LINE
/// VERIFICATION / DEBUG CODE BELOW THIS LINE     VERIFICATION / DEBUG CODE BELOW THIS LINE
/// VERIFICATION / DEBUG CODE BELOW THIS LINE     VERIFICATION / DEBUG CODE BELOW THIS LINE


//      oW_Ortho()
// Makes output weights orthogonal (one neuron -> one opt position)     //
void mcNeuron::oW_ortho( int numLeft ) {                                //
                                                                        //
for( int i = 0; i < numOutputs; i++ ) {                                 //
        oWeights[ i ] = 0.0;                                            //
        }                                                               //        Sets the opt weights
                                                                        //        so a single dimension
oWeights[ numLeft - 1 ] = 1.0;                                          //        is 1.0 for each cell
                                                                        //
if( next && numLeft >= 0 ) { next->oW_ortho( numLeft - 1 ); }           //
                                                                        //
}                                                                       //
//______________________________________________________________________


//      iW_dump()
// Dumps the weight arrays to the screen                                //
void mcNeuron::iW_dump() {                                              //
                                                                        //
for( int i = 0; i < numWeights; i++ ) {                                 //
        cout << iWeights[ i ] << "\n";                                  //       Dumps weights to stdout
        }                                                               //
cout << "=========\n";                                                  //
if( next ) { next->iW_dump(); }                                         //
}                                                                       //
//______________________________________________________________________


//      iW_dump()
// Dumps the weight arrays to the screen                                //
void mcNeuron::oW_dump() {                                              //
                                                                        //
for( int i = 0; i < numOutputs; i++ ) {                                 //
        cout << oWeights[ i ] << "\n";                                  //       Dumps weights to stdout
        }                                                               //
cout << "=========\n";                                                  //
if( next ) { next->oW_dump(); }                                         //
}                                                                       //
//______________________________________________________________________
```

APPENDIX B: GENETIC ALGORITHM SOURCE CODE

This Appendix provides the source code for the genetic algorithm that was used in this thesis. This genetic algorithm is implemented by the mcEVO object class, in addition to the two helper functions, rankNodes() and breedNets().

The mcEVO class is structured as a linked list, and encapsulates the neural network, the memory stack, and the physical dimensions of the simulated robot to be controlled. It provides functions to load, save and manipulate the population of neural networks. The member functions of this class are described in detail in section 3.5.

The rankNodes() helper function implements a sorting algorithm that ranks a chain of mcEVO instances based on the score values stored in them, in descending order. Note that the instance passed to rankNodes() will be moved to a random location in the chain. Thus, the function mcEVO::getFirst() is used after the ranking to reacquire the beginning of the chain.

The breedNets() helper function comprises most of the actual genetic algorithm. Specifically, it implements the selection, crossover, and mutation operations that are discussed in sections 3.3 and 3.4.

```
/*
        Genetic algorithm add-on for mcNeuron class
        R. Bishop       21 June 2008
                                                                                            */



///                                                              _______________________________//
///_____________________________________________________________/ @@  CLASS DECLARATION  @@     //
// ____________________________________________________________________________________________//
class mcEVO {                                                                                   //
                                                                                                //
private:                                                                                        //
        mcEVO * previous;                                                                       //
        mcEVO *next;                                                                            //
        mcNeuron *myBrain;                                                                      //
        historyBuffer *myIHistory;                                                              //
        historyBuffer *myOHistory;                                                              //
        float myScore;                                                                          //
                                                                                                //
public:                                                                                         //
        dReal mySpiderParams[ 27 ];     // This isn't inherited; the brain must learn to        //
                                        // drive different-sized spiderbots.                    //
        dReal mySKLL;                           // knee link length (implicitly generated)      //
        dReal mySKBR;                           // knee ball radius (not saved elsewhere)       //
        //------------------------------------------------------------------------------------//
        mcEVO( int, mcEVO *, dReal *, dReal * );                                                //
        ~mcEVO();                                                                               //
        mcEVO * getMax( mcEVO *, float );// returns max scoring chain below this                //
        void setPrevious( mcEVO *);     // Sets this member's previous-member pointer           //
        void setNext( mcEVO *);         // Sets called member's next-member pointer             //
        void detach();                  // attaches previous to next, and vice versa            //
        mcEVO *getNext();               // Returns next in chain                                //
        mcEVO *getPrevious();           // Returns previous in chain                            //
        mcEVO *getFirst();              // Returns first in chain (can be called on any member) //
        mcEVO *getLast();               // Returns last in chain (can be called on any member)  //
        float getScore();               // Returns the score assigned to this member            //
        mcEVO *getLastAbove( float );   // Gets last in chain above given score (after ranking) //
        mcEVO *getNth( int );           // Returns Nth in chain after called member             //
        void insBefore( mcEVO * );      // Inserts given mcEVO before the called member         //
        void dumpScores();                      // FIXME FIXME FIXME  TEST CODE TEST CODE TEST CODE
        void dumpWeights();             // Dumps weights to stdout (DEBUG code)                 //
                                                                                                //
        void setScore( float );         // Sets called member's score                           //
                                                                                                //
        dReal *getParams();             // Returns pointer to this member's spider geometry     //
        void appendChain( mcEVO * );            // append target to this chain                  //
        int killLast( int );                    // kills last N in chain                        //
        void svBrains( ofstream * );            // Saves mcNeuron chain to a file               //
        void ldBrains( ifstream * );                                                            //
        void mkBrains( int, int, int );         // Creates a mcNeuron chain to use              //
        void mkBrains_random( int, int, int, float * ); // Creates a mcNeuron chain w/ random w //
                        /*numNeurons, RHL, THL */                                               //
        mcNeuron *getBrain();           // returns this node's mcNeuron chain                   //
        historyBuffer *getIHist();      // Returns a pointer to this member's iHistory chain    //
        historyBuffer *getOHist();      // Returns a pointer to this member's oHistory chain    //
        void setIHist( historyBuffer * );       // Assigns a historyBuffer chain to this node   //
        void setOHist( historyBuffer * );       // Assigns an output historyBuffer chain to this//
};                                                                                              //
//____________________________________________________________________________________________//


///                                                              _______________________________//
///_____________________________________________________________/  Constructor                  //
// ____________________________________________________________________________________________//
mcEVO::mcEVO( int numNodes, mcEVO *myParent, dReal *paramsMin, dReal *paramsMax ) {            //
                                                                                                //
myScore = 0.0;                                                                                  //
myBrain = NULL;                                                                                 //
                                                                                                //
previous = myParent;                                                                            //
```

```
                                                                                                //
if( numNodes > 1 ) {                                                                            //
        next = new mcEVO( numNodes - 1, this, paramsMin, paramsMax );                           //
        }                                                                                       //
                                                                                                //
else { next = NULL; }                                                                           //
                                                                                                //
for( int i = 0; i < 27; i++ ) {                                                                 //
        mySpiderParams[ i ] = rollFloat( paramsMin[ i ], paramsMax[ i ] );                      //
        }                                                                                       //
                                                                                                //
mySKBR = mySpiderParams[ 9 ];                                                                   //
                                                                                                //
}                                                                                               //
//______________________________________________________________________________________________//


///                                                              _______________________________//
///_____________________________________________________________/  Destructor                   //
//                                                                                              //
mcEVO::~mcEVO() {                                                                               //
                                                                                                //
delete myBrain;                                                                                 //
                                                                                                //
if( next ) { delete next; }                                                                     //
                                                                                                //
}                                                                                               //
//______________________________________________________________________________________________//


///                                                              _______________________________//
///_____________________________________________________________/  setPrevious()                //
//                                                                                              //
void mcEVO::setPrevious( mcEVO *newPrevious )   {                                               //
                                                                                                //
previous = newPrevious;                                                                         //
                                                                                                //
}                                                                                               //
//______________________________________________________________________________________________//


///                                                              _______________________________//
///_____________________________________________________________/  setNext()                    //
//                                                                                              //
void mcEVO::setNext( mcEVO *newNext ) {                                                         //
                                                                                                //
next = newNext;                                                                                 //
                                                                                                //
}                                                                                               //
//______________________________________________________________________________________________//


///                                                              _______________________________//
///_____________________________________________________________/  detach()                     //
//                                                                                              //
void mcEVO::detach() {                                                                          //
                        /*  This function detaches this from the chain, so that  */             //
                        /*  next and previous are continuous.                    */             //
if( next ) {                                                                                    //
        next->setPrevious( previous );                                                          //
        }                                                                                       //
                                                                                                //
if( previous ) {                                                                                //
        previous->setNext( next );                                                              //
        }                                                                                       //
                                                                                                //
next = NULL;                                                                                    //
previous = NULL;                                                                                //
                                                                                                //
}                                                                                               //
//______________________________________________________________________________________________//
```

```
///                                                            ______________________________//
///____________________________________________________________/  getMax()                  //
//                                                                                          //
mcEVO * mcEVO::getMax( mcEVO * curWinner, float curMax )       {                            //
                                                                                            //
if( myScore > curMax ) {                                                                    //
        if( next ) { return( next->getMax( this, myScore ) ); }                             //
        else{ return( this ); }                                                             //
        }                                                                                   //
                                                                                            //
else{                                                                                       //
        if( next ) { return( next->getMax( curWinner, curMax ) ); }                         //
        else{ return( curWinner ); }                                                        //
        }                                                                                   //
                                                                                            //
}                                                                                           //
//__________________________________________________________________________________________//


///                                                            ______________________________//
///____________________________________________________________/  getNext()                 //
//                                                                                          //
mcEVO * mcEVO::getNext()        {                                                           //
                                                                                            //
return( next );                                                                             //
                                                                                            //
}                                                                                           //
//__________________________________________________________________________________________//


///                                                            ______________________________//
///____________________________________________________________/  getPrevious()             //
//                                                                                          //
mcEVO * mcEVO::getPrevious()    {                                                           //
                                                                                            //
return( previous );                                                                         //
                                                                                            //
}                                                                                           //
//__________________________________________________________________________________________//


///                                                            ______________________________//
///____________________________________________________________/  getFirst()                //
//                                                                                          //
mcEVO * mcEVO::getFirst()       {                                                           //
                                                                                            //
if( previous ) { return( previous->getFirst() ); }                                          //
else{ return( this ); }                                                                     //
                                                                                            //
}                                                                                           //
//__________________________________________________________________________________________//


///                                                            ______________________________//
///____________________________________________________________/  getLast()                 //
//                                                                                          //
mcEVO *mcEVO::getLast() {                                                                   //
                                                                                            //
if( next ) { return( next->getLast() ); }                                                   //
else { return( this ); }                                                                    //
                                                                                            //
}                                                                                           //
//__________________________________________________________________________________________//


///                                                            ______________________________//
///____________________________________________________________/  getScore()                //
//                                                                                          //
float mcEVO::getScore() {                                                                   //
                                                                                            //
return( myScore );                                                                          //
```

```
                                                                                     //
}                                                                                    //
//___________________________________________________________________________________//


///                                                          ________________________//
///_________________________________________________________/  getLastAbove()       //
// ________________________________________________________________________________//
mcEVO * mcEVO::getLastAbove( float minScore ) {                                      //
        /* Returns last node above given score. Note that this call is sent to the LAST */   //
        /* node in the chain, rather than the first!                              */   //
                                                                                     //
if( myScore > minScore ) { return( this ); }                                         //
else if( previous ) { return( previous->getLastAbove( minScore ) ); }                //
else { return( this ); }        /* This line will never be reached as long */        //
                                /* as the program is working right.       */        //
}                                                                                    //
//___________________________________________________________________________________//


///                                                          ________________________//
///_________________________________________________________/  getNth()             //
// ________________________________________________________________________________//
mcEVO * mcEVO::getNth( int N ) {                                                     //
                                            /* Returns Nth MCE in chain */          //
if( N ) { return( next->getNth( N - 1 ) ); }    /* (first one is ZERO)      */          //
else{ return( this ); }                                                              //
                                                                                     //
}                                                                                    //
//___________________________________________________________________________________//

///                                                          ________________________//
///_________________________________________________________/  insBefore()          //
// ________________________________________________________________________________//
void mcEVO::insBefore( mcEVO * target ) {                                            //
                                                                                     //
if( previous ) {                                                                     //
previous->setNext( target );                                                         //
target->setPrevious( previous );                                                     //
                                                                                     //
previous = target;                                                                   //
target->setNext( this );                                                             //
}                                                                                    //
                                                                                     //
else{                                                                                //
        previous = target;                                                           //
        target->setNext( this );                                                     //
        }                                                                            //
}                                                                                    //
//___________________________________________________________________________________//


///                                                          ________________________//
///_________________________________________________________/  setScore()           //
// ________________________________________________________________________________//
void mcEVO::setScore( float newScore ) {                                             //
                                                                                     //
myScore *= 0.25;                                                                     //
myScore += newScore;                                                                 //
                                                                                     //
}                                                                                    //
//___________________________________________________________________________________//


///                                                          ________________________//
///_________________________________________________________/  getParams()          //
// ________________________________________________________________________________//
dReal *mcEVO::getParams() {                                                          //
                                                                                     //
return( mySpiderParams );                                                            //
                                                                                     //
}                                                                                    //
```

```
//_______________________________________________________________________________________________//


///                                                                ______________________________//
///____________________________________________________________/  appendChain()                 //
// ███████████████████████████████████████████████████████████████████████████████████████████████//
void mcEVO::appendChain( mcEVO *appendix ) {                                                      //
                                                                                                  //
if( next ) { next->appendChain( appendix ); }   /* ASSUMPTION:                          */       //
else{                                           /* Last element in chain to be appended */       //
        next=appendix;                          /* has next = NULL                      */       //
        appendix->setPrevious( this );                                                            //
        }                                                                                         //
                                                                                                  //
}                                                                                                 //
//_______________________________________________________________________________________________//


///                                                                ______________________________//
///____________________________________________________________/  killLast()                    //
// ███████████████████████████████████████████████████████████████████████████████████████████████//
int mcEVO::killLast( int numVictims ) {                                                           //
                                                                                                  //
int a = numVictims;                                /* This function transmits numVictims    */ //
                                                   /* down to the last node, and then       */ //
if( next ) { a = next->killLast( numVictims ); }   /* decrements that value as it returns.  */ //
                                                   /* When a = 0, it has returned numVictims */ //
if( a == 0 ) {                                     /* steps from the final node, and is     */ //
        delete next;                               /* ready to delete all nodes after this. */ //
        next = NULL;                                                                              //
        }                                                                                         //
                                                                                                  //
return( a - 1 );                                                                                  //
}                                                                                                 //
//_______________________________________________________________________________________________//


///                                                                ______________________________//
///____________________________________________________________/  svBrains()                    //
// ███████████████████████████████████████████████████████████████████████████████████████████████//
void mcEVO::svBrains( ofstream *optFile ) {     /*  Saves all the mcNeuron chains in      */     //
                                                /*  the mcEVO chain to one file. To save  */     //
myBrain->svNet( optFile );                      /*  just one, do a getBrain() and call    */     //
if( next ) { next->svBrains( optFile ); }       /*  its save function directly            */     //
                                                                                                  //
}                                                                                                 //
//_______________________________________________________________________________________________//


///                                                                ______________________________//
///____________________________________________________________/  ldBrains()                    //
// ███████████████████████████████████████████████████████████████████████████████████████████████//
void mcEVO::ldBrains( ifstream *iptFile ) {                                                       //
                                                /*  See comments in svBrains() above  */          //
myBrain->ldNet( iptFile );                                                                        //
if( next and !iptFile->eof() ) { next->ldBrains( iptFile ); }                                     //
                                                                                                  //
}                                                                                                 //
//_______________________________________________________________________________________________//


///                                                                ______________________________//
///____________________________________________________________/  mkBrains()                    //
// ███████████████████████████████████████████████████████████████████████████████████████████████//
void mcEVO::mkBrains( int nNeurons, int RHLength, int THLength ) {                                 //
// This function creates a new empty mcNeuron chain (use when you want to load a weights file)   //
                                                /*  Note that not all the brain constructor  */ //
                                                /*  args are passed to this, since some of   */ //
                                                /*  them are implied by this application     */ //
myBrain = new mcNeuron( nNeurons, 35, 16, RHLength, THLength );                                   //
myIHistory = new historyBuffer( 35, NULL );     /* Create base HBuf nodes */                      //
myOHistory = new historyBuffer( 16, NULL );                                                       //
```

```
                                                                                   //
                                                /* Populate the HBuf chain with empty */     //
for( int i = 0; i < RHLength + THLength; i++ ) {   /* buffers, so nothing segfaults...   */     //
        myIHistory = new historyBuffer( 35, myIHistory );                          //
        myOHistory = new historyBuffer( 16, myOHistory );                          //
        }                                 //
                                                                                   //
if( next ) { next->mkBrains( nNeurons, RHLength, THLength ); }                     //
                                                                                   //
}                                                                                  //
//_________________________________________________________________________________//


///                                                        ________________________//
///______________________________________________________/  mkBrains_random()      //
//                                                                                 //
void mcEVO::mkBrains_random( int nNeurons, int RHLength, int THLength, float* usable_array ) {  //
// This function generates a new random mcNeuron chain for each EVO node           //
int weightCount = 51 * nNeurons * ( RHLength + THLength );                         //
//float desiredWeights[ weightCount ];                                             //
//for( int i = 0; i < 100000; i++ ) { usable_array[ i ] = rollFloat( -1.0, 1.0 ); } //
                                             /*  Note that not all the brain constructor  */ //
                                             /*  args are passed to this, since some of   */ //
                                             /*  them are implied by this application     */ //
myBrain = new mcNeuron( nNeurons, 35, 16, RHLength, THLength );                    //
myBrain->setRandomOWeights( 0.001 );                                               //
//myBrain->setCascadingOWeights( 0.001, 0 );                                       //
//myBrain->iW_preset( usable_array );                                              //
myBrain->setRandomIWeights( 0.1 );                                                 //
//myBrain->shakeIptWeights( 0.1 );                                                 //
                                                                                   //
myIHistory = new historyBuffer( 35, NULL );      /* Create base HBuf nodes */      //
myOHistory = new historyBuffer( 16, NULL );                                        //
                                                                                   //
                                                /* Populate the HBuf chain with empty */     //
for( int i = 0; i < RHLength + THLength; i++ ) {   /* buffers, so nothing segfaults...   */     //
        myIHistory = new historyBuffer( 35, myIHistory );                          //
        myOHistory = new historyBuffer( 16, myOHistory );                          //
        }                                                                          //
                                                                                   //
if( next ) { next->mkBrains_random( nNeurons, RHLength, THLength, usable_array ); } //
                                                                                   //
}                                                                                  //
//_________________________________________________________________________________//


///                                                        ________________________//
///______________________________________________________/  getBrain()             //
//                                                                                 //
mcNeuron* mcEVO::getBrain()     {                                                  //
                                                                                   //
return( myBrain );                                                                 //
                                                                                   //
}                                                                                  //
//_________________________________________________________________________________//


///                                                        ________________________//
///______________________________________________________/  getIHist()             //
//                                                                                 //
historyBuffer *mcEVO::getIHist() {                                                 //
                                                                                   //
return( myIHistory );                                                              //
                                                                                   //
}                                                                                  //
//_________________________________________________________________________________//


///                                                        ________________________//
///______________________________________________________/  getOHist()             //
//                                                                                 //
historyBuffer *mcEVO::getOHist() {                                                 //
```

```
                                                                                                 //
return( myOHistory );                                                                            //
                                                                                                 //
}                                                                                                //
//_______________________________________________________________________________________________//


///                                                              ________________________________//
///____________________________________________________________/   setIHist()                   //
//  ==============================================================================================//
void mcEVO::setIHist( historyBuffer *newIHist ) {                                                //
                                                                                                 //
myIHistory = newIHist;                                                                           //
                                                                                                 //
}                                                                                                //
//_______________________________________________________________________________________________//


///                                                              ________________________________//
///____________________________________________________________/   setOHist()                   //
//  ==============================================================================================//
void mcEVO::setOHist( historyBuffer *newOHist ) {                                                //
                                                                                                 //
myOHistory = newOHist;                                                                           //
                                                                                                 //
}                                                                                                //
//_______________________________________________________________________________________________//


void mcEVO::dumpWeights() {

myBrain->iW_dump();
myBrain->oW_dump();

//if( next ) { next->dumpWeights(); }

}




///     ● •  ●  •  ●  •  ● • ●  •  ●  • • ●  • ●  •  ●  ●  •  •  ●  • ●  • • ●
///       ●  • • ●  • ●  •  ●  ●  •  •  ●  • ●  ● • ●  • ●  •  ●  ●  •  •  ●  •
///     ●•+-----------------------------------------------------------------+ ●
///     • | RELATED FUNCTIONS (not members of mcEVO class, but used with it) |  •
///       +-----------------------------------------------------------------+ •
///     ● •  ●  •  ●  •  ● • ●  •  ●  • • ●  • ●  •  ●  ●  •  •  ●  • ●  • • ●
///     • ●  • • ●  • ●  •  ●  ●  •  •  ●  • ●  • • ●  • ●  •  ●  ●  •  •  ●  •


///                                                              ________________________________//
///____________________________________________________________/   rankNodes()                  //
//  ==============================================================================================//
void rankNodes( mcEVO *myChain ) {                                                               //
mcEVO *first = myChain;                                                                          //
mcEVO *max;                                                                                      //
                                                                                                 //
while( first ) {                                                                                 //
                                                                                                 //
        max = first->getMax( NULL, -1000.0 );                                                    //
                                                                                                 //
        if( max != first ) {                                                                     //
                max->detach();                                                                   //
                first->insBefore( max );                                                         //
                }                                                                                //
                                                                                                 //
        else{                                                                                    //
                first = first->getNext();                                                        //
                }                                                                                //
        }                                                                                        //
                                                                                                 //
first = max->getFirst();                                                                         //
```

```
}                                                                                  //
//_______________________________________________________________________________//


//mcEVO::mcEVO( int numNodes, mcEVO *myParent, dReal *paramsMin, dReal *paramsMax )
//void mcEVO::mkBrains( int nNeurons, int RHLength, int THLength )
//mcNeuron* mcEVO::getBrain()
//void appendChain( mcEVO * ) <-- call this on the one having the new stuff appended to it


///                                                          __________________________//
///__________________________________________________________/  breedNets()          //
//                                                                                   //
void breedNets( mcEVO *thePopulation, int popSize, int nReplaced, dReal *pMin, dReal *pMax,     //
        int nNeurons, int RHL, int THL, float mutProb, float maxMut, float iRnd, float oRnd ) { //
                                                                                   //
float indexVal, bestScore, scaleVal;                                               //
mcEVO *working, *last;                                                             //
int curNeuron, cpIndex, len1, len2;
mcNeuron *mom, *dad, *junior, *segment1, *segment2, *segment3;
float newOpts[ 16 ];
float storedIpts [ 35 * RHL ];
float *brainOpts, *brainIpts;

        // Start by creating the new MCE chain                                     //
mcEVO *newEVO = new mcEVO( nReplaced, NULL, pMin, pMax );                          //
mcEVO *new_unchanged = newEVO;
newEVO->mkBrains( nNeurons, RHL, THL );                                            //
                                                                                   //
last = thePopulation->getLast();
working = thePopulation->getMax( NULL, -1000.0 );       /* Get the winning score */
bestScore = working->getScore();

for( int i = 0; i < nReplaced; i++ ) {                                             //
                                                                                   //
   /*  NEW selection system:  */

//      indexVal = 1.0 - pow( rollFloat( 0.0, 1.0 ), 2.0 );
//      indexVal *= bestScore;
//      working = last->getLastAbove( indexVal );
//      mom = working->getBrain();

//      indexVal = 1.0 - pow( rollFloat( 0.0, 1.0 ), 2.0 );
//      indexVal *= bestScore;
//      working = last->getLastAbove( indexVal );
//      dad = working->getBrain();

    /* The next several lines randomly select two parent nodes, using a nonlinear (x^2)   */    //
    /* indexing function. The result is that early nodes (high scorers) are usually picked*/    //
        indexVal = pow( rollFloat( 0.0, 1.0 ), 2.0 );                              //
        indexVal *= (float)popSize;                                                //
                                                                                   //
        working = thePopulation->getNth( (int)indexVal );                          //
        mom = working->getBrain();                                                 //
                                                                                   //
        indexVal = pow( rollFloat( 0.0, 1.0 ), 2.0 );                              //
        indexVal *= (float)popSize;                                                //
                                                                                   //
        working = thePopulation->getNth( (int)indexVal );                          //
        dad = working->getBrain();                                                 //
                                                                                   //
        junior = newEVO->getBrain();                                               //
                                                                                   //
        /* Now, we have all three MCN chains selected */                           //
                                                                                   //
curNeuron = 0;
        while( mom ) {                                                             //
                if( rollFloat( -1.0, 1.0 ) > 0.0 ) {                               //
                        junior->iW_preset_justOne( mom->getIWeights() );           //
                        junior->oW_preset_justOne( mom->getOWeights() );           //
                        }                                                          //
                                                                                   //
```

```
            else{                                                                  //
                    junior->iW_preset_justOne( dad->getIWeights() );               //
                    junior->oW_preset_justOne( dad->getOWeights() );               //
                    }                                                              //
                                                        /* Occasionally create a     */ //
                                                        /* major mutation. Options   */ //
            if( mutProb > rollFloat( 0.0, 1.0 ) ) {     /* include position swap,    */ //
                    if( rollFloat( -1.0, 1.0 ) > 0.25 ) {   /* leg-side reversal, and    */ //
    /* Mess with the */     junior->mutateIptWeights( maxMut );/* a new (random) segment */ //
    /* weights a bit */     junior->mutateOptWeights( maxMut );/* of input weights. */      //
                            }

                    /* 15% chance of a side reversal...                              */
                    if( rollFloat( 0.0, 1.0 ) > 0.85 ) {

                                    brainOpts = junior->getOWeights();
                                    brainIpts = junior->getIWeights();

                                    for( int i = 0; i < 16; i++ ) {
                                            newOpts[ i ] = brainOpts[ i ];
                                            }

                                    brainOpts[ 0 ] = newOpts[ 3 ];
                                    brainOpts[ 3 ] = newOpts[ 0 ];

                                    brainOpts[ 1 ] = newOpts[ 2 ];
                                    brainOpts[ 2 ] = newOpts[ 1 ];

                                    brainOpts[ 12 ] = newOpts[ 13 ];
                                    brainOpts[ 13 ] = newOpts[ 12 ];

                                    brainOpts[ 8 ] = newOpts[ 9 ];
                                    brainOpts[ 9 ] = newOpts[ 8 ];

                                    brainOpts[ 4 ] = newOpts[ 7 ];
                                    brainOpts[ 7 ] = newOpts[ 4 ];

                                    brainOpts[ 5 ] = newOpts[ 6 ];
                                    brainOpts[ 6 ] = newOpts[ 5 ];

                                    brainOpts[ 10 ] = newOpts[ 11 ];
                                    brainOpts[ 11 ] = newOpts[ 10 ];

                                    brainOpts[ 14 ] = newOpts[ 15 ];
                                    brainOpts[ 15 ] = newOpts[ 14 ];

                                    for( int i = 0; i < 35; i++ ) {
                                            brainIpts[ i ] = -brainIpts[ i ];
                                            }
                            }

                    /* 15% chance of a front-back reversal...                        */
                    if( rollFloat( 0.0, 1.0 ) > 0.85 ) {

                                    brainOpts = junior->getOWeights();
                                    brainIpts = junior->getIWeights();

                                    for( int i = 0; i < 16; i++ ) {
                                            newOpts[ i ] = brainOpts[ i ];
                                            }

                                    brainOpts[ 0 ] = newOpts[ 7 ];
                                    brainOpts[ 7 ] = newOpts[ 0 ];

                                    brainOpts[ 1 ] = newOpts[ 6 ];
                                    brainOpts[ 6 ] = newOpts[ 1 ];

                                    brainOpts[ 2 ] = newOpts[ 5 ];
                                    brainOpts[ 5 ] = newOpts[ 2 ];

                                    brainOpts[ 3 ] = newOpts[ 4 ];
                                    brainOpts[ 4 ] = newOpts[ 3 ];
```

```
                                brainOpts[ 8 ] = newOpts[ 11 ];
                                brainOpts[ 11 ] = newOpts[ 8 ];

                                brainOpts[ 9 ] = newOpts[ 10 ];
                                brainOpts[ 10 ] = newOpts[ 9 ];

                                brainOpts[ 12 ] = newOpts[ 15 ];
                                brainOpts[ 15 ] = newOpts[ 12 ];

                                brainOpts[ 13 ] = newOpts[ 14 ];
                                brainOpts[ 14 ] = newOpts[ 13 ];

        /*-------------------------------------------------------------------*/

/* When mirroring, add a */      for( int i = 0; i < 35 * RHL; i++ ) {
/* random time delay.    */              storedIpts[ i ] = brainIpts[ i ];
                                         }

                                len1 = 35 * (int)rollFloat( 0.0, (float)RHL );
                                cpIndex = 0;

/* This time, i indexes    */   for( int i = len1; i < 35 * RHL; i++ ) {
/* the source, and cpIndex */           brainIpts[ cpIndex ] = storedIpts[ i ];
/* indexes the target.     */           cpIndex++;
                                        }

/* The later stages loop   */   for( int i = 0; i < len1; i++ ) {
/* back to the start, to   */           brainIpts[ cpIndex ] = storedIpts[ i ];
/* encourage limit cycles. */   }

        /*-------------------------------------------------------------------*/
                        }

                /* 50% chance of strengthening / weakening the oWeights a bit  */
                if( rollFloat( 0.0, 1.0 ) > 0.5 ) {
                        brainOpts = junior->getOWeights();
                        scaleVal = rollFloat( 0.9, 1.1 );
                        for( int i = 0; i < 16; i++ ) {
                                brainOpts[ i ] *= scaleVal;
                                }
                        }


                /* 25% chance of a random time delay (without mirroring)  */
                if( rollFloat( 0.0, 1.0 ) > 0.25 ) {
                        brainIpts = junior->getIWeights();
                        for( int i = 0; i < 35 * RHL; i++ ) {
                                storedIpts[ i ] = brainIpts[ i ];
                                        }

                                len1 = 35 * (int)rollFloat( 0.0, (float)RHL );
                                cpIndex = 0;

/* This time, i indexes    */   for( int i = len1; i < 35 * RHL; i++ ) {
/* the source, and cpIndex */           brainIpts[ cpIndex ] = storedIpts[ i ];
/* indexes the target.     */           cpIndex++;
                                        }

/* The later stages loop   */   for( int i = 0; i < len1; i++ ) {
/* back to the start, to   */           brainIpts[ cpIndex ] = storedIpts[ i ];
/* encourage limit cycles. */   }
                        }

                }       // <--- This curly bracket is the end of the mutation section.  //
                                                                                        //
        junior->shakeIptWeights( iRnd );        /* Add a small bit of randomness to  */ //
        junior->shakeOptWeights( oRnd );        /* all the input / output weights    */ //
                                                                                        //
        mom = mom->getNext();           /* Move all three of these to the next */       //
        dad = dad->getNext();           /* mcNeuron in this mcEVO object       */       //
        junior = junior->getNext();                                                     //
```

```
curNeuron++;
                }                                                                           //

                junior = newEVO->getBrain();                                                //

                /*  50-50 chance of moving a segment of the neuron chain...  */
                if( rollFloat( 0.0, 1.0 ) > mutProb ) {
/* Move neurons  */     cpIndex = (int)rollFloat( 1.0, nNeurons - 3 );
//                              cout << "CPindex_1: " << cpIndex << "\n";
                        len1 = cpIndex;                         // Length before cut
                        segment1 = junior->cutNth( cpIndex );
                        cpIndex = (int)rollFloat( 1.0, (float)(nNeurons - len1 - 3) );
//                              cout << "CPindex_2: " << cpIndex << "\n";
                        len2 = nNeurons - len1 - cpIndex;       // Length of last seg   //
//                              cout << "Len2: " << len2 << "\n";
                        segment2 = segment1->cutNth( cpIndex );
                        junior->appendChain( segment2 );
                                // Now pick somewhere to paste segment1                 //
                        cpIndex = (int)rollFloat( 1.0, (float)( len1 + len2 ) );
//                              cout << "CPindex_3: " << cpIndex << "\n";
                        segment2 = junior->cutNth( cpIndex );
                        junior->appendChain( segment1 );
                        junior->appendChain( segment2 );
                        }


                                                                                            //
        newEVO = newEVO->getNext();     /* Move on to next MCE */                           //
                                                                                            //
        }                                                                                   //
                                                                                            //
/* Now, replace the last nReplaced elements in thePopulation with newEVO */                 //
thePopulation->killLast( nReplaced );                                                       //
thePopulation->appendChain( new_unchanged );                                                //
}                                                                                           //
//__________________________________________________________________________________________//


//
void mcEVO::dumpScores() {              //
                                        //
cout << myScore << "\n";                //
                                        //        dumpScores()
if( next ) { next->dumpScores(); }      //
                                        //
}                                       //
//
```

APPENDIX C: ROBOT CLASS SOURCE CODE

This Appendix provides the source code for the software library that creates and manipulates the simulated robot used in this thesis. This library consists of the spiderBody object class, and several helper functions that are used with it.

This source code is divided into three files. The first file (spider6.h) is a header file that is included in any program which uses this library. This header defines the macros referenced in Tables 4.1 and 4.2, and declares the spiderBody object class.

The second file (spider6.cpp) contains the member functions for the spiderBody object class and the helper functions that are used with it. These functions, and their usage, are discussed in detail in sections 4.4 and 4.5.

The third file (actuator.cpp) contains the code which is used to model the linear servos that are used by the spiderBody class. This file consists of three parts:

- Definition of typedef struct actuator{}, which encapsulates all of the ODE objects that are required to model the linear servo.
- Function genActuator(), which creates an ODE model of the actuator.
- Function delActuator(), which deletes all of the ODE objects that comprise an actuator.

```
/*
	Spiderbot physics class
	R. Bishop		14 May 2008
								*/

#define UCR	params[ 0 ]
#define VAO	params[ 1 ]
#define RISE	params[ 2 ]
#define LCR	params[ 3 ]
#define ULL	params[ 4 ]
#define LLL	params[ 5 ]
#define IBR	params[ 6 ]
#define RBR	params[ 7 ]
//#define KLL	params[ 8 ]
#define KBR	params[ 9 ]
#define UPM	params[ 10 ]
#define LPM	params[ 11 ]
#define LINDENS params[ 12 ]
#define THICK	params[ 13 ]
#define POSX	params[ 14 ]
#define POSY	params[ 15 ]
#define POSZ	params[ 16 ]
#define ULZA	params[ 17 ]
#define LRZA	params[ 18 ]
#define LLZA	params[ 19 ]
#define FBR	params[ 20 ]
#define FBM	params[ 21 ]
#define VABM	params[ 22 ]
#define VATM	params[ 23 ]
#define RABM	params[ 24 ]
#define RATM	params[ 25 ]
#define ULM	params[ 26 ]

#include <math.h>
#include "actuator3.cpp"
//#include "linalg.cpp"


//	New Attribute Order:
//		0	Upper platform radius					UCR
//		1	V actuator upper mount offset (from centers of UP)	VAO
//		2	Distance between upper and lower platforms		RISE
//		3	Lower platform radius					LCR
//		4	Upper leg length					ULL
//		5	Lower Leg Length					LLL
//		6	Distance hip -> V ball on upper leg			IBR
//		7	Hip rotation linkage length				RBR
//		8	Knee link length					KLL
//		9	Distance knee -> knee link attachment			KBR
//		10	Upper platform mass					UPM
//		11	Lower platform mass					LPM
//		12	Square tubing density	(mass / unit length)		LINDENS
//		13	Platform and Leg thickness				THICK
//		14	Starting Position X					POSX
//		15	Starting Position Y					POSY
//		16	Starting Position Z					POSZ
//		17	Upper leg zero angle					ULZA
//		18	Leg rotation zero angle					LRZA
//		19	Lower leg zero angle					LLZA
//		20	Foot ball radius					FBR
//		21	Foot ball mass						FBM
//		22	V Actuator base mass					VABM
//		23	V Actuator tip mass					VATM
//		24	R Actuator base mass					RABM
//		25	R Actuator tip mass					RATM
//		26	Upper leg mass						ULM
class spiderBody {

	///-----------------------------
private:

	dWorldID world;			// Pointer to the ODE "world"
```

```
        dSpaceID mySpace;               // Space for this
        float timeStep;                 // Time step to be used for simulations

        dBodyID core;                   // the "core" of the body
        dBodyID upperLegs[ 4 ];         // the L-shaped upper legs
        dBodyID lowerLegs[ 4 ];         // the lower legs (square bar + sphere on end)

        dJointID hips[ 4 ];             // Ball joints where legs attach to body
        dJointID knees[ 4 ];            // Hinge1 joints at knees

        dMass coreMass;                 // Represents the central body
        dMass upperLegsMass;            // L-shaped upper legs
        dMass lowerLegsMass;            // Lower legs and feet

        dGeomID coreGeom[ 3 ];          // Has three geoms (top, riser, bottom)
        dGeomID upperLegsGeom[ 8 ];     // Has two geoms (leg and rotation bar)
        dGeomID lowerLegsGeom[ 8 ];     // Has two geoms (leg and foot sphere)

        dReal actuatorOffsets[ 12 ];    // Internal "desired" actuator positions
        actuator myActuators[ 12 ];     // Only 12 (not 16) because the knees are being
                                        // modeled as powered hinges, with external
                                        // force equations.
        dReal getKLL();
        dReal kneeZeroAngle;

        ///-----------------------------
public:

        const dReal* corePositions[ 3 ];
        const dReal* coreRotations[ 3 ];

        const dReal* upperLegPositions[ 4 ];
        const dReal* upperLegRotations[ 4 ];

        const dReal* rotLinkPositions[ 4 ];
        const dReal* rotLinkRotations[ 4 ];

        const dReal* lowerLegPositions[ 4 ];
        const dReal* lowerLegRotations[ 4 ];

        const dReal *footBallPositions[ 4 ];
        // It's stupid to rotate a sphere for rendering...

        const dReal* actBasePositions[ 12 ];
        const dReal* actBaseRotations[ 12 ];

        const dReal* actTipPositions[ 12 ];
        const dReal* actTipRotations[ 12 ];

        float upperPlatformBox[ 6 ];
        float riserBox[ 6 ];
        float lowerPlatformBox[ 6 ];
        float upperLegBox[ 6 ];
        float rotLinkBox[ 6 ];
        float lowerLegBox[ 6 ];
        float vactBaseBox[ 6 ];
        float vactTipBox[ 6 ];
        float ractBaseBox[ 6 ];
        float ractTipBox[ 6 ];
        float footBallRadius;
//      float kneeLinkBox[ 6 ];

//      dReal *feltForces;              // Felt forces in actuators
//      dReal *actuatorDutyCycles;      // Desired duty cycles (with error protection)

        spiderBody( dWorldID, dSpaceID, dReal* );       // Constructor
        ~spiderBody();                                  // Destructor

//      void addForce( dReal, dReal, dReal );
        dReal getPos( int );
        dReal getVel( int );
        void addForce( int, dReal );
        void addKneeTorque( int, dReal );
        dReal getKneeAngle( int );
        dReal getKneeOmega( int );
        dReal calcKneeActOffset( dReal, dReal, dReal );
        dReal calcKneeTorque( dReal, dReal, dReal, dReal );
        dReal calcKneeActVel( dReal, dReal, dReal, dReal );
        dBodyID getCore();

        dReal KLL;

//      void updateActuators();         // Updates actuators with actuatorDutyCycles
};

//          Actuator order per leg:
//                      M1, M2, Rotation, Knee
//                (M1, M2 counterclockwise, looking down from above)
//          Leg order:
//                      RF, LF, LR, RR

#include "spider6.cpp"
```

```
//      New Attribute Order:
//              0       Upper platform radius                          UCR
//              1       V actuator upper mount offset (from centers of UP)      VAO
//              2       Distance between upper and lower platforms     RISE
//              3       Lower platform radius                          LCR
//              4       Upper leg length                               ULL
//              5       Lower Leg Length                               LLL
//              6       Distance hip -> V ball on upper leg            IBR
//              7       Hip rotation linkage length                    RBR
//  OBSOLETE    8       Knee link length                               KLL
//              9       Distance knee -> knee link attachment          KBR
//              10      Upper platform mass                            UPM
//              11      Lower platform mass                            LPM
//              12      Square tubing density   (mass / unit length)   LINDENS
//              13      Platform and Leg thickness                     THICK
//              14      Starting Position X                            POSX
//              15      Starting Position Y                            POSY
//              16      Starting Position Z                            POSZ
//              17      Upper leg zero angle                           ULZA
//              18      Leg rotation zero angle                        LRZA
//              19      Lower leg zero angle                           LLZA
//              20      Foot ball radius                               FBR
//              21      Foot ball mass                                 FBM
//              22      V Actuator base mass                           VABM
//              23      V Actuator tip mass                            VATM
//              24      R Actuator base mass                           RABM
//              25      R Actuator tip mass                            RATM
//              26      Upper leg mass                                 ULM

spiderBody::spiderBody( dWorldID targetWorld, dSpaceID targetSpace, dReal *params ) {

/// +----------------------------------------------------------------+
/// | ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻  VARIABLE DECLARATIONS  ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻  |
/// | ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻     AND ASSIGNMENTS     ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻  |
/// +----------------------------------------------------------------+

dReal massOffset[ 3 ];  // Working variables used for
dReal totalMass;        // setup of various body parts
dReal curCOM[ 3 ];      // Current CoM (working var.)
dReal xPrime[ 3 ];
dReal yPrime[ 3 ];
dReal zPrime[ 3 ];
dReal xRot[ 3 ];
dReal yRot[ 3 ];
dReal zRot[ 3 ];

dReal kneeXPrimes[ 4 ][ 3 ];    // These are the local coords that are used
dReal kneeYPrimes[ 4 ][ 3 ];    // in the creation of the upper and lower
dReal kneeZPrimes[ 4 ][ 3 ];    // legs. Note that kXP points from the hip
                                // toward the knee until we reach the lower
                                // leg portion of the constructor, and then
                                // they rotate so kXP points at the foot.

dReal kneeCoords[ 4 ][ 3 ];     // Position of the knees wrt. CoM of core
dReal rotBalls[ 4 ][ 3 ];       // Position of rotation ball joints (ABSOLUTE)
dReal vactBalls[ 4 ][ 3 ];      // Position of V actuator tips (ABSOLUTE)

dReal workingActuator[ 19 ];    // Param array that's filled out
                                // and sent to genActuator()
dMatrix3 rotation;

dReal B1[] = {  LCR + 0.01,  LCR + 0.01, -RISE*0.5 };   // Positions of the hip
dReal B2[] = { -LCR - 0.01,  LCR + 0.01, -RISE*0.5 };   // joints, relative to
dReal B3[] = { -LCR - 0.01, -LCR - 0.01, -RISE*0.5 };   // CoM of core
dReal B4[] = {  LCR + 0.01, -LCR - 0.01, -RISE*0.5 };

world = targetWorld;            // dWorldID of world to build spider in
mySpace = targetSpace;          // dSpaceID of space to build spider in

dMass working;                  // Working dMass object to build legs / body in
```

```
//      CREATE boxes for drawbox for openGL display          //
//___________________________________________________________//
                                                             //
upperPlatformBox[ 0 ] = 2.0 * UCR;                           //
upperPlatformBox[ 1 ] = 2.0 * UCR;                           //
upperPlatformBox[ 2 ] = THICK;                               //
upperPlatformBox[ 3 ] = 0.8;    // The main body and legs are //
upperPlatformBox[ 4 ] = 0.1;    // red, so they stand out from //
upperPlatformBox[ 5 ] = 0.1;    // the terrain               //
                                                             //
lowerPlatformBox[ 0 ] = 2.0 * LCR;                           //
lowerPlatformBox[ 1 ] = 2.0 * LCR;                           //
lowerPlatformBox[ 2 ] = THICK;                               //
lowerPlatformBox[ 3 ] = 0.8;                                 //
lowerPlatformBox[ 4 ] = 0.1;                                 //
lowerPlatformBox[ 5 ] = 0.1;                                 //
                                                             //
riserBox[ 0 ] = THICK;                                       //
riserBox[ 1 ] = THICK;                                       //
riserBox[ 2 ] = RISE;                                        //
riserBox[ 3 ] = 0.8;                                         //
riserBox[ 4 ] = 0.1;                                         //
riserBox[ 5 ] = 0.1;                                         //
                                                             //
upperLegBox[ 0 ] = ULL;                                      //
upperLegBox[ 1 ] = THICK;                                    //
upperLegBox[ 2 ] = THICK;                                    //
upperLegBox[ 3 ] = 0.6;                                      //
upperLegBox[ 4 ] = 0.1;                                      //
upperLegBox[ 5 ] = 0.1;                                      //
                                                             //
rotLinkBox[ 0 ] = THICK - 0.005;                             //
rotLinkBox[ 1 ] = THICK - 0.005;                             //
rotLinkBox[ 2 ] = RBR;                                       //
rotLinkBox[ 3 ] = 0.0;          // Linkages are bright blue  //
rotLinkBox[ 4 ] = 0.0;          // so they can be seen easily //
rotLinkBox[ 5 ] = 1.0;                                       //
                                                             //
lowerLegBox[ 0 ] = LLL;                                      //
lowerLegBox[ 1 ] = THICK;                                    //
lowerLegBox[ 2 ] = THICK;                                    //
lowerLegBox[ 3 ] = 0.5;                                      //
lowerLegBox[ 4 ] = 0.03;                                     //
lowerLegBox[ 5 ] = 0.03;                                     //
                                                             //
//vactBaserBox[ 0 ] = ???       <--- We'll get this later    //
vactBaseBox[ 1 ] = THICK * 1.1;                              //
vactBaseBox[ 2 ] = THICK * 1.1;                              //
vactBaseBox[ 3 ] = 0.4;                                      //
vactBaseBox[ 4 ] = 0.4;                                      //
vactBaseBox[ 5 ] = 0.4;                                      //
                                                             //
//vactTipBox[ 0 ] = ???         <--- We'll get this later    //
vactTipBox[ 1 ] = THICK * 0.88;                              //
vactTipBox[ 2 ] = THICK * 0.88;                              //
vactTipBox[ 3 ] = 0.7;                                       //
vactTipBox[ 4 ] = 0.7;                                       //
vactTipBox[ 5 ] = 0.7;                                       //
                                                             //
//ractBaseBox[ 0 ] = ???        <--- We'll get this later    //
ractBaseBox[ 1 ] = THICK * 1.1;                              //
ractBaseBox[ 2 ] = THICK * 1.1;                              //
ractBaseBox[ 3 ] = 0.7;                                      //
ractBaseBox[ 4 ] = 0.7;                                      //
ractBaseBox[ 5 ] = 0.0;                                      //
                                                             //
//ractTipBox[ 0 ] = ???         <--- We'll get this later    //
ractTipBox[ 1 ] = THICK * 0.88;                              //
ractTipBox[ 2 ] = THICK * 0.88;                              //
ractTipBox[ 3 ] = 0.5;                                       //
ractTipBox[ 4 ] = 0.5;                                       //
ractTipBox[ 5 ] = 0.1;                                       //
```

```
                                                                //
                                                                //
//                                                              //


/// +━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━+
/// | ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻ CREATE CORE BODY OBJECT ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻  |
/// +━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━+

core = dBodyCreate( world );

dMassSetZero( &coreMass );

//  Accumulate the body masses                                  //
//______________________________________________________________//
dMassSetBoxTotal( &working, UPM, 2.0*UCR, 2.0*UCR, THICK );     //
dMassTranslate (&working, 0.0, 0.0, (RISE+THICK)*0.5 );         //
dMassAdd( &coreMass, &working );                                //
                                                                //
dMassSetBoxTotal( &working, RISE*LINDENS, THICK, THICK, RISE ); //
dMassAdd( &coreMass, &working );                                //
                                                                //
dMassSetBoxTotal( &working, LPM, 2.0*LCR, 2.0*LCR, THICK );     //
dMassTranslate( &working, 0.0, 0.0, (RISE+THICK)*0.5 );         //
dMassAdd( &coreMass, &working );                                //
//


//      Position and attach the core mass                       //
//______________________________________________________________//
                                                                //
// Save this so we can position the core later                  //
massOffset[ 0 ] = coreMass.c[ 0 ];                              //
massOffset[ 1 ] = coreMass.c[ 1 ];                              //
massOffset[ 2 ] = coreMass.c[ 2 ];                              //
                                                                //
// Move the mass so its CoM is at the origin                    //
dMassTranslate (&coreMass, -coreMass.c[0],-coreMass.c[1],       //
                                -coreMass.c[2]);                //
                                                                //
dBodySetMass( core, &coreMass );                                //
                                                                //
// Move it back now that it's created                           //
dBodySetPosition( core,massOffset[0]+POSX, massOffset[1]+POSY,  //
                                massOffset[2] + POSZ );         //
                                                                //
//______________________________________________________________//


//      Create and attach the core geoms                        //
//______________________________________________________________//
coreGeom[ 0 ] = dCreateBox( mySpace, UCR*2.0, UCR*2.0, THICK ); //
coreGeom[ 1 ] = dCreateBox( mySpace, THICK, THICK, RISE );      //
coreGeom[ 2 ] = dCreateBox( mySpace, LCR*2.0, LCR*2.0, THICK ); //
                                                                //
// Attach the geoms to the bodies and set their offset positions//
dGeomSetBody(coreGeom[0], core);                                //
dGeomSetBody(coreGeom[1], core);                                //
dGeomSetBody(coreGeom[2], core);                                //
                                                                //
dGeomSetOffsetPosition( coreGeom[ 0 ], -massOffset[ 0 ],        //
        -massOffset[ 1 ],(RISE+THICK)*0.5 - massOffset[ 2 ]);   //
dGeomSetOffsetPosition( coreGeom[ 1 ],-massOffset[ 0 ],         //
        -massOffset[ 1 ],-massOffset[ 2 ]);                     //
dGeomSetOffsetPosition(coreGeom[ 2 ], -massOffset[ 0 ],         //
        -massOffset[ 1 ],-(RISE+THICK)*0.5 -massOffset[ 2 ]);   //
//______________________________________________________________//


corePositions[ 0 ] = dGeomGetPosition( coreGeom[ 0 ] ); // Upper platform
coreRotations[ 0 ] = dGeomGetRotation( coreGeom[ 0 ] );
```

```
corePositions[ 1 ] = dGeomGetPosition( coreGeom[ 1 ] ); // Vertical bar
coreRotations[ 1 ] = dGeomGetRotation( coreGeom[ 1 ] );

corePositions[ 2 ] = dGeomGetPosition( coreGeom[ 2 ] ); // Lower platform
coreRotations[ 2 ] = dGeomGetRotation( coreGeom[ 2 ] );


/// +━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━+
/// | ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻   CREATE UPPER LEGS  ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻ |
/// +━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━+

upperLegs[ 0 ] = dBodyCreate( world );  // leg 1: front right
upperLegs[ 1 ] = dBodyCreate( world );  // leg 2: front left
upperLegs[ 2 ] = dBodyCreate( world );  // leg 3: back left
upperLegs[ 3 ] = dBodyCreate( world );  // leg 4: back right

dMassSetZero( &upperLegsMass );                                 //
//      Create the upper legs                                   //  B
//______________________________________________________________//  U
                                                                //  I
dMassSetZero( &working );                                       //  L
// Leg starts out aligned with X axis                           //  D
//totalMass = ULL * LINDENS;                                    //
dMassSetBoxTotal( &working, ULM, ULL, THICK, THICK );           //  U
// Move so that base end is at origin                           //  P
dMassTranslate (&working, ULL*0.5, 0.0, 0.0 );                  //  P
dMassAdd( &upperLegsMass, &working );                           //  E
// Now create the rotational part                               //  R
totalMass = RBR * LINDENS;                                      //
dMassSetBoxTotal( &working, totalMass,                          //
                        THICK - 0.005, THICK - 0.005, RBR );    //  L
// Move it into position                                        //  E
dMassTranslate(&working, THICK*0.5,0.0,RBR*0.5+THICK*0.5);      //  G
dMassAdd( &upperLegsMass, &working );                           //
// Save this so we can position the core later                  //  R
massOffset[ 0 ] = upperLegsMass.c[ 0 ];                         //  I
massOffset[ 1 ] = upperLegsMass.c[ 1 ];                         //  G
massOffset[ 2 ] = upperLegsMass.c[ 2 ];                         //  I
// Move the mass so its CoM is at the origin                    //  D
dMassTranslate(&upperLegsMass, -upperLegsMass.c[0],             //
                -upperLegsMass.c[1],-upperLegsMass.c[2]);       //  B
// The bodies can all share one mass object, since it's never   //  O
// modified after being created.                                //  D
dBodySetMass( upperLegs[ 0 ], &upperLegsMass );                 //  Y
dBodySetMass( upperLegs[ 1 ], &upperLegsMass );                 //
dBodySetMass( upperLegs[ 2 ], &upperLegsMass );                 //  O
dBodySetMass( upperLegs[ 3 ], &upperLegsMass );                 //  B
// Move it back now that it's created                           //  J
dBodySetPosition( upperLegs[0],massOffset[0],massOffset[1],     //  E
                        massOffset[2] );                        //  C
dBodySetPosition( upperLegs[1],massOffset[0],massOffset[1],     //  T
                        massOffset[2] );                        //  S
dBodySetPosition( upperLegs[2],massOffset[0],massOffset[1],     //
                        massOffset[2] );                        //
dBodySetPosition( upperLegs[3],massOffset[0],massOffset[1],     //
                        massOffset[2] );                        //
                                                                //
//______________________________________________________________//


//      Create / attach upper leg and rotational link geoms     //
//______________________________________________________________//
// EVEN numbered geoms are the upper legs                       //
// ODD numbered geoms are the rotational linkages               //
upperLegsGeom[ 0 ] = dCreateBox( mySpace, ULL, THICK, THICK );  //
upperLegsGeom[ 1 ] = dCreateBox( mySpace, THICK - 0.005,        //
                                          THICK - 0.005, RBR ); //
upperLegsGeom[ 2 ] = dCreateBox( mySpace, ULL, THICK, THICK );  //
upperLegsGeom[ 3 ] = dCreateBox( mySpace, THICK - 0.005,        //
                                          THICK - 0.005, RBR ); //
upperLegsGeom[ 4 ] = dCreateBox( mySpace, ULL, THICK, THICK );  //
upperLegsGeom[ 5 ] = dCreateBox( mySpace, THICK - 0.005,        //
                                          THICK - 0.005, RBR ); //
upperLegsGeom[ 6 ] = dCreateBox( mySpace, ULL, THICK, THICK );  //
upperLegsGeom[ 7 ] = dCreateBox( mySpace, THICK - 0.005,        //
                                          THICK - 0.005, RBR ); //
                                                                //
dGeomSetBody( upperLegsGeom[ 0 ], upperLegs[ 0 ] );             //
dGeomSetBody( upperLegsGeom[ 1 ], upperLegs[ 0 ] );             //
dGeomSetBody( upperLegsGeom[ 2 ], upperLegs[ 1 ] );             //
dGeomSetBody( upperLegsGeom[ 3 ], upperLegs[ 1 ] );             //
dGeomSetBody( upperLegsGeom[ 4 ], upperLegs[ 2 ] );             //
dGeomSetBody( upperLegsGeom[ 5 ], upperLegs[ 2 ] );             //
dGeomSetBody( upperLegsGeom[ 6 ], upperLegs[ 3 ] );             //
dGeomSetBody( upperLegsGeom[ 7 ], upperLegs[ 3 ] );             //
                                                                //
dGeomSetOffsetPosition(upperLegsGeom[0],-massOffset[0]+ULL*0.5, //
                -massOffset[1],-massOffset[2]);                 //
                                                                //
dGeomSetOffsetPosition(upperLegsGeom[1],-massOffset[0]+THICK*0.5,//
                -massOffset[1],-massOffset[2]+(THICK+RBR)*0.5); //
                                                                //
dGeomSetOffsetPosition(upperLegsGeom[2],-massOffset[0]+ULL*0.5, //
                -massOffset[1],-massOffset[2]);                 //
                                                                //
dGeomSetOffsetPosition(upperLegsGeom[3],-massOffset[0]+THICK*0.5,//
                -massOffset[1],-massOffset[2]+(THICK+RBR)*0.5); //
                                                                //
dGeomSetOffsetPosition(upperLegsGeom[4],-massOffset[0]+ULL*0.5, //
                -massOffset[1],-massOffset[2]);                 //
                                                                //
dGeomSetOffsetPosition(upperLegsGeom[5],-massOffset[0]+THICK*0.5,//
                -massOffset[1],-massOffset[2]+(THICK+RBR)*0.5); //
                                                                //
dGeomSetOffsetPosition(upperLegsGeom[6],-massOffset[0]+ULL*0.5, //
                -massOffset[1],-massOffset[2]);                 //
                                                                //
dGeomSetOffsetPosition(upperLegsGeom[7],-massOffset[0]+THICK*0.5,//
                -massOffset[1],-massOffset[2]+(THICK+RBR)*0.5); //
                                                                //
                                                                //
//______________________________________________________________//


//      Position the upper legs                                 //
//______________________________________________________________//
///    LEG 1: RIGHT FRONT _____________________________________ //
xPrime[ 0 ] = cos( ULZA );                                      //
xPrime[ 1 ] = 0.0;                                              //
xPrime[ 2 ] = sin( ULZA );                                      //
                                                                //
yPrime[ 0 ] = 0.0;                                              //
yPrime[ 1 ] = 1.0;                                              //
yPrime[ 2 ] = 0.0;                                              //
                                                                //
zPrime[ 0 ] = -sin( ULZA );                                     //
zPrime[ 1 ] = 0.0;                                              //
zPrime[ 2 ] = cos( ULZA );;                                     //
// xRot and yRot are used to rotate each leg into place abt Z   //
xRot[ 0 ] = 0.7071;                                             //
xRot[ 1 ] = 0.7071;                                             //
xRot[ 2 ] = 0.0;                                                //
                                                                //
yRot[ 0 ] = -0.7071;                                            //
yRot[ 1 ] = 0.7071;                                             //
yRot[ 2 ] = 0.0;                                                //
                                                                //
// Copy this because it will get rotated around for all 4 legs  //
curCOM[ 0 ] = massOffset[ 0 ];                                  //
curCOM[ 1 ] = massOffset[ 1 ];                                  //
curCOM[ 2 ] = massOffset[ 2 ];                                  //
                // Rotate prime coords about Z axis             //
rotCoords_xy( xPrime, yPrime, zPrime, xRot, yRot );             //
kneeXPrimes[ 0 ][ 0 ] = xPrime[ 0 ];    // Save these for the   //
```

```
kneeXPrimes[ 0 ][ 1 ] = xPrime[ 1 ];    // lower legs, where    //
kneeXPrimes[ 0 ][ 2 ] = xPrime[ 2 ];    // they will be needed  //
                                        // again.               //
kneeYPrimes[ 0 ][ 0 ] = yPrime[ 0 ];                            //
kneeYPrimes[ 0 ][ 1 ] = yPrime[ 1 ];                            //
kneeYPrimes[ 0 ][ 2 ] = yPrime[ 2 ];                            //
                                                                //
kneeZPrimes[ 0 ][ 0 ] = zPrime[ 0 ];                            //
kneeZPrimes[ 0 ][ 1 ] = zPrime[ 1 ];                            //
kneeZPrimes[ 0 ][ 2 ] = zPrime[ 2 ];                            //
                                                                //
rotAxes( zPrime, yPrime, LRZA );                                //
                // xPrime, yPrime, and zPrime now give a local  //
                // coordinate system for the upper leg          //
rotV_xy( curCOM, xPrime, yPrime );                              //
                // With massOffset rotated and added to POS,    //
                // it gives the CoM for the rotated / translated//
                // upper leg.                                   //
dBodySetPosition(upperLegs[ 0 ],POSX+B1[ 0 ]+curCOM[ 0 ],       //
                POSY+B1[1]+curCOM[1],POSZ+B1[2]+curCOM[2]);     //
                                                                //
dRFrom2Axes( rotation, xPrime[ 0 ], xPrime[ 1 ], xPrime[ 2 ],   //
                yPrime[ 0 ], yPrime[ 1 ], yPrime[ 2 ] );        //
                                                                //
dBodySetRotation( upperLegs[ 0 ], rotation );                   //
                                                                //
rotBalls[0][0]=zPrime[0]*(RBR+THICK*0.5)+POSX+B1[0];            //
rotBalls[0][1]=zPrime[1]*(RBR+THICK*0.5)+POSY+B1[1];            //
rotBalls[0][2]=zPrime[2]*(RBR+THICK*0.5)+POSZ+B1[2];            //
                                                                //
vactBalls[0][0]=xPrime[0]*IBR+POSX+B1[0];                       //
vactBalls[0][1]=xPrime[1]*IBR+POSY+B1[1];                       //
vactBalls[0][2]=xPrime[2]*IBR+POSZ+B1[2];                       //
                                                                //
///    LEG 2: LEFT FRONT ______________________________________ //
// Regenerate these, since they get eaten each time             //
xPrime[ 0 ] = cos( ULZA );                                      //
xPrime[ 1 ] = 0.0;                                              //
xPrime[ 2 ] = sin( ULZA );                                      //
                                                                //
yPrime[ 0 ] = 0.0;                                              //
yPrime[ 1 ] = 1.0;                                              //
yPrime[ 2 ] = 0.0;                                              //
                                                                //
zPrime[ 0 ] = -sin( ULZA );                                     //
zPrime[ 1 ] = 0.0;                                              //
zPrime[ 2 ] = cos( ULZA );;                                     //
// xRot and yRot are used to rotate each leg into place abt Z   //
xRot[ 0 ] = -0.7071;                                            //
xRot[ 1 ] = 0.7071;                                             //
xRot[ 2 ] = 0.0;                                                //
                                                                //
yRot[ 0 ] = -0.7071;                                            //
yRot[ 1 ] = -0.7071;                                            //
yRot[ 2 ] = 0.0;                                                //
                                                                //
// Copy this because it will get rotated around for all 4 legs  //
curCOM[ 0 ] = massOffset[ 0 ];                                  //
curCOM[ 1 ] = massOffset[ 1 ];                                  //
curCOM[ 2 ] = massOffset[ 2 ];                                  //
                // Rotate prime coords about Z axis             //
rotCoords_xy( xPrime, yPrime, zPrime, xRot, yRot );             //
                                                                //
kneeXPrimes[ 1 ][ 0 ] = xPrime[ 0 ];    // Save these for the   //
kneeXPrimes[ 1 ][ 1 ] = xPrime[ 1 ];    // lower legs, where    //
kneeXPrimes[ 1 ][ 2 ] = xPrime[ 2 ];    // they will be needed  //
                                        // again.               //
kneeYPrimes[ 1 ][ 0 ] = yPrime[ 0 ];                            //
kneeYPrimes[ 1 ][ 1 ] = yPrime[ 1 ];                            //
kneeYPrimes[ 1 ][ 2 ] = yPrime[ 2 ];                            //
                                                                //
kneeZPrimes[ 1 ][ 0 ] = zPrime[ 0 ];                            //
```

```
kneeZPrimes[ 1 ][ 1 ] = zPrime[ 1 ];                          //
kneeZPrimes[ 1 ][ 2 ] = zPrime[ 2 ];                          //
rotAxes( zPrime, yPrime, LRZA );                              //
                // xPrime, yPrime, and zPrime now give a local  //
                // coordinate system for the upper leg          //
rotV_xy( curCOM, xPrime, yPrime );                            //
                // With massOffset rotated and added to POS,    //
                // it gives the CoM for the rotated / translated//
                // upper leg.                                   //
dBodySetPosition(upperLegs[ 1 ],POSX+B2[ 0 ]+curCOM[ 0 ],       //
                POSY+B2[1]+curCOM[1],POSZ+B2[2]+curCOM[2]);     //
                                                              //
dRFrom2Axes( rotation, xPrime[ 0 ], xPrime[ 1 ], xPrime[ 2 ],   //
                yPrime[ 0 ], yPrime[ 1 ], yPrime[ 2 ] );        //
                                                              //
dBodySetRotation( upperLegs[ 1 ], rotation );                 //
                                                              //
rotBalls[1][0]=zPrime[0]*(RBR+THICK*0.5)+POSX+B2[0];          //
rotBalls[1][1]=zPrime[1]*(RBR+THICK*0.5)+POSY+B2[1];          //
rotBalls[1][2]=zPrime[2]*(RBR+THICK*0.5)+POSZ+B2[2];          //
                                                              //
vactBalls[1][0]=xPrime[0]*IBR+POSX+B2[0];                     //
vactBalls[1][1]=xPrime[1]*IBR+POSY+B2[1];                     //
vactBalls[1][2]=xPrime[2]*IBR+POSZ+B2[2];                     //
                                                              //
///    LEG 3: LEFT REAR _______________________________________ //
// Regenerate these, since they get eaten each time           //
xPrime[ 0 ] = cos( ULZA );                                    //
xPrime[ 1 ] = 0.0;                                            //
xPrime[ 2 ] = sin( ULZA );                                    //
                                                              //
yPrime[ 0 ] = 0.0;                                            //
yPrime[ 1 ] = 1.0;                                            //
yPrime[ 2 ] = 0.0;                                            //
                                                              //
zPrime[ 0 ] = -sin( ULZA );                                   //
zPrime[ 1 ] = 0.0;                                            //
zPrime[ 2 ] = cos( ULZA );;                                   //
// xRot and yRot are used to rotate each leg into place abt Z  //
xRot[ 0 ] = -0.7071;                                          //
xRot[ 1 ] = -0.7071;                                          //
xRot[ 2 ] = 0.0;                                              //
                                                              //
yRot[ 0 ] = 0.7071;                                           //
yRot[ 1 ] = -0.7071;                                          //
yRot[ 2 ] = 0.0;                                              //
                                                              //
// Copy this because it will get rotated around for all 4 legs  //
curCOM[ 0 ] = massOffset[ 0 ];                                //
curCOM[ 1 ] = massOffset[ 1 ];                                //
curCOM[ 2 ] = massOffset[ 2 ];                                //
                // Rotate prime coords about Z axis             //
rotCoords_xy( xPrime, yPrime, zPrime, xRot, yRot );           //
kneeXPrimes[ 2 ][ 0 ] = xPrime[ 0 ];    // Save these for the   //
kneeXPrimes[ 2 ][ 1 ] = xPrime[ 1 ];    // lower legs, where    //
kneeXPrimes[ 2 ][ 2 ] = xPrime[ 2 ];    // they will be needed  //
                                        // again.               //
kneeYPrimes[ 2 ][ 0 ] = yPrime[ 0 ];                          //
kneeYPrimes[ 2 ][ 1 ] = yPrime[ 1 ];                          //
kneeYPrimes[ 2 ][ 2 ] = yPrime[ 2 ];                          //
                                                              //
kneeZPrimes[ 2 ][ 0 ] = zPrime[ 0 ];                          //
kneeZPrimes[ 2 ][ 1 ] = zPrime[ 1 ];                          //
kneeZPrimes[ 2 ][ 2 ] = zPrime[ 2 ];                          //
                                                              //
rotAxes( zPrime, yPrime, LRZA );                              //
                // xPrime, yPrime, and zPrime now give a local  //
                // coordinate system for the upper leg          //
rotV_xy( curCOM, xPrime, yPrime );                            //
                // With massOffset rotated and added to POS,    //
                // it gives the CoM for the rotated / translated//
                // upper leg.                                   //
```

```
dBodySetPosition(upperLegs[ 2 ],POSX+B3[ 0 ]+curCOM[ 0 ],       //
                POSY+B3[1]+curCOM[1],POSZ+B3[2]+curCOM[2]);     //
                                                                //
dRFrom2Axes( rotation, xPrime[ 0 ], xPrime[ 1 ], xPrime[ 2 ],   //
                yPrime[ 0 ], yPrime[ 1 ], yPrime[ 2 ] );        //
                                                                //
dBodySetRotation( upperLegs[ 2 ], rotation );                   //
                                                                //
rotBalls[2][0]=zPrime[0]*(RBR+THICK*0.5)+POSX+B3[0];            //
rotBalls[2][1]=zPrime[1]*(RBR+THICK*0.5)+POSY+B3[1];            //
rotBalls[2][2]=zPrime[2]*(RBR+THICK*0.5)+POSZ+B3[2];            //
                                                                //
vactBalls[2][0]=xPrime[0]*IBR+POSX+B3[0];                       //
vactBalls[2][1]=xPrime[1]*IBR+POSY+B3[1];                       //
vactBalls[2][2]=xPrime[2]*IBR+POSZ+B3[2];                       //
                                                                //
///    LEG 4: RIGHT REAR ______________________________________ //
// Regenerate these, since they get eaten each time             //
xPrime[ 0 ] = cos( ULZA );                                      //
xPrime[ 1 ] = 0.0;                                              //
xPrime[ 2 ] = sin( ULZA );                                      //
                                                                //
yPrime[ 0 ] = 0.0;                                              //
yPrime[ 1 ] = 1.0;                                              //
yPrime[ 2 ] = 0.0;                                              //
                                                                //
zPrime[ 0 ] = -sin( ULZA );                                     //
zPrime[ 1 ] = 0.0;                                              //
zPrime[ 2 ] = cos( ULZA );;                                     //
// xRot and yRot are used to rotate each leg into place abt Z   //
xRot[ 0 ] = 0.7071;                                             //
xRot[ 1 ] = -0.7071;                                            //
xRot[ 2 ] = 0.0;                                                //
                                                                //
yRot[ 0 ] = 0.7071;                                             //
yRot[ 1 ] = 0.7071;                                             //
yRot[ 2 ] = 0.0;                                                //
                                                                //
// Copy this because it will get rotated around for all 4 legs  //
curCOM[ 0 ] = massOffset[ 0 ];                                  //
curCOM[ 1 ] = massOffset[ 1 ];                                  //
curCOM[ 2 ] = massOffset[ 2 ];                                  //
                // Rotate prime coords about Z axis             //
rotCoords_xy( xPrime, yPrime, zPrime, xRot, yRot );             //
                                                                //
kneeXPrimes[ 3 ][ 0 ] = xPrime[ 0 ];    // Save these for the   //
kneeXPrimes[ 3 ][ 1 ] = xPrime[ 1 ];    // lower legs, where    //
kneeXPrimes[ 3 ][ 2 ] = xPrime[ 2 ];    // they will be needed  //
                                        // again.               //
kneeYPrimes[ 3 ][ 0 ] = yPrime[ 0 ];                            //
kneeYPrimes[ 3 ][ 1 ] = yPrime[ 1 ];                            //
kneeYPrimes[ 3 ][ 2 ] = yPrime[ 2 ];                            //
                                                                //
kneeZPrimes[ 3 ][ 0 ] = zPrime[ 0 ];                            //
kneeZPrimes[ 3 ][ 1 ] = zPrime[ 1 ];                            //
kneeZPrimes[ 3 ][ 2 ] = zPrime[ 2 ];                            //
rotAxes( zPrime, yPrime, LRZA );                                //
                // xPrime, yPrime, and zPrime now give a local  //
                // coordinate system for the upper leg          //
rotV_xy( curCOM, xPrime, yPrime );                              //
                // With massOffset rotated and added to POS,    //
                // it gives the CoM for the rotated / translated//
                // upper leg.                                   //
dBodySetPosition(upperLegs[ 3 ],POSX+B4[ 0 ]+curCOM[ 0 ],       //
                POSY+B4[1]+curCOM[1],POSZ+B4[2]+curCOM[2]);     //
                                                                //
dRFrom2Axes( rotation, xPrime[ 0 ], xPrime[ 1 ], xPrime[ 2 ],   //
                yPrime[ 0 ], yPrime[ 1 ], yPrime[ 2 ] );        //
                                                                //
dBodySetRotation( upperLegs[ 3 ], rotation );                   //
                                                                //
rotBalls[3][0]=zPrime[0]*(RBR+THICK*0.5)+POSX+B4[0];            //
```

```
rotBalls[3][1]=zPrime[1]*(RBR+THICK*0.5)+POSY+B4[1];            //
rotBalls[3][2]=zPrime[2]*(RBR+THICK*0.5)+POSZ+B4[2];            //
                                                                //
vactBalls[3][0]=xPrime[0]*IBR+POSX+B4[0];                       //
vactBalls[3][1]=xPrime[1]*IBR+POSY+B4[1];                       //
vactBalls[3][2]=xPrime[2]*IBR+POSZ+B4[2];                       //
                                                                //
//______________________________________________________________//


//      Get position / rotation pointers for upper legs         //
//__________________________________________________________________
        // Position and rotation of upper leg                   //
upperLegPositions[ 0 ] = dGeomGetPosition( upperLegsGeom[ 0 ] );//
upperLegRotations[ 0 ] = dGeomGetRotation( upperLegsGeom[ 0 ] );//
        // Position and rotation of rotational link             //
rotLinkPositions[ 0 ] = dGeomGetPosition( upperLegsGeom[ 1 ] ); //
rotLinkRotations[ 0 ] = dGeomGetRotation( upperLegsGeom[ 1 ] ); //
        // Position and rotation of upper leg                   //
upperLegPositions[ 1 ] = dGeomGetPosition( upperLegsGeom[ 2 ] );//
upperLegRotations[ 1 ] = dGeomGetRotation( upperLegsGeom[ 2 ] );//
        // Position and rotation of rotational link             //
rotLinkPositions[ 1 ] = dGeomGetPosition( upperLegsGeom[ 3 ] ); //
rotLinkRotations[ 1 ] = dGeomGetRotation( upperLegsGeom[ 3 ] ); //
        // Position and rotation of upper leg                   //
upperLegPositions[ 2 ] = dGeomGetPosition( upperLegsGeom[ 4 ] );//
upperLegRotations[ 2 ] = dGeomGetRotation( upperLegsGeom[ 4 ] );//
        // Position and rotation of rotational link             //
rotLinkPositions[ 2 ] = dGeomGetPosition( upperLegsGeom[ 5 ] ); //
rotLinkRotations[ 2 ] = dGeomGetRotation( upperLegsGeom[ 5 ] ); //
        // Position and rotation of upper leg                   //
upperLegPositions[ 3 ] = dGeomGetPosition( upperLegsGeom[ 6 ] );//
upperLegRotations[ 3 ] = dGeomGetRotation( upperLegsGeom[ 6 ] );//
        // Position and rotation of rotational link             //
rotLinkPositions[ 3 ] = dGeomGetPosition( upperLegsGeom[ 7 ] ); //
rotLinkRotations[ 3 ] = dGeomGetRotation( upperLegsGeom[ 7 ] ); //
                                                                //
                                                                //
//______________________________________________________________//


/// +▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬+
/// | ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻   CREATE LOWER LEGS  ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻ |
/// +▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬+

lowerLegs[ 0 ] = dBodyCreate( world );  // leg 1: front right
lowerLegs[ 1 ] = dBodyCreate( world );  // leg 2: front left
lowerLegs[ 2 ] = dBodyCreate( world );  // leg 3: back left
lowerLegs[ 3 ] = dBodyCreate( world );  // leg 4: back right

dMassSetZero( &lowerLegsMass );                                 //
//      Create the lower legs                                   //  B
//______________________________________________________________//  U
                                                                //  I
dMassSetZero( &working );                                       //  L
// Leg starts out aligned with X axis                           //  D
totalMass = LLL * LINDENS;                                      //
dMassSetBoxTotal( &working, totalMass, LLL, THICK, THICK );     //  L
// Move so that base end is at origin                           //  O
dMassTranslate (&working, LLL*0.5, 0.0, 0.0 );                  //  W
dMassAdd( &lowerLegsMass, &working );                           //  E
// Now add the foot sphere                                      //  R
                                                                //
dMassSetSphereTotal( &working, FBM, FBR  );                     //  L
// Move it into position                                        //  E
dMassTranslate( &working, LLL, 0.0, 0.0 );                      //  G
dMassAdd( &lowerLegsMass, &working );                           //
// Save this so we can position the core later                  //  R
massOffset[ 0 ] = lowerLegsMass.c[ 0 ];                         //  I
massOffset[ 1 ] = lowerLegsMass.c[ 1 ];                         //  G
massOffset[ 2 ] = lowerLegsMass.c[ 2 ];                         //  I
// Move the mass so its CoM is at the origin                    //  D
```

```
dMassTranslate(&lowerLegsMass, -lowerLegsMass.c[0],          //
                -lowerLegsMass.c[1],-lowerLegsMass.c[2]);     //  B
// The bodies can all share one mass object, since it's never  //  O
// modified after being created.                               //  D
dBodySetMass( lowerLegs[ 0 ], &lowerLegsMass );                //  Y
dBodySetMass( lowerLegs[ 1 ], &lowerLegsMass );                //
dBodySetMass( lowerLegs[ 2 ], &lowerLegsMass );                //  O
dBodySetMass( lowerLegs[ 3 ], &lowerLegsMass );                //  B
// Move it back now that it's created                          //  J
dBodySetPosition( lowerLegs[0],massOffset[0],massOffset[1],    //  E
                        massOffset[2] );                       //  C
dBodySetPosition( lowerLegs[1],massOffset[0],massOffset[1],    //  T
                        massOffset[2] );                       //  S
dBodySetPosition( lowerLegs[2],massOffset[0],massOffset[1],    //
                        massOffset[2] );                       //
dBodySetPosition( lowerLegs[3],massOffset[0],massOffset[1],    //
                        massOffset[2] );                       //
                                                               //
//_____________________________________________________________//


//      Create / attach lower leg and foot-ball geoms          //
//_____________________________________________________________//
// EVEN numbered geoms are the upper legs                      //
// ODD numbered geoms are the rotational linkages              //
lowerLegsGeom[ 0 ] = dCreateBox( mySpace, LLL, THICK, THICK ); //
lowerLegsGeom[ 1 ] = dCreateSphere( mySpace, FBR );            //
lowerLegsGeom[ 2 ] = dCreateBox( mySpace, LLL, THICK, THICK ); //
lowerLegsGeom[ 3 ] = dCreateSphere( mySpace, FBR );            //
lowerLegsGeom[ 4 ] = dCreateBox( mySpace, LLL, THICK, THICK ); //
lowerLegsGeom[ 5 ] = dCreateSphere( mySpace, FBR );            //
lowerLegsGeom[ 6 ] = dCreateBox( mySpace, LLL, THICK, THICK ); //
lowerLegsGeom[ 7 ] = dCreateSphere( mySpace, FBR );            //
                                                               //
dGeomSetBody( lowerLegsGeom[ 0 ], lowerLegs[ 0 ] );            //
dGeomSetBody( lowerLegsGeom[ 1 ], lowerLegs[ 0 ] );            //
dGeomSetBody( lowerLegsGeom[ 2 ], lowerLegs[ 1 ] );            //
dGeomSetBody( lowerLegsGeom[ 3 ], lowerLegs[ 1 ] );            //
dGeomSetBody( lowerLegsGeom[ 4 ], lowerLegs[ 2 ] );            //
dGeomSetBody( lowerLegsGeom[ 5 ], lowerLegs[ 2 ] );            //
dGeomSetBody( lowerLegsGeom[ 6 ], lowerLegs[ 3 ] );            //
dGeomSetBody( lowerLegsGeom[ 7 ], lowerLegs[ 3 ] );            //
        // The leg itself                                      //
dGeomSetOffsetPosition(lowerLegsGeom[0],-massOffset[0]+LLL*0.5, //
                -massOffset[1],-massOffset[2]);                //
        // The foot                                            //
dGeomSetOffsetPosition(lowerLegsGeom[1],-massOffset[0]+LLL,    //
                -massOffset[1],-massOffset[2]);                //
        // The leg itself                                      //
dGeomSetOffsetPosition(lowerLegsGeom[2],-massOffset[0]+LLL*0.5, //
                -massOffset[1],-massOffset[2]);                //
        // The foot                                            //
dGeomSetOffsetPosition(lowerLegsGeom[3],-massOffset[0]+LLL,    //
                -massOffset[1],-massOffset[2]);                //
        // The leg itself                                      //
dGeomSetOffsetPosition(lowerLegsGeom[4],-massOffset[0]+LLL*0.5, //
                -massOffset[1],-massOffset[2]);                //
        // The foot                                            //
dGeomSetOffsetPosition(lowerLegsGeom[5],-massOffset[0]+LLL,    //
                -massOffset[1],-massOffset[2]);                //
        // The leg itself                                      //
dGeomSetOffsetPosition(lowerLegsGeom[6],-massOffset[0]+LLL*0.5, //
                -massOffset[1],-massOffset[2]);                //
        // The foot                                            //
dGeomSetOffsetPosition(lowerLegsGeom[7],-massOffset[0]+LLL,    //
                -massOffset[1],-massOffset[2]);                //
                                                               //
                                                               //
//_____________________________________________________________//


//      Position lower leg bodies                              //
```

```
//______________________________________________________________//
        // Start by getting knee coords and local axes          //
        // Note that each knee is offset from its hip by a      //
        // distance of ULL, in the direction of kneeXPrimes[][] //
                        /// LEG 1: RIGHT FRONT                  //
kneeCoords[ 0 ][ 0 ] = B1[0]+(ULL+0.02)*kneeXPrimes[ 0 ][ 0 ];  //
kneeCoords[ 0 ][ 1 ] = B1[1]+(ULL+0.02)*kneeXPrimes[ 0 ][ 1 ];  //
kneeCoords[ 0 ][ 2 ] = B1[2]+(ULL+0.02)*kneeXPrimes[ 0 ][ 2 ];  //
                        /// LEG 2: LEFT FRONT                   //
kneeCoords[ 1 ][ 0 ] = B2[0]+(ULL+0.02)*kneeXPrimes[ 1 ][ 0 ];  //
kneeCoords[ 1 ][ 1 ] = B2[1]+(ULL+0.02)*kneeXPrimes[ 1 ][ 1 ];  //
kneeCoords[ 1 ][ 2 ] = B2[2]+(ULL+0.02)*kneeXPrimes[ 1 ][ 2 ];  //
                        /// LEG 3: LEFT REAR                    //
kneeCoords[ 2 ][ 0 ] = B3[0]+(ULL+0.02)*kneeXPrimes[ 2 ][ 0 ];  //
kneeCoords[ 2 ][ 1 ] = B3[1]+(ULL+0.02)*kneeXPrimes[ 2 ][ 1 ];  //
kneeCoords[ 2 ][ 2 ] = B3[2]+(ULL+0.02)*kneeXPrimes[ 2 ][ 2 ];  //
                        /// LEG 4: RIGHT REAR                   //
kneeCoords[ 3 ][ 0 ] = B4[0]+(ULL+0.02)*kneeXPrimes[ 3 ][ 0 ];  //
kneeCoords[ 3 ][ 1 ] = B4[1]+(ULL+0.02)*kneeXPrimes[ 3 ][ 1 ];  //
kneeCoords[ 3 ][ 2 ] = B4[2]+(ULL+0.02)*kneeXPrimes[ 3 ][ 2 ];  //
//      Rotate the stored upper-leg axes down at the knee by    //
//      the angle LLZA                                          //
rotAxes( kneeZPrimes[ 0 ], kneeXPrimes[ 0 ], LLZA );            //
rotAxes( kneeZPrimes[ 1 ], kneeXPrimes[ 1 ], LLZA );            //
rotAxes( kneeZPrimes[ 2 ], kneeXPrimes[ 2 ], LLZA );            //
rotAxes( kneeZPrimes[ 3 ], kneeXPrimes[ 3 ], LLZA );            //
                        /// LEG 1: RIGHT FRONT                  //
// Copy this because it will get rotated around for all 4 legs  //
curCOM[ 0 ] = massOffset[ 0 ];                                  //
curCOM[ 1 ] = massOffset[ 1 ];                                  //
curCOM[ 2 ] = massOffset[ 2 ];                                  //
rotV_xy( curCOM, kneeXPrimes[ 0 ], kneeYPrimes[ 0 ] );          //
                // With massOffset rotated and added to POS,    //
                // it gives the CoM for the rotated / translated//
                // lower leg.                                   //
dBodySetPosition( lowerLegs[ 0 ],                               //
                POSX + curCOM[ 0 ] + kneeCoords[ 0 ][ 0 ],      //
                POSY + curCOM[ 1 ] + kneeCoords[ 0 ][ 1 ],      //
                POSZ + curCOM[ 2 ] + kneeCoords[ 0 ][ 2 ] );    //
dRFrom2Axes( rotation, kneeXPrimes[ 0 ][ 0 ],                   //
        kneeXPrimes[ 0 ][ 1 ], kneeXPrimes[ 0 ][ 2 ],           //
        kneeYPrimes[ 0 ][ 0 ], kneeYPrimes[ 0 ][ 1 ],           //
        kneeYPrimes[ 0 ][ 2 ] );                                //
                                                                //
dBodySetRotation( lowerLegs[ 0 ], rotation );                   //
                        /// LEG 2: LEFT FRONT                   //
// Copy this because it will get rotated around for all 4 legs  //
curCOM[ 0 ] = massOffset[ 0 ];                                  //
curCOM[ 1 ] = massOffset[ 1 ];                                  //
curCOM[ 2 ] = massOffset[ 2 ];                                  //
rotV_xy( curCOM, kneeXPrimes[ 1 ], kneeYPrimes[ 1 ] );          //
                // With massOffset rotated and added to POS,    //
                // it gives the CoM for the rotated / translated//
                // lower leg.                                   //
dBodySetPosition( lowerLegs[ 1 ],                               //
                POSX + curCOM[ 0 ] + kneeCoords[ 1 ][ 0 ],      //
                POSY + curCOM[ 1 ] + kneeCoords[ 1 ][ 1 ],      //
                POSZ + curCOM[ 2 ] + kneeCoords[ 1 ][ 2 ] );    //
dRFrom2Axes( rotation, kneeXPrimes[ 1 ][ 0 ],                   //
        kneeXPrimes[ 1 ][ 1 ], kneeXPrimes[ 1 ][ 2 ],           //
        kneeYPrimes[ 1 ][ 0 ], kneeYPrimes[ 1 ][ 1 ],           //
        kneeYPrimes[ 1 ][ 2 ] );                                //
                                                                //
dBodySetRotation( lowerLegs[ 1 ], rotation );                   //
                        /// LEG 3: LEFT REAR                    //
// Copy this because it will get rotated around for all 4 legs  //
curCOM[ 0 ] = massOffset[ 0 ];                                  //
curCOM[ 1 ] = massOffset[ 1 ];                                  //
curCOM[ 2 ] = massOffset[ 2 ];                                  //
rotV_xy( curCOM, kneeXPrimes[ 2 ], kneeYPrimes[ 2 ] );          //
                // With massOffset rotated and added to POS,    //
                // it gives the CoM for the rotated / translated//
                // lower leg.                          //
dBodySetPosition( lowerLegs[ 2 ],                      //
                POSX + curCOM[ 0 ] + kneeCoords[ 2 ][ 0 ],     //
                POSY + curCOM[ 1 ] + kneeCoords[ 2 ][ 1 ],     //
                POSZ + curCOM[ 2 ] + kneeCoords[ 2 ][ 2 ] );   //
dRFrom2Axes( rotation, kneeXPrimes[ 2 ][ 0 ],          //
        kneeXPrimes[ 2 ][ 1 ], kneeXPrimes[ 2 ][ 2 ],  //
        kneeYPrimes[ 2 ][ 0 ], kneeYPrimes[ 2 ][ 1 ],  //
        kneeYPrimes[ 2 ][ 2 ] );                       //
                                                       //
dBodySetRotation( lowerLegs[ 2 ], rotation );          //
                        /// LEG 4: RIGHT REAR          //
// Copy this because it will get rotated around for all 4 legs  //
curCOM[ 0 ] = massOffset[ 0 ];                         //
curCOM[ 1 ] = massOffset[ 1 ];                         //
curCOM[ 2 ] = massOffset[ 2 ];                         //
rotV_xy( curCOM, kneeXPrimes[ 3 ], kneeYPrimes[ 3 ] );  //
                // With massOffset rotated and added to POS,    //
                // it gives the CoM for the rotated / translated//
                // lower leg.                          //
dBodySetPosition( lowerLegs[ 3 ],                      //
                POSX + curCOM[ 0 ] + kneeCoords[ 3 ][ 0 ],     //
                POSY + curCOM[ 1 ] + kneeCoords[ 3 ][ 1 ],     //
                POSZ + curCOM[ 2 ] + kneeCoords[ 3 ][ 2 ] );   //
dRFrom2Axes( rotation, kneeXPrimes[ 3 ][ 0 ],          //
        kneeXPrimes[ 3 ][ 1 ], kneeXPrimes[ 3 ][ 2 ],  //
        kneeYPrimes[ 3 ][ 0 ], kneeYPrimes[ 3 ][ 1 ],  //
        kneeYPrimes[ 3 ][ 2 ] );                       //
                                                       //
dBodySetRotation( lowerLegs[ 3 ], rotation );          //
                                                       //
//______________________________________________________________//

footBallRadius = FBR;

//      Get position / rotation pointers for lower legs         //
//__________________________________________________________________
        // Position and rotation of upper leg          //
lowerLegPositions[ 0 ] = dGeomGetPosition( lowerLegsGeom[ 0 ] );//
lowerLegRotations[ 0 ] = dGeomGetRotation( lowerLegsGeom[ 0 ] );//
        // Position and rotation of foot sphere        //
footBallPositions[ 0 ] = dGeomGetPosition( lowerLegsGeom[ 1 ] );//
//footBallRotations[ 0 ] = dGeomGetRotation(lowerLegsGeom[ 1 ]);//
        // Position and rotation of lower leg          //
lowerLegPositions[ 1 ] = dGeomGetPosition( lowerLegsGeom[ 2 ] );//
lowerLegRotations[ 1 ] = dGeomGetRotation( lowerLegsGeom[ 2 ] );//
        // Position and rotation of foot sphere        //
footBallPositions[ 1 ] = dGeomGetPosition( lowerLegsGeom[ 3 ] );//
//footBallRotations[ 1 ] = dGeomGetRotation(lowerLegsGeom[ 3 ]);//
        // Position and rotation of lower leg          //
lowerLegPositions[ 2 ] = dGeomGetPosition( lowerLegsGeom[ 4 ] );//
lowerLegRotations[ 2 ] = dGeomGetRotation( lowerLegsGeom[ 4 ] );//
        // Position and rotation of foot sphere        //
footBallPositions[ 2 ] = dGeomGetPosition( lowerLegsGeom[ 5 ] );//
//footBallRotations[ 2 ] = dGeomGetRotation(lowerLegsGeom[ 5 ]);//
        // Position and rotation of lower leg          //
lowerLegPositions[ 3 ] = dGeomGetPosition( lowerLegsGeom[ 6 ] );//
lowerLegRotations[ 3 ] = dGeomGetRotation( lowerLegsGeom[ 6 ] );//
        // Position and rotation of foot sphere        //
footBallPositions[ 3 ] = dGeomGetPosition( lowerLegsGeom[ 7 ] );//
//footBallRotations[ 3 ] = dGeomGetRotation(lowerLegsGeom[ 7 ]);//
//______________________________________________________________//


/// +━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━+
/// | ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻   PLACE JOINTS   ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻ |
/// +━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━+


//      CREATE and place the hip ball joints           //
//______________________________________________________________//
        // Create the hip joints                       //
```

```
hips[ 0 ] = dJointCreateBall( world, 0 );                       //
hips[ 1 ] = dJointCreateBall( world, 0 );                       //
hips[ 2 ] = dJointCreateBall( world, 0 );                       //
hips[ 3 ] = dJointCreateBall( world, 0 );                       //
        // Attach the hip joints to the body and legs           //
dJointAttach( hips[ 0 ], core, upperLegs[ 0 ] );                //
dJointAttach( hips[ 1 ], core, upperLegs[ 1 ] );                //
dJointAttach( hips[ 2 ], core, upperLegs[ 2 ] );                //
dJointAttach( hips[ 3 ], core, upperLegs[ 3 ] );                //
        // Set the ball joints to B1, B2, B3, B4                //
dJointSetBallAnchor(hips[0],B1[0]+POSX,B1[1]+POSY,B1[2]+POSZ ); //
dJointSetBallAnchor(hips[1],B2[0]+POSX,B2[1]+POSY,B2[2]+POSZ ); //
dJointSetBallAnchor(hips[2],B3[0]+POSX,B3[1]+POSY,B3[2]+POSZ ); //
dJointSetBallAnchor(hips[3],B4[0]+POSX,B4[1]+POSY,B4[2]+POSZ ); //
                                                                //
//______________________________________________________________//


//      CREATE and place the knee hinge joints                  //
//______________________________________________________________//
        // Create the knee joints                               //
knees[ 0 ] = dJointCreateHinge( world, 0 );                     //
knees[ 1 ] = dJointCreateHinge( world, 0 );                     //
knees[ 2 ] = dJointCreateHinge( world, 0 );                     //
knees[ 3 ] = dJointCreateHinge( world, 0 );                     //
        // Attach the knee joints                               //
dJointAttach( knees[ 0 ], upperLegs[ 0 ], lowerLegs[ 0 ] );     //
dJointAttach( knees[ 1 ], upperLegs[ 1 ], lowerLegs[ 1 ] );     //
dJointAttach( knees[ 2 ], upperLegs[ 2 ], lowerLegs[ 2 ] );     //
dJointAttach( knees[ 3 ], upperLegs[ 3 ], lowerLegs[ 3 ] );     //
        // Position the knee joints                             //
dJointSetHingeAnchor( knees[0], kneeCoords[0][0]+POSX,          //
                kneeCoords[0][1]+POSY, kneeCoords[0][2]+POSZ);  //
dJointSetHingeAnchor( knees[1], kneeCoords[1][0]+POSX,          //
                kneeCoords[1][1]+POSY, kneeCoords[1][2]+POSZ);  //
dJointSetHingeAnchor( knees[2], kneeCoords[2][0]+POSX,          //
                kneeCoords[2][1]+POSY, kneeCoords[2][2]+POSZ);  //
dJointSetHingeAnchor( knees[3], kneeCoords[3][0]+POSX,          //
                kneeCoords[3][1]+POSY, kneeCoords[3][2]+POSZ);  //
        // Set the hinge axes for the knee joints from the knee //
        // prime coords. Note that X' points toward the foot,   //
        // Y' points left and is the hinge axis, and Z' points  //
        // up and out of the knee.                              //
dJointSetHingeAxis( knees[ 0 ], kneeYPrimes[ 0 ][ 0 ],          //
        kneeYPrimes[ 0 ][ 1 ], kneeYPrimes[ 0 ][ 2 ] );         //
dJointSetHingeAxis( knees[ 1 ], kneeYPrimes[ 1 ][ 0 ],          //
        kneeYPrimes[ 1 ][ 1 ], kneeYPrimes[ 1 ][ 2 ] );         //
dJointSetHingeAxis( knees[ 2 ], kneeYPrimes[ 2 ][ 0 ],          //
        kneeYPrimes[ 2 ][ 1 ], kneeYPrimes[ 2 ][ 2 ] );         //
dJointSetHingeAxis( knees[ 3 ], kneeYPrimes[ 3 ][ 0 ],          //
        kneeYPrimes[ 3 ][ 1 ], kneeYPrimes[ 3 ][ 2 ] );         //
                                                                //
//______________________________________________________________//


/// +-----------------------------------------------------------------+
/// | ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻   CREATE ACTUATORS   ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻ ☻ |
/// +-----------------------------------------------------------------+


//______________________________________________________________//
// Actuator Param order:                                        //
//      0:      Base_X                                          //
//      1:      Base_Y                                          //
//      2:      Base_Z                                          //
//      3:      Tip_X                                           //
//      4:      Tip_Y                                           //
//      5:      Tip_Z                                           //
//      6:      Base_Normal_X                                   //
//      7:      Base_Normal_Y                                   //
//      8:      Base_Normal_Z                                   //
//      9:      Base_Rot_Axis_1_X                               //
//      10:     Base_Rot_Axis_1_Y                               //
//      11:     Base_Rot_Axis_1_Z                               //
//      12:     Maximum Offset                                  //
//      13:     Base mass                                       //
//      14:     Tip mass                                        //
//      15:     Base width                                      //
//      16:     Base height                                     //
//      17:     Tip width                                       //
//      18:     Tip height                                      //
//  NOTE: The length the actuator "starts" at is position zero! //
//______________________________________________________________//
//                                                              //
// Actuator is created in this state:                           //
// |-----|-----|-----|        ( divided into thirds )           //
//  ^     ^     ^     ^                                         //
//  |     |     |      Tip of inner slider                      //
//  |     |      Tip of outer sleeve                            //
//  |      Base of inner slider                                 //
//   Base of outer sleeve                                       //
//______________________________________________________________//


//      CREATE the actuators in-place                           //
//______________________________________________________________//
                       /// LEG 1: RIGHT FRONT                   //
// Base position                                                //
workingActuator[ 0 ] = UCR + POSX;                              //
workingActuator[ 1 ] = VAO + POSY;                              //
workingActuator[ 2 ] = RISE * 0.5 + POSZ;                       //
// Tip position                                                 //
workingActuator[ 3 ] = vactBalls[ 0 ][ 0 ];                     //
workingActuator[ 4 ] = vactBalls[ 0 ][ 1 ];                     //
workingActuator[ 5 ] = vactBalls[ 0 ][ 2 ];                     //
// Base mount normal                                            //
workingActuator[ 6 ] = 0.0;                                     //
workingActuator[ 7 ] = 0.0;                                     //
workingActuator[ 8 ] = -1.0;                                    //
// Base rot axis 1                                              //
workingActuator[ 9 ] = 1.0;                                     //
workingActuator[ 10 ] = 0.0;                                    //
workingActuator[ 11 ] = 0.0;                                    //
workingActuator[ 12 ] = -1.0;   // Max Offset < 0 means autoset //
workingActuator[ 13 ] = VABM;                                   //
workingActuator[ 14 ] = VATM;                                   //
workingActuator[ 15 ] = THICK*1.1;                              //
workingActuator[ 16 ] = THICK*1.1;                              //
workingActuator[ 17 ] = THICK*0.9;                              //
workingActuator[ 18 ] = THICK*0.9;                              //
                                                                //
vactBaseBox[ 0 ] = 0.666 * genActuator( workingActuator, world, //
        &myActuators[ 0 ], core, upperLegs[ 0 ], mySpace );     //
                                    // Note that this is where  //
vactTipBox[ 0 ] = vactBaseBox[ 0 ]; // the vact-box lengths     //
                                    // are generated / stored   //
// New base position                                            //
workingActuator[ 0 ] = VAO + POSX;                              //
workingActuator[ 1 ] = UCR + POSY;                              //
workingActuator[ 2 ] = RISE * 0.5 + POSZ;                       //
// Keep same tip position                                       //
// New BRA1                                                     //
workingActuator[ 9 ] = 0.0;                                     //
workingActuator[ 10 ] = 1.0;                                    //
workingActuator[ 11 ] = 0.0;                                    //
                                                                //
genActuator( workingActuator, world, &myActuators[ 1 ], core,   //
                                upperLegs[ 0 ], mySpace );      //
                       /// LEG 2: LEFT FRONT                    //
// Base position                                                //
workingActuator[ 0 ] = -VAO + POSX;                             //
workingActuator[ 1 ] = UCR + POSY;                              //
workingActuator[ 2 ] = RISE * 0.5 + POSZ;                       //
// Tip position                                                 //
```

```
workingActuator[ 3 ] = vactBalls[ 1 ][ 0 ];                     //
workingActuator[ 4 ] = vactBalls[ 1 ][ 1 ];                     //
workingActuator[ 5 ] = vactBalls[ 1 ][ 2 ];                     //
// New BRA1                                                     //
workingActuator[ 9 ] = 0.0;                                     //
workingActuator[ 10 ] = 1.0;                                    //
workingActuator[ 11 ] = 0.0;                                    //
                                                                //
genActuator( workingActuator, world, &myActuators[ 2 ], core,   //
                                upperLegs[ 1 ], mySpace );      //
// New base position                                            //
workingActuator[ 0 ] = -UCR + POSX;                             //
workingActuator[ 1 ] = VAO + POSY;                              //
workingActuator[ 2 ] = RISE * 0.5 + POSZ;                       //
// New BRA1                                                     //
workingActuator[ 9 ] = 1.0;                                     //
workingActuator[ 10 ] = 0.0;                                    //
workingActuator[ 11 ] = 0.0;                                    //
                                                                //
genActuator( workingActuator, world, &myActuators[ 3 ], core,   //
                                upperLegs[ 1 ], mySpace );      //
                        /// LEG 3: LEFT REAR                    //
// Base position                                                //
workingActuator[ 0 ] = -UCR + POSX;                             //
workingActuator[ 1 ] = -VAO + POSY;                             //
workingActuator[ 2 ] = RISE * 0.5 + POSZ;                       //
// Tip position                                                 //
workingActuator[ 3 ] = vactBalls[ 2 ][ 0 ];                     //
workingActuator[ 4 ] = vactBalls[ 2 ][ 1 ];                     //
workingActuator[ 5 ] = vactBalls[ 2 ][ 2 ];                     //
// New BRA1                                                     //
workingActuator[ 9 ] = 1.0;                                     //
workingActuator[ 10 ] = 0.0;                                    //
workingActuator[ 11 ] = 0.0;                                    //
                                                                //
genActuator( workingActuator, world, &myActuators[ 4 ], core,   //
                                upperLegs[ 2 ], mySpace );      //
// New base position                                            //
workingActuator[ 0 ] = -VAO + POSX;                             //
workingActuator[ 1 ] = -UCR + POSY;                             //
workingActuator[ 2 ] = RISE * 0.5 + POSZ;                       //
// New BRA1                                                     //
workingActuator[ 9 ] = 0.0;                                     //
workingActuator[ 10 ] = 1.0;                                    //
workingActuator[ 11 ] = 0.0;                                    //
                                                                //
genActuator( workingActuator, world, &myActuators[ 5 ], core,   //
                                upperLegs[ 2 ], mySpace );      //
                        /// LEG 4: RIGHT REAR                   //
// Base position                                                //
workingActuator[ 0 ] = VAO + POSX;                              //
workingActuator[ 1 ] = -UCR + POSY;                             //
workingActuator[ 2 ] = RISE * 0.5 + POSZ;                       //
// Tip position                                                 //
workingActuator[ 3 ] = vactBalls[ 3 ][ 0 ];                     //
workingActuator[ 4 ] = vactBalls[ 3 ][ 1 ];                     //
workingActuator[ 5 ] = vactBalls[ 3 ][ 2 ];                     //
// New BRA1                                                     //
workingActuator[ 9 ] = 0.0;                                     //
workingActuator[ 10 ] = 1.0;                                    //
workingActuator[ 11 ] = 0.0;                                    //
                                                                //
genActuator( workingActuator, world, &myActuators[ 6 ], core,   //
                                upperLegs[ 3 ], mySpace );      //
// New base position                                            //
workingActuator[ 0 ] = UCR + POSX;                              //
workingActuator[ 1 ] = -VAO + POSY;                             //
workingActuator[ 2 ] = RISE * 0.5 + POSZ;                       //
// New BRA1                                                     //
workingActuator[ 9 ] = 1.0;                                     //
workingActuator[ 10 ] = 0.0;                                    //
workingActuator[ 11 ] = 0.0;                                    //
```

```
                                                              //
genActuator( workingActuator, world, &myActuators[ 7 ], core,   //
                              upperLegs[ 3 ], mySpace );      //
                                                              //
//____________________________________________________________//


//      GET actuator geom positions for render                //
//____________________________________________________________//
actBasePositions[0] = dGeomGetPosition(myActuators[0].baseGeom);//
actBaseRotations[0] = dGeomGetRotation(myActuators[0].baseGeom);//
actBasePositions[1] = dGeomGetPosition(myActuators[1].baseGeom);//
actBaseRotations[1] = dGeomGetRotation(myActuators[1].baseGeom);//
actBasePositions[2] = dGeomGetPosition(myActuators[2].baseGeom);//
actBaseRotations[2] = dGeomGetRotation(myActuators[2].baseGeom);//
actBasePositions[3] = dGeomGetPosition(myActuators[3].baseGeom);//
actBaseRotations[3] = dGeomGetRotation(myActuators[3].baseGeom);//
actBasePositions[4] = dGeomGetPosition(myActuators[4].baseGeom);//
actBaseRotations[4] = dGeomGetRotation(myActuators[4].baseGeom);//
actBasePositions[5] = dGeomGetPosition(myActuators[5].baseGeom);//
actBaseRotations[5] = dGeomGetRotation(myActuators[5].baseGeom);//
actBasePositions[6] = dGeomGetPosition(myActuators[6].baseGeom);//
actBaseRotations[6] = dGeomGetRotation(myActuators[6].baseGeom);//
actBasePositions[7] = dGeomGetPosition(myActuators[7].baseGeom);//
actBaseRotations[7] = dGeomGetRotation(myActuators[7].baseGeom);//
//              TIP geoms                                     //
actTipPositions[0] = dGeomGetPosition( myActuators[0].tipGeom );//
actTipRotations[0] = dGeomGetRotation( myActuators[0].tipGeom );//
actTipPositions[1] = dGeomGetPosition( myActuators[1].tipGeom );//
actTipRotations[1] = dGeomGetRotation( myActuators[1].tipGeom );//
actTipPositions[2] = dGeomGetPosition( myActuators[2].tipGeom );//
actTipRotations[2] = dGeomGetRotation( myActuators[2].tipGeom );//
actTipPositions[3] = dGeomGetPosition( myActuators[3].tipGeom );//
actTipRotations[3] = dGeomGetRotation( myActuators[3].tipGeom );//
actTipPositions[4] = dGeomGetPosition( myActuators[4].tipGeom );//
actTipRotations[4] = dGeomGetRotation( myActuators[4].tipGeom );//
actTipPositions[5] = dGeomGetPosition( myActuators[5].tipGeom );//
actTipRotations[5] = dGeomGetRotation( myActuators[5].tipGeom );//
actTipPositions[6] = dGeomGetPosition( myActuators[6].tipGeom );//
actTipRotations[6] = dGeomGetRotation( myActuators[6].tipGeom );//
actTipPositions[7] = dGeomGetPosition( myActuators[7].tipGeom );//
actTipRotations[7] = dGeomGetRotation( myActuators[7].tipGeom );//
                                                              //
//____________________________________________________________//


//      CREATE Rotation actuators                             //
//____________________________________________________________//
                      /// LEG 1: RIGHT FRONT                  //
// Base position                                              //
workingActuator[ 0 ] = 2.0 * LCR + POSX;                      //
workingActuator[ 1 ] = POSY;                                  //
workingActuator[ 2 ] = RISE * 0.5 + POSZ;                     //
// Tip position                                               //
workingActuator[ 3 ] = rotBalls[ 0 ][ 0 ];                    //
workingActuator[ 4 ] = rotBalls[ 0 ][ 1 ];                    //
workingActuator[ 5 ] = rotBalls[ 0 ][ 2 ];                    //
// Base mount normal                                          //
workingActuator[ 6 ] = 0.0;                                   //
workingActuator[ 7 ] = 0.0;                                   //
workingActuator[ 8 ] = -1.0;                                  //
// Base rot axis 1                                            //
workingActuator[ 9 ] = 1.0;                                   //
workingActuator[ 10 ] = 0.0;                                  //
workingActuator[ 11 ] = 0.0;                                  //
workingActuator[ 12 ] = -1.0;   // Max Offset < 0 means autoset //
workingActuator[ 13 ] = RABM;                                 //
workingActuator[ 14 ] = RATM;                                 //
workingActuator[ 15 ] = THICK*1.1;                            //
workingActuator[ 16 ] = THICK*1.1;                            //
workingActuator[ 17 ] = THICK*0.9;                            //
workingActuator[ 18 ] = THICK*0.9;                            //
```

```
        // Generate the actuator:                              //
        // NOTE: The ractBaseBox and ractTipBox lengths are    //
        //       being returned by genActuator() here.         //
ractBaseBox[ 0 ] = 0.666 * genActuator( workingActuator, world, //
        &myActuators[ 8 ], core, upperLegs[ 0 ], mySpace );    //
ractTipBox[ 0 ] = ractBaseBox[ 0 ];                            //
                        /// LEG 2: LEFT FRONT                  //
// Base position                                               //
workingActuator[ 0 ] = POSX;                                   //
workingActuator[ 1 ] = 2.0 * LCR + POSY;                       //
workingActuator[ 2 ] = RISE * 0.5 + POSZ;                      //
// Tip position                                                //
workingActuator[ 3 ] = rotBalls[ 1 ][ 0 ];                     //
workingActuator[ 4 ] = rotBalls[ 1 ][ 1 ];                     //
workingActuator[ 5 ] = rotBalls[ 1 ][ 2 ];                     //
// BRA-1                                                       //
workingActuator[ 9 ] = 0.0;                                    //
workingActuator[ 10 ] = 1.0;                                   //
workingActuator[ 11 ] = 0.0;                                   //
        // Generate the actuator:                              //
genActuator( workingActuator, world, &myActuators[ 9 ], core,  //
                                upperLegs[ 1 ], mySpace );     //
                        /// LEG 3: LEFT REAR                   //
// Base position                                               //
workingActuator[ 0 ] = -2.0 * LCR + POSX;                      //
workingActuator[ 1 ] = POSY;                                   //
workingActuator[ 2 ] = RISE * 0.5 + POSZ;                      //
// Tip position                                                //
workingActuator[ 3 ] = rotBalls[ 2 ][ 0 ];                     //
workingActuator[ 4 ] = rotBalls[ 2 ][ 1 ];                     //
workingActuator[ 5 ] = rotBalls[ 2 ][ 2 ];                     //
// BRA-1                                                       //
workingActuator[ 9 ] = 1.0;                                    //
workingActuator[ 10 ] = 0.0;                                   //
workingActuator[ 11 ] = 0.0;                                   //
        // Generate the actuator:                              //
genActuator( workingActuator, world, &myActuators[ 10 ], core, //
                                upperLegs[ 2 ], mySpace );     //
                        /// LEG 4: RIGHT REAR                  //
// Base position                                               //
workingActuator[ 0 ] = POSX;                                   //
workingActuator[ 1 ] = -2.0 * LCR + POSY;                      //
workingActuator[ 2 ] = RISE * 0.5 + POSZ;                      //
// Tip position                                                //
workingActuator[ 3 ] = rotBalls[ 3 ][ 0 ];                     //
workingActuator[ 4 ] = rotBalls[ 3 ][ 1 ];                     //
workingActuator[ 5 ] = rotBalls[ 3 ][ 2 ];                     //
// BRA-1                                                       //
workingActuator[ 9 ] = 0.0;                                    //
workingActuator[ 10 ] = 1.0;                                   //
workingActuator[ 11 ] = 0.0;                                   //
        // Generate the actuator:                              //
genActuator( workingActuator, world, &myActuators[ 11 ], core, //
                                upperLegs[ 3 ], mySpace );     //
                                                               //
//_____________________________________________________________//


//      GET Rotation actuator body positions / rotations       //
//_____________________________________________________________//
actBasePositions[8] = dGeomGetPosition(myActuators[8].baseGeom);//
actBaseRotations[8] = dGeomGetRotation(myActuators[8].baseGeom);//
actBasePositions[9] = dGeomGetPosition(myActuators[9].baseGeom);//
actBaseRotations[9] = dGeomGetRotation(myActuators[9].baseGeom);//
actBasePositions[10]=dGeomGetPosition(myActuators[10].baseGeom);//
actBaseRotations[10]=dGeomGetRotation(myActuators[10].baseGeom);//
actBasePositions[11]=dGeomGetPosition(myActuators[11].baseGeom);//
actBaseRotations[11]=dGeomGetRotation(myActuators[11].baseGeom);//
actTipPositions[8] = dGeomGetPosition(myActuators[8].tipGeom);  //
actTipRotations[8] = dGeomGetRotation(myActuators[8].tipGeom);  //
actTipPositions[9] = dGeomGetPosition(myActuators[9].tipGeom);  //
actTipRotations[9] = dGeomGetRotation(myActuators[9].tipGeom);  //
```

```
actTipPositions[10]=dGeomGetPosition(myActuators[10].tipGeom);  //
actTipRotations[10]=dGeomGetRotation(myActuators[10].tipGeom);  //
actTipPositions[11]=dGeomGetPosition(myActuators[11].tipGeom);  //
actTipRotations[11]=dGeomGetRotation(myActuators[11].tipGeom);  //
                                                                //
//______________________________________________________________//

//cout << "Knee starting angles:\n";
//cout << dJointGetHingeAngle( knees[ 0 ] ) << "\n";
//cout << dJointGetHingeAngle( knees[ 1 ] ) << "\n";
//cout << dJointGetHingeAngle( knees[ 2 ] ) << "\n";
//cout << dJointGetHingeAngle( knees[ 3 ] ) << "\n";

dJointSetHingeParam( knees[ 0 ], dParamLoStop, -1.3 );
dJointSetHingeParam( knees[ 0 ], dParamHiStop, 1.3 );

dJointSetHingeParam( knees[ 1 ], dParamLoStop, -1.3 );
dJointSetHingeParam( knees[ 1 ], dParamHiStop, 1.3 );

dJointSetHingeParam( knees[ 2 ], dParamLoStop, -1.3 );
dJointSetHingeParam( knees[ 2 ], dParamHiStop, 1.3 );

dJointSetHingeParam( knees[ 3 ], dParamLoStop, -1.3 );
dJointSetHingeParam( knees[ 3 ], dParamHiStop, 1.3 );

//cout << "Knee starting angles after stops set:\n";
//cout << dJointGetHingeAngle( knees[ 0 ] ) << "\n";
//cout << dJointGetHingeAngle( knees[ 1 ] ) << "\n";
//cout << dJointGetHingeAngle( knees[ 2 ] ) << "\n";
//cout << dJointGetHingeAngle( knees[ 3 ] ) << "\n";

xPrime[ 0 ] = KBR * cos( LLZA );
xPrime[ 1 ] = -KBR * sin( LLZA );
xPrime[ 2 ] = 0.0;

yPrime[ 0 ] = xPrime[ 0 ] + ULL / 2.0;
yPrime[ 1 ] = xPrime[ 1 ];
yPrime[ 2 ] = 0.0;

KLL = getLength( yPrime );
kneeZeroAngle = LLZA;

}



///                                     ~spiderBody
///_____________________________________________________________________________
spiderBody::~spiderBody() {

//       DESTROY actuators                                      //
//______________________________________________________________//
delActuator( &myActuators[ 0 ] );                               //
delActuator( &myActuators[ 1 ] );                               //
delActuator( &myActuators[ 2 ] );                               //
delActuator( &myActuators[ 3 ] );                               //
delActuator( &myActuators[ 4 ] );                               //
delActuator( &myActuators[ 5 ] );                               //
delActuator( &myActuators[ 6 ] );                               //
delActuator( &myActuators[ 7 ] );                               //
delActuator( &myActuators[ 8 ] );                               //
delActuator( &myActuators[ 9 ] );                               //
delActuator( &myActuators[ 10 ] );                              //
delActuator( &myActuators[ 11 ] );                              //
//______________________________________________________________//


//       DESTROY body joints                                    //
//______________________________________________________________//
dJointDestroy( hips[ 0 ] );                                     //
dJointDestroy( hips[ 1 ] );                                     //
dJointDestroy( hips[ 2 ] );                                     //
dJointDestroy( hips[ 3 ] );                                   //
                                                              //
dJointDestroy( knees[ 0 ] );                                  //
dJointDestroy( knees[ 1 ] );                                  //
dJointDestroy( knees[ 2 ] );                                  //
dJointDestroy( knees[ 3 ] );                                  //
//____________________________________________________________//


//      DESTROY body core                                     //
//____________________________________________________________//
dGeomDestroy( coreGeom[ 0 ] );                                //
dGeomDestroy( coreGeom[ 1 ] );                                //
dGeomDestroy( coreGeom[ 2 ] );                                //
                                                              //
dBodyDestroy( core );                                         //
//____________________________________________________________//


//      DESTROY upper legs ( geoms then bodies )              //
//____________________________________________________________//
dGeomDestroy( upperLegsGeom[ 0 ] );                           //
dGeomDestroy( upperLegsGeom[ 1 ] );                           //
dGeomDestroy( upperLegsGeom[ 2 ] );                           //
dGeomDestroy( upperLegsGeom[ 3 ] );                           //
dGeomDestroy( upperLegsGeom[ 4 ] );                           //
dGeomDestroy( upperLegsGeom[ 5 ] );                           //
dGeomDestroy( upperLegsGeom[ 6 ] );                           //
dGeomDestroy( upperLegsGeom[ 7 ] );                           //
                                                              //
dBodyDestroy( upperLegs[ 0 ] );                               //
dBodyDestroy( upperLegs[ 1 ] );                               //
dBodyDestroy( upperLegs[ 2 ] );                               //
dBodyDestroy( upperLegs[ 3 ] );                               //
//____________________________________________________________//


//      DESTROY lower legs ( geoms then bodies )              //
//____________________________________________________________//
dGeomDestroy( lowerLegsGeom[ 0 ] );                           //
dGeomDestroy( lowerLegsGeom[ 1 ] );                           //
dGeomDestroy( lowerLegsGeom[ 2 ] );                           //
dGeomDestroy( lowerLegsGeom[ 3 ] );                           //
dGeomDestroy( lowerLegsGeom[ 4 ] );                           //
dGeomDestroy( lowerLegsGeom[ 5 ] );                           //
dGeomDestroy( lowerLegsGeom[ 6 ] );                           //
dGeomDestroy( lowerLegsGeom[ 7 ] );                           //
                                                              //
dBodyDestroy( lowerLegs[ 0 ] );                               //
dBodyDestroy( lowerLegs[ 1 ] );                               //
dBodyDestroy( lowerLegs[ 2 ] );                               //
dBodyDestroy( lowerLegs[ 3 ] );                               //
//____________________________________________________________//

}

///                                    spiderBody::addForce()
///_____________________________________________________________________________
void spiderBody::addForce( int i, dReal force ) {

dJointAddSliderForce( myActuators[ i ].slider , force);

}

///                                    spiderBody::addKneeTorque()
///_____________________________________________________________________________
void spiderBody::addKneeTorque( int i, dReal torque ) {

dJointAddHingeTorque( knees[ i ], torque );

}
```

```
///                                   spiderBody::getPos()
///_____________________________________________________________________________
dReal spiderBody::getPos( int i ) {

return( dJointGetSliderPosition( myActuators[ i ].slider ) );

}

///                                   spiderBody::getVel()
///_____________________________________________________________________________
dReal spiderBody::getVel( int i ) {

return( dJointGetSliderPositionRate( myActuators[ i ].slider ) );

}

///                                   spiderBody::getKneeAngle()
///_____________________________________________________________________________
dReal spiderBody::getKneeAngle( int i ) {

return( dJointGetHingeAngle( knees[ i ] ) + kneeZeroAngle );

}

///                                   spiderBody::getKneeOmega()
///_____________________________________________________________________________
dReal spiderBody::getKneeOmega( int i ) {

return( dJointGetHingeAngleRate( knees[ i ] ) );

}

///                                   spiderBody::getCore()
///_____________________________________________________________________________

dBodyID spiderBody::getCore() {

return( core );

}

///+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
///+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
///+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
///+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
///+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


///                                   calcKneeActOffset()
///_____________________________________________________________________________
dReal calcKneeActOffset( dReal theta, dReal KBR_s, dReal KLL ) {

dReal P[ 3 ];
dReal b, c; // , a;

dReal d, sol;

P[ 0 ] = KBR_s * cos( theta );
P[ 1 ] = -KBR_s * sin( theta );
P[ 2 ] = 0.0;

// a = 1.0;
b = -2.0 * P[ 0 ];
c = P[0]*P[0] + P[1]*P[1] - KLL*KLL;

//      x = [  -b +- sqrt( b^2 - 4*a*c )  ]   /   2*a

d = sqrt( b*b - 4.0*c );        // a is always 1.

sol = ( -b - d ) * 0.5;

if( sol > 0.0 ) { return( ( -b - d ) * 0.5 ); }
return( sol );

}

///                                 calcKneeTorque()
///_________________________________________________________________
//      This function calculates the torque at the knee joint, given a force in
//      the linear actuator driving the linkage.
dReal calcKneeTorque( dReal theta, dReal slidePos, dReal KBR_s, dReal f ) {

dReal P[ 2 ];
dReal linkVec[ 3 ];

dReal phi;

dReal fLink[ 3 ];
dReal flAbs;

P[ 0 ] = KBR_s * cos( theta );
P[ 1 ] = -KBR_s * sin( theta );
//P[ 2 ] = 0.0;

phi = atan( -P[ 1 ] / ( P[ 0 ] - slidePos ) );

linkVec[ 0 ] = P[ 0 ] - slidePos;
linkVec[ 1 ] = -P[ 1 ];
linkVec[ 2 ] = 0.0;

flAbs = f / cos( phi );

toUVec( linkVec );

fLink[ 0 ] = flAbs * linkVec[ 0 ];
fLink[ 1 ] = flAbs * linkVec[ 1 ];
fLink[ 2 ] = flAbs * linkVec[ 2 ];

return( fLink[ 1 ] * P[ 0 ] - fLink[ 0 ] * P[ 1 ] );

}

///                                 calcKneeActVel()
///_________________________________________________________________
dReal calcKneeActVel( dReal theta, dReal slidePos, dReal KBR_s, dReal omega ) {

dReal P[ 2 ];
dReal vB[ 2 ];
dReal linkVec[ 3 ];

dReal phi, vLink;

P[ 0 ] = KBR_s * cos( theta );
P[ 1 ] = -KBR_s * sin( theta );

vB[ 0 ] = -omega * KBR_s * sin( theta );
vB[ 1 ] = -omega * KBR_s * cos( theta );

phi = atan( -P[ 1 ] / ( P[ 0 ] - slidePos ) );

linkVec[ 0 ] = P[ 0 ] - slidePos;
linkVec[ 1 ] = -P[ 1 ];
linkVec[ 2 ] = 0.0;

toUVec( linkVec );

vLink = -( vB[0]*linkVec[0] + vB[1]*linkVec[1] );

return( vLink / cos( phi ) );

}
```

```
/*
      Actuator utility functions
            --Generates a linear servo to work with an ODE world
                                                                                       */

// baseMass is mass of motor end of actuator
// tipMass is mass of tip end of actuator

#define baseMass   param[ 13 ]
#define tipMass    param[ 14 ]
#define baseWidth  param[ 15 ]
#define baseHeight param[ 16 ]
#define tipWidth   param[ 17 ]
#define tipHeight  param[ 18 ]

struct actuator {
dBodyID base_body;
dBodyID tip_body;
dJointID slider;
dJointID tip_ball;
dJointID base_ujoint;
dMass mass1, mass2;
dGeomID baseGeom;       // Collision testing these two geoms against each other will
dGeomID tipGeom;        // result in epic fail (so don't do it).
};

///                                                           genActuator()
///==========================================================================

// Param order:
//      0:      Base_X
//      1:      Base_Y
//      2:      Base_Z
//      3:      Tip_X
//      4:      Tip_Y
//      5:      Tip_Z
//      6:      Base_Normal_X
//      7:      Base_Normal_Y
//      8:      Base_Normal_Z
//      9:      Base_Rot_Axis_1_X
//      10:     Base_Rot_Axis_1_Y
//      11:     Base_Rot_Axis_1_Z
//      12:     Maximum Offset
//      13:     Base mass
//      14:     Tip mass
//      15:     Base width
//      16:     Base height
//      17:     Tip width
//      18:     Tip height
//  NOTE: The length the actuator "starts" at is position zero!

// Actuator is created in this state:
// |-----|-----|-----|        ( divided into thirds )
//  ^     ^     ^     ^
//  |     |     |      Tip of inner slider
//  |     |      Tip of outer sleeve
//  |      Base of inner slider
//   Base of outer sleeve

#define localX uJointAxis2

dReal genActuator( dReal* param, dWorldID world, actuator* target, dBodyID baseMount, dBodyID tipMount,
dSpaceID gSpace ) {

dReal zeroLength;
dReal mainAxis[ 3 ];
dReal uJointAxis2[ 3 ];
//dReal localX[ 3 ];         // This is the same as uJointAxis2
dReal localY[ 3 ];
dReal base_bodyPos[ 3 ];
dReal tip_bodyPos[ 3 ];
```

```
dMatrix3 rotation;                      // Rotation matrix for bodies

// Calculate some stuff that will be needed later
mainAxis[ 0 ] = param[ 3 ] - param[ 0 ];        // Get main axis of actuator
mainAxis[ 1 ] = param[ 4 ] - param[ 1 ];
mainAxis[ 2 ] = param[ 5 ] - param[ 2 ];

scaleVec( mainAxis, 0.33334, base_bodyPos );
scaleVec( mainAxis, 0.66667, tip_bodyPos );

base_bodyPos[ 0 ] += param[ 0 ];
base_bodyPos[ 1 ] += param[ 1 ];
base_bodyPos[ 2 ] += param[ 2 ];

tip_bodyPos[ 0 ] += param[ 0 ];
tip_bodyPos[ 1 ] += param[ 1 ];
tip_bodyPos[ 2 ] += param[ 2 ];

zeroLength = getLength( mainAxis );                     // Get zero length of actuator
toUVec( mainAxis );                                     // Convert to unit vector
xProduct( &param[ 6 ], &param[ 9 ], uJointAxis2 );      // Get second uJoint axis
xProduct( uJointAxis2, mainAxis, localY );

dRFrom2Axes( rotation,mainAxis[0],mainAxis[1],mainAxis[2],localY[0],localY[1],localY[2] );

// Generate the mass models for the two bodies
dMassSetBoxTotal( &(target->mass1), baseMass, zeroLength * 0.667, baseWidth, baseHeight );
dMassSetBoxTotal( &(target->mass2), tipMass, zeroLength * 0.667, tipWidth, tipHeight );

// Generate the two internal bodies
target->base_body = dBodyCreate( world );
target->tip_body = dBodyCreate( world );

// Generate the collision geoms and attach them to the bodies
target->baseGeom = dCreateBox( gSpace, zeroLength * 0.667, baseWidth, baseHeight );
target->tipGeom = dCreateBox( gSpace, zeroLength * 0.667, tipWidth, tipHeight );
dGeomSetBody( target->baseGeom, target->base_body );
dGeomSetBody( target->tipGeom, target->tip_body );

// Attach the masses to the bodies
dBodySetMass( target->base_body, &(target->mass1) );
dBodySetMass( target->tip_body, &(target->mass2) );

// Position the bodies where they need to be
dBodySetPosition( target->base_body, base_bodyPos[ 0 ], base_bodyPos[ 1 ], base_bodyPos[ 2 ] );
dBodySetPosition( target->tip_body, tip_bodyPos[ 0 ], tip_bodyPos[ 1 ], tip_bodyPos[ 2 ] );
dBodySetRotation( target->base_body, rotation );
dBodySetRotation( target->tip_body, rotation );

// Generate the three joints and attach them
        // Generation
target->slider = dJointCreateSlider( world, 0 );
target->base_ujoint = dJointCreateUniversal( world, 0 );
target->tip_ball = dJointCreateBall( world, 0 );

dJointAttach( target->slider, target->base_body, target->tip_body );
dJointAttach( target->base_ujoint, baseMount, target->base_body );
dJointAttach( target->tip_ball, target->tip_body, tipMount );

dJointSetSliderAxis( target->slider, mainAxis[0], mainAxis[1], mainAxis[2] );
dJointSetBallAnchor( target->tip_ball, param[3], param[4], param[5] );
dJointSetUniversalAnchor( target->base_ujoint, param[0], param[1], param[2] );
dJointSetUniversalAxis1( target->base_ujoint, param[9], param[10], param[11] );
dJointSetUniversalAxis2( target->base_ujoint, uJointAxis2[0], uJointAxis2[1], uJointAxis2[2] );
        // Attach the joints to their respective bodies

if( param[ 12 ] < 0.0 ) {
        // If max offset (param 12) is negative, then auto-set it to be 1/3 of length
        dJointSetSliderParam ( target->slider, dParamLoStop, -0.333 * zeroLength );
        dJointSetSliderParam ( target->slider, dParamHiStop, 0.333 * zeroLength );
        }
```

```
else {
        dJointSetSliderParam ( target->slider, dParamLoStop, -param[ 12 ] );
        dJointSetSliderParam ( target->slider, dParamHiStop, param[ 12 ] );
        }

//dJointSetSliderParam ( target->slider, dParamVel, 0.0 );
//dJointSetSliderParam ( target->slider, dParamFMax, 800.0 );

return( zeroLength );

}


        // Positioning
//dGeomSetPosition( target.baseGeom, base_bodyPos[0], base_bodyPos[1], base_bodyPos[2] );
//dGeomSetPosition( target.tipGeom, tip_bodyPos[0], tip_bodyPos[1], tip_bodyPos[2] );
//dGeomSetRotation( target.baseGeom, rotation );
//dGeomSetRotation( target.tipGeom, rotation );
        // Attachment


///                                                          delActuator()
///===========================================================================
void delActuator( actuator* target ) {
dGeomDestroy( target->baseGeom );
dGeomDestroy( target->tipGeom );
dJointDestroy( target->slider );
dJointDestroy( target->base_ujoint );
dJointDestroy( target->tip_ball );
dBodyDestroy( target->base_body );
dBodyDestroy( target->tip_body );
}
```

REFERENCES

[1] G. Liu, M. Habib, K. Watanabe. “Central Pattern Generators Based on Matsuoka Oscillators For the Locomotion of Biped Robots.” *Artificial Life and Robotics*, vol. 12, issue 1-2, pp. 264-269, Mar 2008.

[2] H. Inada and K. Ishii. “Bipedal Walk Using a Central Pattern Generator.” *International Congress Series*, Vol. 1269-complete, August 2004.

[3] A. Kuo. “The Relative Roles of Feedforward and Feedback in the Control of Rhythmic Movements”. *Motor Control.* Vol. 6, 2002.

[4] Luk, B. L., S. Galt and S. Chen. “Using Genetic Algorithms to Establish Efficient Walking Gaits for an Eight-Legged Robot.” *International Journal of Systems Science.* Vol. 32, Issue: 6, pp. 703-713, 2001.

[5] A. Lewis, A. Fagg and G. Bekey. “Genetic Algorithms for Gait Synthesis in a Hexapod Robot.” In Y.F. Zheng, editor, Recent trends in mobile robots. *World Scientific*, 1993.

[6] Auer, P., Burgsteiner, H., and Maass, W. (2001). “A Learning Rule for Very Simple Universal Approximators Consisting of a Single Layer of Perceptrons.”*Neural Networks.* vol. 21, Issue 5, pp. 786-795, June, 2008.

[7] T. Weise. (June, 2008) *Global Optimization Algorithms Theory and Application, 2nd ed.* [online] http://www.it-weise.de

[8] R. Smith. (June, 2008) *Open Dynamics Engine*. Vers. 0.9x [online] https://opende.svn.sourceforge.net/svnroot/opende/trunk
(SVN Checkout, accessed 06 Jun. 2008)

[9] ---. *Gazebo*. Vers. 0.8-pre2, 05 Mar. 2008 [online] http://sourceforge.net/project/showfiles.php ?group_id=42445&package_id=90519&release_id=581798

[10] K. Demura. *Robot Simulation — Robot Programming With Open Dynamics Engine.* Morikita Publishing Co., Ltd., Tokyo, 2007.