2011

Punctuated Anytime Learning and the Xpilot-AI Combat Environment

Phillip Fritzsche
Connecticut College, phillip.fritzsche@conncoll.edu

Punctuated Anytime Learning and the Xpilot-AI Combat Environment

Phil Fritzsche; Advisor: Gary Parker

In this paper, research is presented on an application of Punctuated Anytime Learning with Fitness Biasing, a type of computational intelligence and evolutionary learning, for real-time learning of autonomous agents controllers in the space combat game Xpilot. Punctuated Anytime Learning was originally developed as a means of effective learning in the field of evolutionary robotics. An analysis was performed on the game environment to determine optimal environmental settings for use during learning, and Fitness Biasing is employed using this information to learn intelligent behavior for a video game agent controller in real-time. Xpilot-AI, an Xpilot add-on designed for testing learning systems, is used alongside evolutionary learning techniques to evolve optimal behavior in the background while periodic checks in normal game play are used to compensate for errors produced by running the system at a high frame rate. The resultant learned controllers are comparable to our best hand-coded Xpilot-AI agents, display complex behavior that resemble human strategies, and are capable of adapting to a changing enemy in real-time. The work presented in this paper is also general enough to further the development of Punctuated Anytime Learning in evolutionary robotic systems.

Table of Contents

Introduction

The objective of this research is to develop a robust method for real-time learning using evolutionary computation. The methods are applied to an autonomous agent in the Xpilot-AI game environment, but are applicable to the development of techniques that can be useful in advancing many areas of computational and artificial intelligence. An important issue faced by artificially intelligent systems is that they are limited in effectiveness without a method for being adaptable. They can often perform well in specific situations but do not have the ability to cope when an unanticipated element is introduced. Though the systems may be intelligent, their intelligence is only applicable to these specific situations. This issue makes the use of many such systems impractical in the real, or even virtual worlds, simply because of the unpredictable nature of any complex environment. It is impossible to predict everything that will happen in real life, so writing a program that can only work in specific situations can easily be rendered ineffective by a simple change that was not anticipated. A real-time learning system, such as the one being developed in this research, is capable of adapting to changes in its environment in real-time. Such systems constantly learn, so any significant changes that might affect the behavior of the system are recognized and taken into account in the learning process. This allows the system to stay up to date and effective in all situations, no matter how variable.

Real-time learning is of particular importance in the development of control systems for autonomous robots as well as in interactive video games. Autonomous robots capable of real-time learning can be used to help solve dozens of significant real-world problems. One of the main applications of autonomous robots is for exploration in unknown or dangerous situations, for example with space exploration or rescue situations. Consider a robot that, upon being sent to a new planet, was not only able to explore its surroundings but also adapt to the unknowns it

would likely encounter. In such an environment, there is no way the robot's software designers could account for every situation the robot could encounter simply because the designers are not familiar enough with the environment. For the robot to be effective in the long term, it is imperative that it learns quickly so that fast changing or unknown environments do not cause problems. Another beneficial use for robots capable of real-time learning is in disaster scenarios. In the aftermath of a large-scale natural disaster, search and rescue efforts are undertaken to find and aid those in danger. Would it not be beneficial to send in a robot, capable of adapting to any new dangers it may encounter in its environment, rather than sending in teams of people and putting more lives at risk?

Of interest in the video game industry is the ability to produce human-like agents for opponents. One characteristic of humans is their ability to learn and adapt to the manner of play of others in the game. This is a large part of what makes video games enjoyable. Creating artificially intelligent systems that are capable of doing this in real-time is an important aspect, as well as one of the most difficult aspects, in game development (Yannakakis and Hallam). Computer controlled game agents, labeled "bots", have existed for years as opponents, especially in combat-related games. It is only recently that they have started to behave somewhat intelligently, and there is still plenty of room for improvement.

To conduct research in methods for addressing this issue, Xpilot-AI, a complex testing environment for autonomous agent learning, was selected. Video games in general and Xpilot in particular provide a rigorous testing environment for artificial intelligence research. Its intricate nature has the capability to test agent control programs in different scenarios made possible by the variable parameters of the game.

In order to test methods for providing agent learning in real-time in an environment, it is beneficial to first show that a static learning system can produce a controller that behaves intelligently. This has been done in the Xpilot environment using genetic algorithms, a type of learning in the field of computational intelligence, to evolve parameters for a rule-based system (Parker and Parker, The evolution of multi-layer neural networks for the control of Xpilot agents). In this system, a rule-based system-controlled agent was modified to work with a genetic algorithm. The algorithm evolved optimal parameters for the rules of the rule-based system. Other methods have also been used to demonstrate learning in Xpilot. For example, a genetic algorithm was used to evolve the weights for an agent controlled by neural network, another type of computational intelligence, (Parker and Parker, Evolving parameters for Xpilot combat agents) and a cyclic genetic algorithm was used to directly evolve a control program for an agent (Parker and Parker, Evolving parameters for Xpilot combat agents). These learning systems, while useful, learned control behaviors before the agent was active and as a result had no means of adapting to changes in capabilities or the enemy's behavior once the agent was put into play.

Though these methods used a genetic algorithm to learn an effective controller, the final products are lacking the ability to adapt to changes in enemy behavior in real-time. In previous research, dynamic programming-based reinforcement learning techniques, such as Q-learning, were used to produce an Xpilot agent capable of real-time learning, but the controllers learned were for a very simple environment. Reinforcement learning is a type of machine learning that relies on training by way of exposure to different situations. Agents implemented in this manner are given rewards or disincentives for every action they take and the subsequent situations they end up in. These rewards are remembered and used for future decision-making. The Q-learning method requires an accurate model of the agent's environment to be successful. Since the Xpilot

combat environment is very complicated, this method was applied only to a limited scenario containing a single agent in a simple environment with no opponents (Allen, Dirmaier and Parker). Though it was a successful implementation of real-time learning, it was determined to not be as scalable as desired. As the complexity of the environment increases, such a system's ability to cope rapidly deteriorates. One possible solution for this issue is described by Lucas' paper on DynaQ, a form of Q-learning, which updates the agent's model of the environment as it explores. Doing so could allow an agent to more adeptly adapt to the large and constantly changing Xpilot environment. In another attempt to demonstrate real-time learning in Xpilot, evolutionary strategies, another type of computational intelligence learning, were used to learn agent controllers (Parker and Probst, Using evolutionary strategies for the real-time learning of controllers for autonomous agents in Xpilot-AI). Though capable of real-time learning, their reliance on mutations of a single chromosome to evolve led to slower learning and a system that was only partially effective.

For other games, a number of different strategies have been used to attempt real-time learning. For the DEFCON computer game, researchers applied decision-tree learning and case-based reasoning combined with simulated annealing methods with the intention of creating human-like behavior (Baumgarten, Colton and Morris). Others applied evolutionary techniques to neural networks by starting with simple networks then slowly adding nodes and connections while the game is running to make the agent learn increasingly complex behavior in real-time (Stanley, Bryant and Karpov). While both of these do learn in real-time, they also both rely on past knowledge and pre-defined courses of action. Others have used a genetic algorithm approach, attempting to learn competitive, human-like behavior in the video game Quake (Priesterjahn, Kramer and Weimer).

While useful, none of the above systems were able to fully solve the problem of creating a real-time learning system in the interactive game environment. Although genetic algorithms have been used successfully to generate controllers for interactive game environments, they have not been used in a real-time learning system since they require each individual of the population to be tested. In order to be able to adapt to an opponent's play in real-time, these tests would have to be done by playing the opponent. In addition, the genetic algorithm typically starts with a random population of solutions. This would make particularly poor play for someone attempting to enjoy the game. What is needed is to be able to do the learning on a model of the game with periodic checks to make sure the model represents the actual play.

In the study of learning in robotics, systems have been created to solve the issue of real-time learning with genetic algorithms. One such example is Anytime Learning developed by Grefenstette and Ramsey. Anytime Learning places a learning module in a robot and uses an observer module to learn from the robot's environment. The information gathered is used to influence the learning processes so that there is a link between the actual robot and environment to the simulation. As the learning system is on-board the robot, it has the potential to learn indefinitely. As long as the robot is running, it will continue to attempt to improve the controller.

Anytime Learning worked well for robots with the capability to carry the learning system on the robot. However, this is not always practical. For example, if the robot's environment is highly dangerous, it may be more practical to use several less expensive and expendable robots as opposed to one to complete the mission. Rather than having each robot carry an expensive on-board learning system, it would be more practical to have one off-board learning system used for all of the robots. One other issue with anytime learning is that the observer module had to recognize and categorize changes in the environment, a task that requires extensive computation.

Punctuated Anytime Learning (PAL), which was originally developed for evolving robot controllers (Parker, Punctuated Anytime Learning for Hexapod Gait Generation), is a modification of Anytime Learning that was developed to address these issues. In this paper, PAL is used to create a real-time learning system for control of agents in Xpilot-AI. One of the major benefits of PAL is its flexibility. In this work it is being applied to a learning problem pertaining to a video game combat environment, but that is not the only situation in which this system is useful. PAL has been implemented in the past on a robot, but the tests required a researcher to manually record measurements as a part of the system's learning process (Parker and Larochelle, Punctuated Anytime Learning for Evolutionary Robotics). PAL has never been implemented in a fully automated manner. One of the goals of this work is to do just that, and in doing so, help demonstrate the viability of this system not only for virtual environments but for real world ones as well. Showing that PAL is a practical automated system in an environment as complex as Xpilot is beneficial for evolutionary robotic research as well as video game research by helping show the realistic capabilities of the system.

Xpilot

Xpilot is an open source, multiplayer, two-dimensional space combat game. Players navigate triangular ships through custom-made maps collecting power ups, avoiding traps, and attempting to shoot down their opponents in order to be the last ship alive. It was first released in 1992 by a group of developers that included Bjørn Stabell, Ken Ronny Schouten, Bert Gÿsbers, and Dick Balaska (Stabell and Schouten). Since its initial release, the Xpilot project has been split into multiple development lines, each with their own differences in style and gameplay.

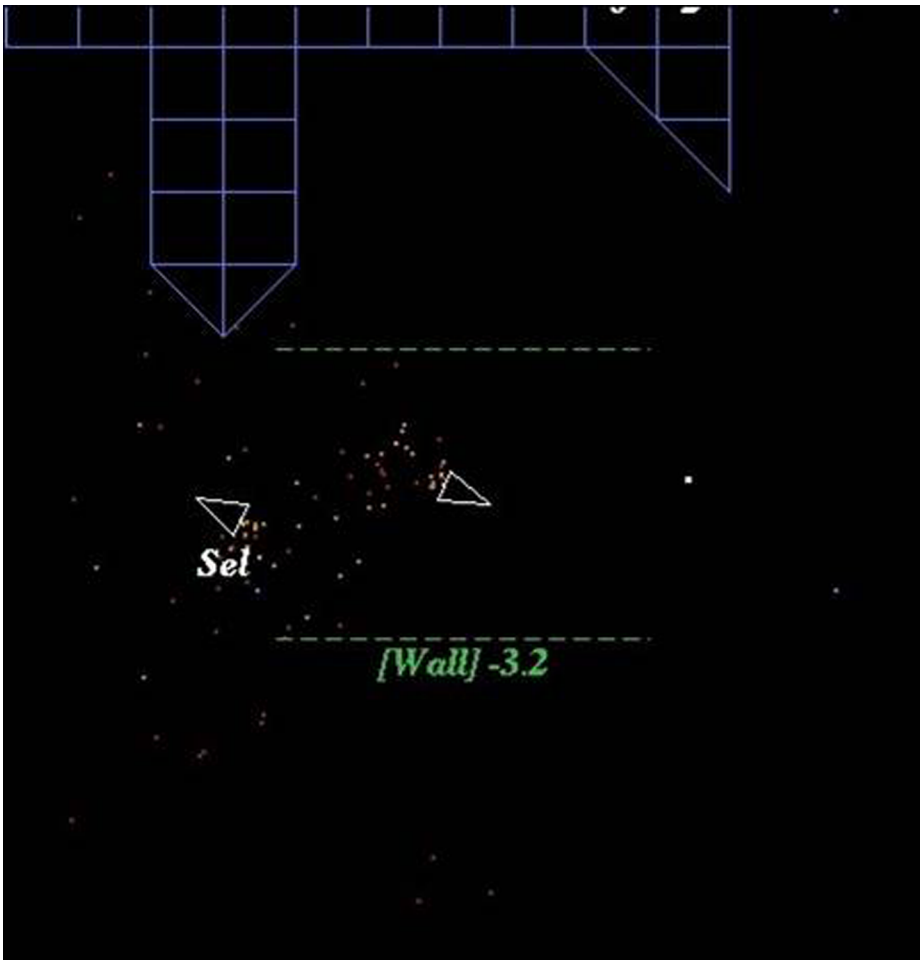

Figure 1 The Xpilot game environment.

Currently, the game is available for use on Windows, OS X, and Linux, as well as on the Apple iOS platform (XPilot Development).

Xpilot consists primarily of two components: the server and the client (Figure 1). The server is used to configure settings for a game. For example, it can change the number of frames per second (FPS) in a game and the map being used. It is also the server's responsibility to track the players playing the game, their scores, and other information. All communication between clients is routed through the server. Rather than having each client send information directly to each other, they send and receive information to and from the server. The centralized nature of the game helps to improve reliability and efficiency during gameplay involving multiple users.

To play the game, users control the client. The client is a simple application, capable of running on any operating system (Windows, Mac OS X, Linux, etc.), that when launched allows the user to control a single agent in the Xpilot environment. Specific keystrokes enable the users to control their ships by thrusting, turning, and shooting. Pressing shift, for example, thrusts, while the arrow keys on the keyboard turn the ship. Some of the Xpilot development lines include the ability to interact with various objects on the map – for example in Xpilot NG there are shields and bombs available on the map for use.

In order to join a game of Xpilot being hosted on the internet, the client can connect to the Xpilot meta server. The meta server is an informational server whose purpose is to maintain an up to date list of all active Xpilot game servers. From this list, the client can obtain the address of a specific server and connect to it to begin gameplay. Multiple clients from different locations can use the meta server to connect to the same game server. As a result, it is easy to have players from all around the world playing together on one server.

The Xpilot environment contains relatively realistic physics. Forces such as inertia and gravity are present and interact with agents on the map as one might expect in the real world. Agents explode if they run into a wall too fast, but will merely bounce off and lose some speed if they run into a wall at a slow speed. Since the environment is in a frictionless space setting, an agent can glide without continued thrusting and still maintain constant velocity. When an agent fires a bullet, it will not instantaneously hit its target regardless of distance but must travel there, allowing time for an opponent to dodge an incoming bullet. Further still, firing a bullet has an effect on the ship. When a ship fires, it is pushed backwards, or slowed down if it is moving fast enough. As a result, properly controlling the ship both in the correct direction and at the correct speed is difficult and requires skill and practice.

Xpilot is seen as a robust environment for artificial intelligence research for a number of reasons. The game has a large number of gameplay modes, each of which provide their own unique sets of challenges and scenarios. This flexibility provides programmers with great freedom when designing advanced tests for their agents. Xpilot requires minimal system resources. As the game was initially designed in 1992 and has not changed much since in regards to how much computation the game requires to run, any modern day computer can not only run the game easily but can run multiple clients and servers simultaneously. Because it can run multiple clients at once, Xpilot allows researchers to test multiple theories at the same time or run teams of agents all on one machine without requiring expensive supercomputers. It can also lead to a decrease in time required for an agent to learn – a computer that has multiple servers running can process multiple trials at the same time, allowing the algorithm to more quickly develop intelligent behavior. Thanks to the realistic nature of the physics present in the game, it is also a useful platform for developing autonomous robot learning systems, which are required for robots to function in highly complex environments.

Xpilot-AI

Xpilot-AI is an add-on to the Xpilot game that allows users to write programs to control Xpilot agents. It is one of the development lines created as a result of the initial Xpilot project. In its current form, it modifies Xpilot Classic, the original version of Xpilot, by building in the necessary functionality for these agent control programs to work. Programs can be used to implement computational and artificial intelligence techniques in the game. As a result, Xpilot-AI has become a powerful testing ground for researchers in these fields to develop learning methodologies. Agents controlled by programs can play along with other program-controlled

agents, humans, or server-controlled robots on any standard Xpilot server. To a user playing the game, a program-controlled agent appears no different than any other player on the server. This is not the case when dealing with pre-programmed agents that come as part of the Xpilot game, which use different ship icons during gameplay to differentiate themselves as simplistic computer coded agents.

The Xpilot-AI client, built into the standard Xpilot client, serves as a filter between the server and standard client. Normally, the server would send information directly to the client, which would then respond with the actions taken by the user at the keyboard. With Xpilot-AI installed, a program written by the researcher receives this information and is allowed to respond with actions as the client. This allows a script to programmatically send commands to the server, as if a user had been sitting at the keyboard controlling the agent. Xpilot-AI provides support for a number of common programming languages, specifically C, Java, Python, and Scheme. Programmers in each of these languages are given direct access to information required for making intelligent decisions as it is received from the server. This includes the agent's current location and bearing and the location of opponents on the map. All of the commonly used key presses a user might make are built into the Xpilot-AI library. The programmer defines a function, a specific block of code, that will be called once every frame of the game. In a game running at a standard FPS of 16, this means that the function will be called 16 times every second.

This provides an interesting opportunity to test autonomous agents by running them against both other computer-learned agents as well as human players. Seeing the agent perform in both situations can not only increase our understanding of what behavior is being learned but also make sure that it is able to compete against all types of opponents. Thanks to this capability,

combined with the fact that Xpilot itself is already an exceedingly complex environment, Xpilot-AI allows for rigorous testing of artificial intelligence techniques in a broad range of scenarios against a broad range of opponents. Because of the scope and flexibility with which the environment can be used effectively, it is the primary development tool for this research.

Rule-Based Systems

Rule-based systems, commonly known as expert systems, serve as a method of automating the problem-solving process in such a way as to emulate the know-how of human experts (Hayes-Roth). A rule-based system is comprised of a set of facts or assertions and a set of rules that specify how the system should react based on the current set of facts. The facts and assertions form the system's working memory – they represent all that the system knows about its environment that will help it make a decision to solve its current problem. The rules are a set of guides that, given the system's working memory, emulate the same behavior a human expert would follow in attempting to solve a similar problem. Each rule has two components: the antecedent and the consequent. The antecedent is a condition or conditions that, based on the system's working memory, evaluate to true or false. If all of the conditions in a particular rule's antecedent are true, then the rule is said to have been fired and the consequent occurs. The consequent can add to, change, or subtract from the working memory, cause the system to take an action, or both.

Each rule-based system contains a number of rules, potentially dozens if not more. One of the largest issues for such systems is the method with which they are used. Developers of these systems often overlap rules where appropriate. Though they will not have all of the same conditions necessary to take action, it is not uncommon to see rules that have a few of the same

conditions. This can lead to multiple rules having true firing conditions at the same time. It is possible that rules that fire could have conflicting consequents. One rule might say turn left while another rule might say turn right – obviously both of these cannot occur simultaneously, thus causing a problematic conflict. As a result, conflict resolution strategies must be implemented to insure that appropriate actions are taken based on the system's current working memory. There are five such strategies that are commonly used: first applicable, random, most specific, least recently used, and "best" rule (Rule-based systems and identification trees).

The first applicable resolution strategy is implemented by placing the system's rules in a specific order. The system goes through the set of rules from start to finish in the same order every time. The first rule found that is true is used, and no other rules further down the list are checked once one has been used. A system using a random conflict resolution strategy will randomly choose one of the rules that are found to be usable. If only one usable rule is found, then that rule is used. The most specific conflict resolution method fires whichever rule is the most specific. As different rules have different numbers of required conditions to fire, the rule that requires the highest number of true conditions is considered to be the most specific. The least recently used rule conflict resolution method chooses the rule that has fired the least frequently, in effect making sure that every rule is used close to the same number of times when a conflict occurs. Finally, the "best" rule conflict resolution method is implemented by assigning each rule in the system a specific weight that is used to determine its importance. When multiple rules are fired simultaneously, only the one with the heaviest weight is actually used.

Consider a simple rule-based system to control an Xpilot agent. Some of the primary concerns at any given moment for an agent to be used for combat are: where is the enemy, does the agent have a clear shot on the enemy, is the agent in danger of being hit by a bullet, and is the

agent in danger of running into a wall? A small sample set of rules for an agent to survive might look like this:

1. IF agent is in danger of being shot THEN turn 180º AND thrust.
2. IF agent is near a wall AND agent is moving too fast THEN turn 180º AND thrust.
3. IF agent can see its opponent AND has at least a good chance of hitting its enemy THEN calculate where to shoot AND turn in that direction AND shoot.
4. IF agent can see its opponent AND has at least a mediocre chance of hitting its enemy THEN turn towards opponent AND thrust.
5. IF the agent cannot see its opponent THEN pick a bearing not aimed at a wall AND thrust.

These rules are intended to be read in order and takes advantage of the first applicable method of conflict resolution mentioned above, though hard coded into that order is the method of most specific. Per the order defined in Figure 2, if starting at the node on top, the system would first gather information on its current state. Suppose that in its first frame, the opponent is not in danger of being shot, is not near a wall, and has a good chance of hitting its opponent. It would check rules one and two, in order, determining that their conditions for action are not true. Then once it hits rule three, it would see that the agent can see its opponent and has a good chance, causing rule three to fire. The agent would calculate an appropriate angle to turn and shoot. Rule four would not fire, though, even though it is equally as true as rule three, due to the conflict resolution method being used.

Suppose that the random method of conflict resolution was being used instead of first applicable. In this situation, it would be possible for the system to randomly select rule four, even though rule three would be a better choice in this current situation. As a result, the agent would thrust towards its enemy but would not shoot even though it has a clear shot. While the random method can be useful, this is one example in which it is not. For a rule-based system to be effective, the methods of implementation must be examined and designed to work well with each environment in which the system is being deployed. In this research, the first applicable method is used in order to give greater control over what actions the agent takes and when. Given that it is learning in a combat environment, consistency is more useful when the occasional act of randomness could result in the death of the agent.

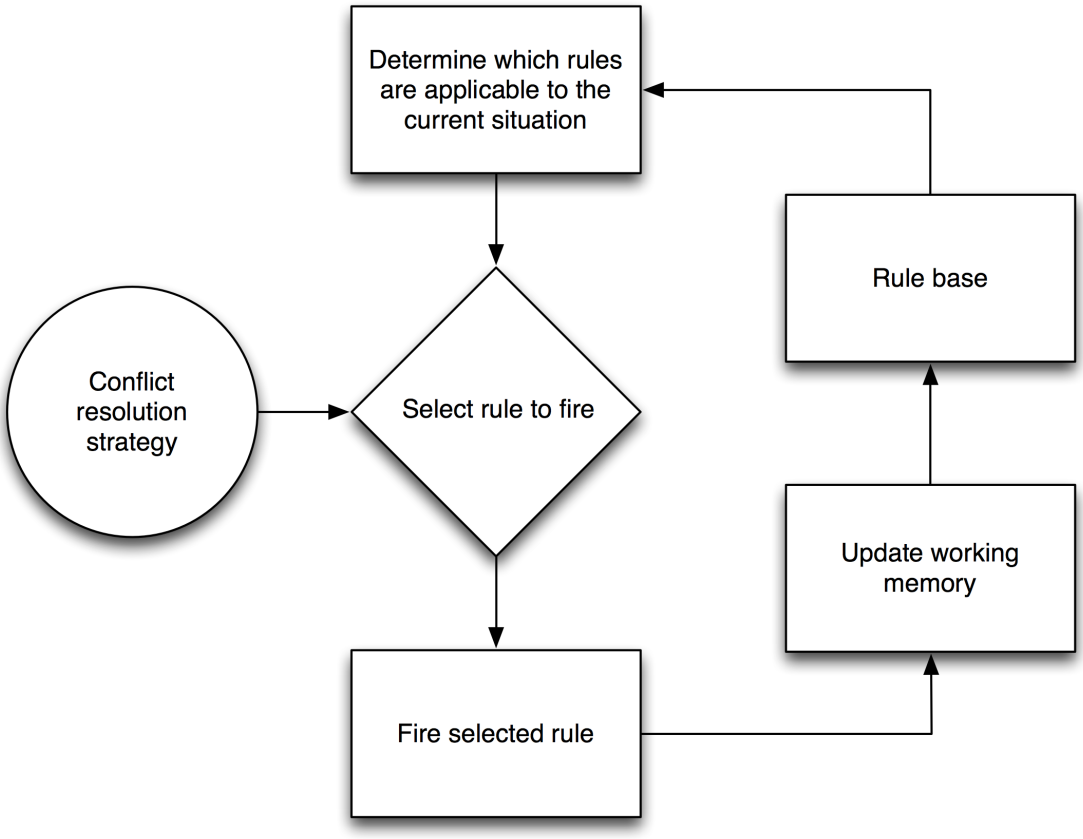

Figure 2 A visual representation of the actions taken by a rule-based system

Computational Intelligence

Computational intelligence describes three broad nature-inspired methodologies whose purpose is to solve complex problems that other arguably simpler and previously developed methods are unable to solve. The field of computational intelligence is a sub area of artificial intelligence and generally is utilized to solve the same problems. It primarily consists of fuzzy logic, neural networks, and evolutionary computation. Fuzzy logic, introduced in 1965 by L. A. Zadeh, is a tool used to formally represent the concept of human reasoning. Neural networks, initially introduced in the 1940s, though not fully developed until the 1980s, are designed to mimic the neural connections in the brain. Using mathematical processes modeled after the ways in which neurons connect and communicate with each other, neural networks attempt to computationally reproduce the signals neurons create. Evolutionary computation is a comprehensive term that is used to represent a number of different kinds of algorithms, all of which revolve around an iterative process designed to mimic the biological process of evolution.

As a whole, these three sub-fields that make up computational intelligence are areas of study for those wishing to design intelligent agents. Artificial intelligence, as a field, has to do with the intelligence of machines. It is defined as the "study and design of intelligent agents" (Poole, Mackworth and Goebel), where an intelligent agent is any kind of system that acts intelligently. The field, though it does promote the idea of systems that use active reasoning to do so, does not require it. Many subfields of artificial intelligence explicitly rely on the fact that their systems only appear intelligent, even if the processes driving such apparent intellect can not reason as a human would. One of the main hypotheses of computational intelligence, on the other hand, is that reasoning is fundamental to an intelligent system, as displayed by the nature of the three methodologies associated with the field.

A. Fuzzy Logic

Fuzzy logic is a subset of many-valued logic, a kind of propositional calculus, which is a formal system for representing logic and statements as propositions. In traditional propositional calculus, a statement may be either true or false; it deals with discrete terms. Many-valued logic deals with the idea that a statement may have more than two values of truth (true or false). Fuzzy logic is a system of reasoning where statements can have an infinite number of values in regards to truth. Fuzzy logic reasoning is dynamic rather than discrete. Consider the idea of describing a man's height. Using crisp logic, one might say a particular man is either tall or not. Applying such a limited view to an autonomous agent can be a bit cumbersome. In order for a computer to properly understand and identify the difference between a tall man and a short man, the computer must be given defined values for when the man becomes tall, or should no longer be considered so. Is 5' 11" considered tall? Or is the man not tall until he is over 6'? If 6' is considered the barrier, what happens to a man who is 5' 11.5"? He is, by this logic, not tall, but he is certainly not short either. Fuzzy logic exists to help represent the concept of, for example, a man being tall in such a way that a computer would be able to understand it while still making sense to a human as well. In a fuzzy logic based system, the man could be tall, very tall, short, or even of average height. Fuzzy logic sets are defined to represent the different categories possible. In this case, tall is one set, medium height is another, and short is a third. The man who is 6' 6" is entirely in the tall fuzzy set, while the man who is 5' 11" is tall, but also a member of the medium height fuzzy set. Fuzzy logic does not deal with discrete terms, allowing for greater flexibility and an unlimited number of ways to describe a particular concept. It allows a computer to both store exact data, such as the fact that the man is 5' 11.5" tall, while still being able to communicate

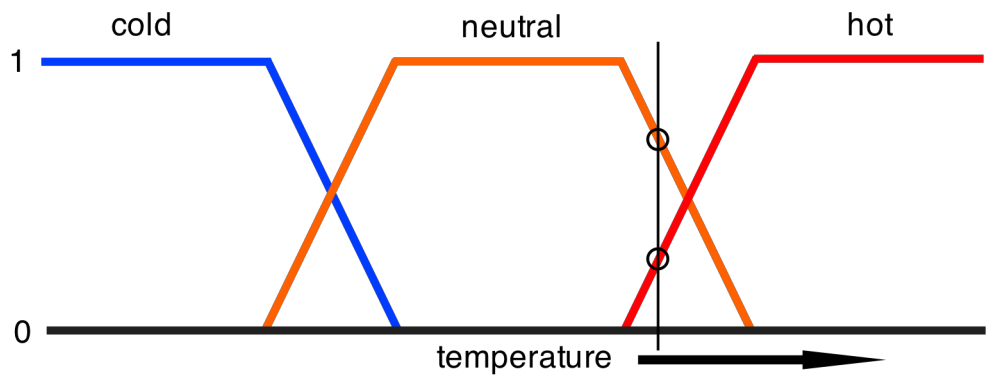

Figure 3 A sample fuzzy logic graph of temperatures from hot to cold

and reason in the abstract, by saying the man is somewhat tall, in a manner similar to that which we humans use.

Fuzzy logic works by breaking down a specific concept into the components it is made of. Consider the concept of a glass of water. The water can either be cold, neutral, or hot. In order to determine which it is, a fuzzy logic system must first break each of these values into a mathematical function, used to represent the varying degrees of truth to each value. Figure 3 shows an example of one such graph. The vertical line represents water that is somewhere in between neutral and hot. A fuzzy logic system would say that the glass of water at that temperature is 70% neutral, 30% hot, and 0% cold. Because of this, it could deduce that the glass is not cold, but is instead neutral and a little bit hot, or warm. Each temperature line on the graph represents a mathematical breakdown of the linguistic variables of this particular fuzzy logic system. Linguistic variables are the general values being used to describe the item in question – in this case, the linguistic variables being used are “cold”, “neutral”, and “hot”. Each linguistic variable is given its own mathematical function that represents the temperature range at which the water is 100% cold, neutral, or hot, as well as the range during which it becomes no longer so

and how fast the change occurs. This breakdown of temperature into ranges like cold, neutral, and hot can be extended further to create a more finely detailed gradient of temperatures. The linguistic variables could easily have been cold, cool, neutral, warm, hot, and so on.

Fuzzy systems are convenient in that they are easily adapted to work with other types of systems. Typically, a fuzzy logic system is implemented by way of a rule-based system. Consider a simple rule used to avoid walls for an Xpilot agent. It could be stated as follows: IF the agent is less than 100 pixels from the wall AND is traveling greater than 10 pixels per second THEN turn 180º AND thrust. The rule is relatively straightforward, but what if we could make it a bit simpler – suppose the rule was instead like this: IF the agent is near the wall AND is traveling fast THEN turn around AND thrust. Not only does this make more immediate sense when reading it, but it is also possible for the computer to understand it to. Using fuzzy logic, a rule based system could make decisions based on specifics without being forced to resort to exact numbers. This allows the system to be more flexible while maintaining the accuracy needed to make intelligent decisions. Unlike in a rule-based system, is possible to have more than one related rule fire in a fuzzy system. This way, when multiple conditions partially apply to a situation, as is the point with a fuzzy system, then their respective rules can each fire. Once this is done, the outputs are combined and a process called defuzzification is applied to deliver the outcome.

One point worthy of mention with fuzzy systems is that they are heavily influenced by their designers. Choices must be made as to when certain linguistic variables become true or false and how quickly these changes occur. Developers must determine at what temperature a glass of water is no longer considered to be cold – these concepts are subjective at best and as such the systems that result will often vary based on who developed them and for what purpose. That said,

this is not necessarily a bad or even a strange concept. It simply means that each fuzzy system will view things a little bit differently, which is precisely what happens when humans interact with each other. Further still, if consistency is required then all a developer must do is gather data on how others view the problem in question. Given a multitude of opinions, an average opinion can be found and applied to the fuzzy system. One of the main drawbacks of this system, though, also relates to the fact that the specific ways in which a fuzzy logic system is broken down. The fact that they are hard coded by their designers also means that these systems do not themselves have a method for learning. If the problem in question changes, the system will not change with it, effectively ending its usefulness. A lack of learning means that it must be regularly kept up to date to reflect changes in the environment, causing an arguably unnecessary increase in amount of work required to deploy such a system. A form of evolutionary computation can be used for learning in a fuzzy logic system, but using evolutionary computation for learning on a rule-based system is more straight forward and can typically result in acceptable solutions.

B. Neural Networks

The term neural network refers to a computational model whose design is motivated by the structure and functional aspects of the biological neural networks found in a brain. Neural networks are comprised of artificial neurons that have been connected together to promote communication for the purpose of solving a specific problem. Artificial neural networks are used to determine patterns in large amounts of data as well as find ways of representing complex data. There are several types of neural networks, some considerably more complex than others. Simple neural networks are often represented as having layers: the network will have *n* nodes in the input layer and *m* nodes in the output layer. The nodes in the input layer accept data then send

the data to the output layer, performing transformations on the way to generate the desired output. These transformations occur by multiplying the inputs by specific weights, associated with each neuron. For representing more complex data, additional layers are often used. These extra layers are called “hidden layers”, given that they are used by the system but do not directly interact with the user as they produce no viewable output of their own and only receive input from other neurons in the system.

One example of a simple neural network is the Perceptron, developed in 1957 by Frank Rosenblatt (Rosenblatt). The Perceptron is a single-layer neural network, meaning it has only one input layer and one output layer (Figure 4), and is capable of learning models for the simpler range of problems accessible by a neural network. It is limited to certain data models as it is only capable of producing output that is linearly related to the input data. A simple example of one

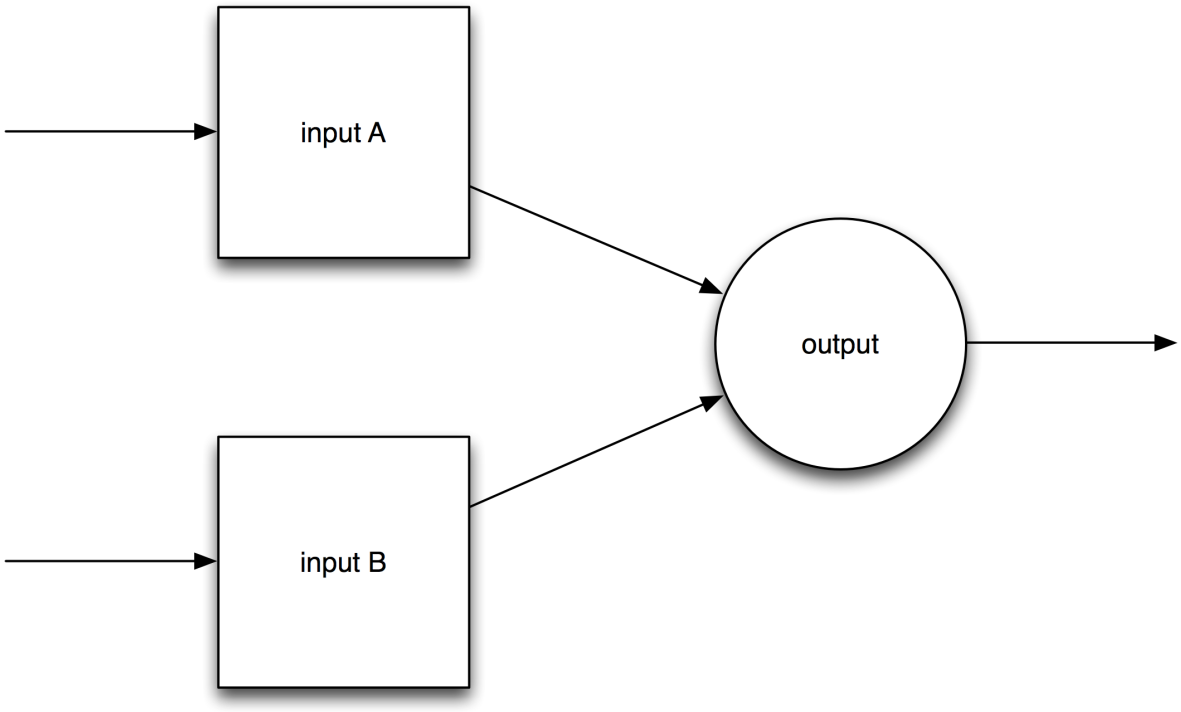

Figure 4 A visualization of the nodes in a single-layer Perceptron containing two inputs and one output neuron.

model the Perceptron could easily learn is the logical AND. This would be accomplished by implementing a Perceptron with two nodes in the input layer and one neuron on the output layer. If the two input nodes receive the inputs “true” and “true”, then the output would also be “true”. If, however, the inputs were “true” and “false”, then the definition of the logical AND states that the output should be “false”. The Perceptron would need to be trained, but, once it is, the system is self-sufficient and can respond with the appropriate output based on the training it previously received given any input.

The Perceptron (a single-layer network) as well as multi-layer networks, train by inputting data for which the output is known, using a method called delta learning. Delta learning works by calculating the difference between the actual output and the expected output for each output neuron in the system. It multiplies this information by the neuron’s input and again by a learning constant α. The resultant value is called the delta weight (Δw). The delta is added to the weight of the current neuron, and the sum of those two numbers is stored as the new weight for that neuron. Neural networks that contain multiple layers use a method called back propagation, of which delta learning is a specific case, to achieve learning. Once the output has been calculated and the weights for the output nodes are adjusted, the new information is propagated back through the network to the start, updating each neuron along the way. This process is repeated many times with large datasets in order to fully train a network. These datasets consist of several different expected input and output pairs.

Neural networks have been proven to be quite effective, but they do have their limitations in that they must be supplied with extensive training sets of data in order to effectively learn how to solve their given problem. For complex systems, or systems designed for unknown environments, this can be problematic, if not impossible, simply because it is not safe to assume that you will

always know what the output should be for any given problem. An ideal intelligent system would be able to, given no a priori knowledge, learn how to solve a problem. Unfortunately, neural networks are currently not capable of this. Though these systems have great potential, especially when combined with other computational intelligence techniques, in this regard neural networks are lacking.

C. Evolutionary Computation

Evolutionary computation is a relatively broad term. It covers a number of different computational intelligence techniques, including genetic programming, evolutionary strategies, and genetic algorithms, among others. Each of these methods are iterative in nature and are modeled after biological evolution. They each break down the problem such that the solutions can be represented as a chromosome. In many cases, this means turning the problem into a binary string. The system then creates a large set, or population, of randomly generated chromosomes to solve the problem. These individual solutions are then tested and evaluated so as to determine how well they perform. As they are tested, the system refines the population over time by removing weaker individuals and generating newer stronger ones by manipulating the already existing stronger individuals in the population. The ways in which the population is improved over time vary between each of the three methods mentioned above.

Genetic algorithms are one of the most common forms of evolutionary computation. The chromosomes that they work with are generally key numeric parameters that can be inserted into a control program for use. These parameters can vary greatly and subsequently affect greatly the quality of a particular solution to a problem. The algorithm refines these parameters over time, allowing the program they are used in to benefit as they improve. Genetic algorithms progress in large part due to the way they generate new individuals. Once every individual in the population

has been evaluated, the algorithm mimics sexual reproduction by joining together chromosomes to create offspring to be inserted into the next generation. Individuals are chosen at random from the current population to generate a new population that will replace the old, with the higher performing individuals getting a higher chance of being chosen. To generate an offspring from the two selected parents, the system uses a method called crossover. Two individuals are selected and merged together to produce one offspring containing part of the solution from both parents. The offspring is subjected to possible mutation and added to the new population. This process is repeated enough to replace every member in the current population. Genetic algorithms will be discussed in greater detail in the next section.

Genetic programming is similar to genetic algorithms in how the learning process happens, but instead tries to solve problems by generating entire functions rather than bits of information. Each individual of a population in a genetic programming system is a portion computer program potentially capable of solving the given task. The individuals are represented in a tree structure for ease of manipulation. Each individual is tested and once all of the individuals of a population have been tested, the refinement period begins. This works in much the same way as with genetic algorithms, with the difference primarily resting in how two selected parents are joined. To merge the two individuals, a branch from one individual is replaced with a branch of the other tree. The resultant individual is then inserted into the new generation. This process is repeated until the entire first generation has been replaced with a new one. At this point, the new generation is evaluated in the same manner as the old generation, and the whole process repeats itself with every generation improving over the last.

Evolutionary strategies are similar, but often rely on a different method for producing offspring that mimics asexual reproduction. Rather than crossing two parents together, single

parent mutation is used to create new children. When it is time to create a new individual, a mutation procedure goes through the individual randomly altering it in such a way as to emulate the unexpected mutation that can occur in real life. In general, mutation leads to changes that are negative but can occasionally cause a change that will spark a leap in quality.

Genetic Algorithms

A genetic algorithm is a programming technique designed to be a problem-solving method for complex problems by mimicking the biological process of evolution. It is a type of evolutionary computation, which is a type of computational intelligence, a subset of artificial intelligence. Genetic algorithms emulate the evolutionary process in a means that a computer can understand, allowing problems to be solved by evolving an optimal solution rather than trying to develop one outright. Standard genetic algorithms use chromosomes built of binary strings, ones and zeros, to represent particular parameters of a problem. These chromosomes are used to represent the building blocks of DNA. Just as DNA chromosomes are built of strands of adenine, cytosine, guanine, and thymine, the binary chromosomes employed by genetic algorithms are built of zeros and ones.

A. The Chromosome

To learn, the algorithm starts by generating a population of randomly created chromosomes. Population sizes can vary greatly from problem to problem, though a typical population size will be in the low 100s. In this research, each chromosome is divided up into sections of binary that are then evaluated by their binary values. For example, a chromosome of the form “01100001” could be used to represent two specific parameters. In this case, the first half, “0110”, would be used to represent the first parameter while the second half “0001” is used

to represent the second parameter. In decimal notation, the binary value “0110” equates to the number 6, while “0001” equates to the number 1. Choosing how to represent your problem’s solution as a string of binary is one of the most difficult design problems when developing a genetic algorithm-based system. In this research, the method above is used, though it is not the only solution. When dealing with robots, the different bits could pertain to signals to various legs or sensors or to timings on when to activate certain commands.

B. Genetic Operators

These chromosomes are then evaluated in a test environment representing the problem one by one to produce a fitness value for each. The fitness value is simply a measure of quality. For example if trying to learn a control program for an Xpilot agent, the chromosome that allows its agent to defeat its opponent the highest number of times will be awarded the highest fitness. Once the fitness has been determined for all chromosomes in the population, this information will be used for selection during the coming “mating” process.

Selection for the mating process can be handled in many different ways. In general, chromosomes are selected from the current generation at random, with a higher chance of being selected given to chromosomes that have a higher fitness. Two chromosomes are selected at a time. They then go through crossover and mutation to produce a new chromosome. Two example methods for selecting the chromosomes to be used for mating are the heroes method and the roulette wheel method. The heroes method randomly selects two chromosomes from the top performing 25% of all chromosomes and splices them together. The roulette wheel method stochastically selects two chromosomes. In this method, it is best imagined by placing each chromosome on a roulette wheel. Each chromosome’s slice of the wheel is then grown or shrunk to be relative to its fitness value in proportion to the sum of all fitnesses. A chromosome with a

very high fitness has a larger portion of the wheel while a chromosome with a very low fitness has a very small portion of the wheel. Once the two parents have been selected, they are mated together. The classic way to do this is called a single or double point crossover. In a single point crossover, a random point is selected in the chromosome. The algorithm then takes the section of one of the chromosomes up until this point and combines it with the section of the remaining chromosome from that point until the end of the chromosome. In a double point crossover, two points are selected rather than one. The start and end of one chromosome are then joined with the middle of the other chromosome.

After two parents have been mated together and produce a child, it goes through a mutation process. In a standard genetic algorithm, each bit of the chromosome has a small chance of being changed. This is to model the process of mutation that occurs naturally in the biological genome. In general, the changes made by mutation are small and often have detrimental effects. On occasion, though, the changes can lead to leaps in evolution that are beneficial to the population once introduced.

This selection, mating, and mutation process is then repeated until the entire population has been replaced. Each generated population is known as one generation. Each replacement of the population represents another new generation. A genetic algorithm will continue running through generations until some stop criterion has been reached. This stop criterion may happen when you have reached a specific target fitness or, if no target fitness is available, when the fitness levels off. Generally, genetic algorithms learn quickly at the start and plateau as generations go on, in logarithmic fashion. After a certain period of time, the learning gains are minimal when compared to the amount of time required to continue increasing. At this point, it is safe to stop running the algorithm as it will not learn much more. In Xpilot-AI, approximately

100 generations was generally enough to achieve an acceptable amount of learning. For some problems, a genetic algorithm may require thousands of generations to achieve optimal behavior, while with other problems this number may be as low as 50.

C. Genetic Algorithms Applied to Xpilot-AI

In this research, genetic algorithms were selected to be used for learning. The most effective method so far for writing genetic algorithm-based agents in Xpilot-AI has been by using rule-based systems for the framework. One of the benefits of such an agent is that rule-based systems must use numerical data to determine whether or not specific conditions are true. For example, is an agent considered to be in danger of being shot when a bullet is 100 pixels away, or is it only in danger once the bullet is 20 pixels away? Such questions are easily approximated, but it is not an easy task to find optimal values for such parameters without exorbitant amounts of trial and error by the researcher. As a result, using a genetic algorithm is ideal to determine the best values for the parameters to be used for the rules dictating the agent's behavior thanks to its iterative nature and ability to optimize numbers for large systems.

Because of the past success of rule-based systems using genetic algorithms to learn, this research uses such a system as a basis for testing real-time learning. In the first step of the research, an agent was developed solely as a rule-based system. It contained 31 rules and was inspired by the genetic algorithm-based agent present in Parker and Parker's "Evolving parameters for Xpilot combat agents." Upon completion, all numeric parameters present in the rules that had potential for optimization were removed and replaced with genetic algorithm-driven parameters. In effect, they were replaced with programmed placeholders to be filled in by the genetic algorithm as it learned. Figure 5 details the parameters used for this purpose.

• span – the angle between the line from the agent's nose to a target location and the edge of the nearest wall. Used to determine if the agent is blocked from a bullet by a wall.
• offset_inc – indicates the increments used to determine the optimal direction to turn to avoid crashing into a wall
• same_spread – the difference allowed between the distances returned by two wall feelers which would result in considering them equal.
• wall_span1 – the angle off the ship's track used to feel for the closest wall.
• wall_span2 – the angle off the ship's track used to feel for the second closest wall.
• vd_bullet_dist – determines the bullet alert value required to consider the bullet very dangerous.
• d_bullet_dist – determines the bullet alert value required to consider the bullet dangerous.
• vd_dodge_bullet_angle – the angle the ship will turn away from a bullet considered very dangerous in order to dodge it.
• d_dodge_bullet_angle – the angle the ship will turn away from a bullet considered dangerous in order to dodge it.
• close_wall_speed – the speed of the ship in relation to the distance to the closest wall. Used to determine if the ship should take action to avoid the wall.
• medium_wall_speed – similar to close_wall_speed, but for walls that are farther from the agent.
• c_angle_before_thrust – the angle of the ship's heading away from the closest wall before the ship will thrust.
• m_angle_before_thrust – similar to c_angle_before_thrust, but used in a rule with lower priority
• wall_avoid_angle – how small the angle has to be between the ship's heading and its desired track to avoid a wall before it will thrust.
• screen_thrust_speed – if the ship's speed is lower than this and it is turning to attack an enemy on the screen, it will thrust.
• radar_no_thrust_speed – if the ship's speed is lower than this and it is turning to attack an enemy on radar, it will thrust.
• ship_error_to_shoot – the maximum angular difference between the desired aim direction and the ship's heading before it will shoot at an enemy on the screen.
• radar_error_to_shoot – the maximum angular difference between the desired aim direction and the ship's heading before it will shoot at an enemy on radar.
• wall_turn_angleR – the angle between the ship's track and heading that the ship turns to avoid colliding with a wall when responding to a right feeler indicating a wall that is too close.
• wall_turn_angleL – the angle between the ship's track and heading that the ship turns to avoid colliding with a wall when responding to a left feeler indicating a wall that is too close.
• wall_turn_angleB – the angle between the ship's track and heading that the ship turns to avoid colliding with a wall when responding to an equal distance from both walls.
• shoot_dir_rand – the angular range that the ship will use to randomly affect its direction to aim.

Figure 5 A list of the parameters from the control program that the genetic algorithm learned for the Xpilot combat agent

When developing such a program to communicate with Xpilot, the developer must specify a block of code for the Xpilot-AI library to call. Once every frame, this block of code is run by Xpilot-AI. Because of this, the learning algorithm had to be set up to work on demand. When a chromosome needs to be evaluated, the chromosome is queued up in the program that interacts with Xpilot-AI. Upon the next reset of the game, which happens whenever an agent is killed, the new chromosome is loaded and evaluated for a certain period of time. Once the time has run out, the game sends the fitness score for the chromosome back to the learning algorithm, which records the information and continues onto the next chromosome.

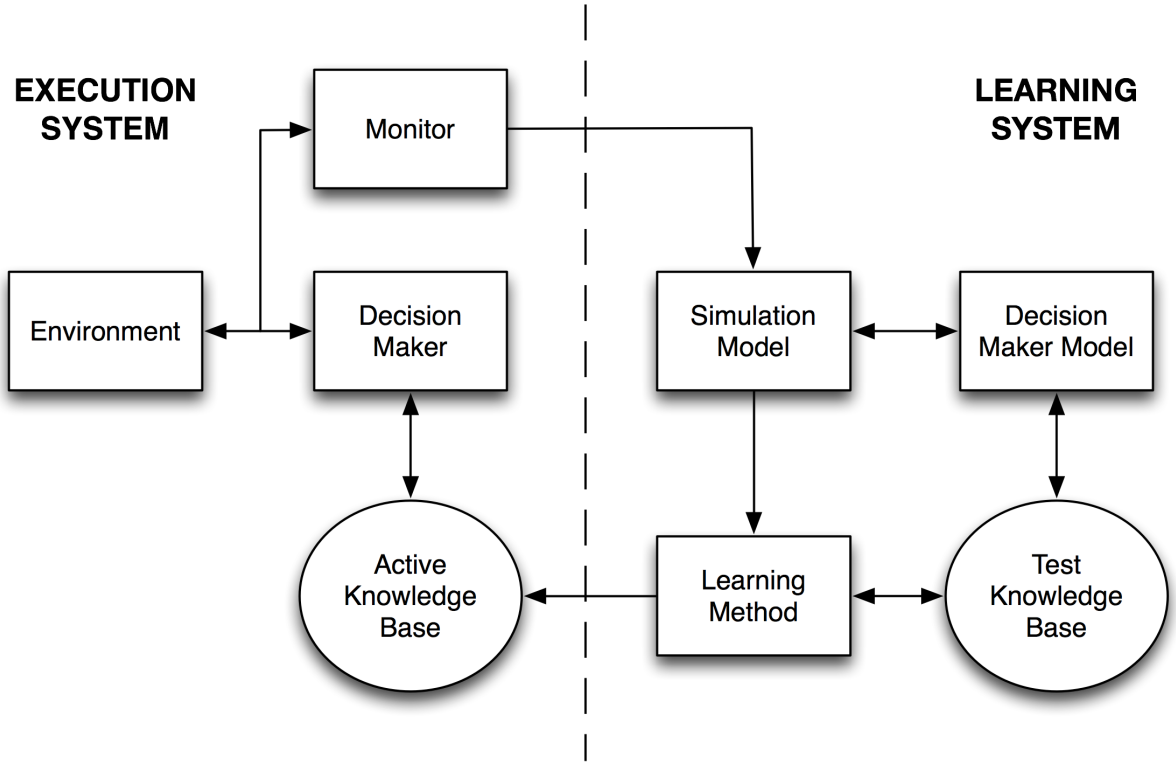

Figure 6 Anytime Learning System (replicated from Grefenstette and Ramsey)

Punctuated Anytime Learning

Punctuated Anytime Learning is a modification of Anytime Learning, a way of learning in evolutionary robotics. It is a method of automated learning for systems where the agent is not best served by housing the learning system as well as its control program on board the agent. Anytime Learning as a system, described for evolutionary robotics by Grefenstette and Ramsey, is split up into two major components: the execution system and the learning system (Figure 6). The execution system handles everything essential to running the agent in its environment. It runs the decision maker, which is responsible for deciding how the agent should react in a given situation. The knowledge base acts as a current strategy for the decision maker as it instructs the agent to execute actions in its environment. Also in the execution system is a monitor, which is

responsible for gathering information about the agent's environment. It identifies and remembers changes in the environment, then sends this information to the learning system.

The learning system uses the information sent to it by the execution system to alter the simulation model to more accurately represent the actual environment. A model of the decision maker also works in simulation to facilitate the learning process. It relies on a test knowledge base, separate from the active knowledge base in the execution system. The learning method works to increase the strength of the system and sends information to the active knowledge base whenever something new and improved is learned. Anytime Learning worked well for on-board learning in evolutionary robotics and was successful in actual tests. That said, one of the largest potential issues with this system is the monitor. Accurately and dynamically determining changes in the environment to update the simulation is a difficult task. It requires careful programming, well-performing sensors, and significant computational power. Another issue with Anytime Learning is that it requires the learning system to be onboard the robot. This is not always possible when it is desirable to have multiple less expensive robots perform the task. It is also often not ideal because robots break; legs, for example, can become weak and break with time. If the robot houses a large amount of expensive computing equipment, it is more wasteful when something does happen to the robot. Losing one of many small robots is better than losing one big robot, especially when it is the only one running.

Punctuated Anytime Learning, an extension of Anytime Learning, was developed to address these issues (Parker, Punctuated Anytime Learning for Hexapod Gait Generation). Rather than requiring the monitor to observe the entire environment and remember any changes, it is instead required only to track the performance of the agent. This simpler task leaves less room for error. In addition, the learning system is designed to be off-board the agent.

Disconnecting the two systems makes it easier to integrate additional agents to a single learning system and, as a result, increase the robustness of the system by allowing the same learning process to continue on even if one agent fails. There are two types of Punctuated Anytime Learning: Fitness Biasing and Co-Evolution of Model Parameters.

These two systems differ in one key factor: the way they handle the problem of trying to model the agent's environment. Currently, a computer model of the real world is not capable of representing every single detail in the real world. It cannot predict random phenomena and would require vast amounts of computational power to do nothing more than accurately keep a 3-D model of a robot's surroundings in the computer's memory for use during learning. The amount of work required to keep a model for use in learning entirely up to date is often prohibitive. Fitness Biasing, rather than trying to do that, simply keeps up to date the necessary information to insure effective learning without going through the work necessary to keep a complex model up to date. Co-Evolution of Model Parameters instead leverages the power learning capabilities present in the Punctuated Anytime Learning system to evolve the simulation's model of the environment with the hope of maintaining an accurate model without requiring excess work on the part of the developer.

A. Fitness Biasing

A Punctuated Anytime Learning system using Fitness Biasing works by keeping the fitnesses received in simulation in line with the fitnesses received while on the real world agent. It primarily exists to address the issues that arise from the fact that simulations are never perfect. Throughout the learning process, as described in the previous section on genetic algorithms, individual chromosomes that are used in learning are given fitness scores to evaluate their performance on a regular basis. In a Punctuated Anytime Learning system, all of the learning

happens in simulation while running on an imperfect model. As a result, the fitness scores that the learning system receives are rarely in sync with how well the agent would perform on the actual agent. To combat this, the Fitness Biasing system periodically tests out the learning system's current generation of individuals on the actual agent to analyze how well they perform in relation to the tests in the simulation. The differences between the fitnesses received in simulation and the fitnesses received on the actual agent are recorded by dividing the actual agent's fitness by the simulation agent's fitness. From that point on, whenever a chromosome receives a fitness in simulation, it is multiplied by the amount resulting from this division, thus altering the fitness by the same ratio as the one tested against the actual agent. When two agents are mated to produce a new offspring, their respective biases are averaged together to calculate a bias for the new chromosome. This allows the biases to be represented as the population evolves from generation to generation while stopping them from having too heavy of an influence on each chromosome. If a highly valued chromosome mates with a low valued chromosome, it would not be ideal for either one's bias value to continue through without averaging them. In either situation, the new chromosome would either be biased too heavily upwards or too heavily downwards.

B. Co-Evolution of Model Parameters

Co-Evolution of Model Parameters tries to tackle the same problem as Fitness Biasing but in quite a different way. While Fitness Biasing avoids the idea of updating the learning system's model, Co-Evolution of Model Parameters works to do just that. This method stores two populations, each learning independently: one representing solutions for the problem and the other representing the model parameters. The solutions evolved by the learning system are periodically tested on the actual agent in order to "co-evolve the accuracy of the robot's model"

(Parker, Punctuated Anytime Learning for Hexapod Gait Generation). Three agents (the best, the worst, and a randomly selected chromosome) are tested. Once they have been properly evaluated on the actual agent, each of the three individuals are tested on each of the individuals in the population learning the model parameters. The fitnesses of these agents are evaluated based on a comparison of three individuals' fitnesses on the actual agent and their fitnesses received against the individuals in the model parameter population.

Optimizing Xpilot-AI Learning Speeds

One of the most limiting factors in using Xpilot-AI for learning controllers is the amount of time required for a genetic algorithm to learn. All possible solutions in the population must be tested, which could require a significant amount of time. In Xpilot, the normal speed of play is 16 frames per second (FPS). In order to speed up learning in past research, Xpilot was run at 64 FPS as a genetic algorithm needs to perform many experiments in order to develop intelligent behavior. Observationally, 64 FPS appeared to be the fastest the system could run without encountering adverse effects on the quality of learning. A few attempts were made at 128 FPS, but the agent's performance was worse than expected. When compared to the agent learned at 64 FPS, it appeared considerably worse. Agents running at such high speeds lacked intelligence and coordination. In observing them, it seemed as if there was a communication gap between the client and the server, causing agents to not be able to properly respond to their current situation. Reactions were delayed, if they came at all. As a result, the learning system was learning on an inaccurate model of the environment. The Xpilot world at 128 FPS is simply not the same as it is at 16 FPS. Consequently, it was determined that frame rates above 64 FPS were not effective.

However, since experimental testing was not performed, part of this research was to analyze the effects of varying frame rates on Xpilot agent learning.

In an ideal situation, Xpilot would be able to be run at considerably higher speeds, thus improving the rate of learning while making better use of modern computers. Though a game learning at 64 FPS is a four-fold increase over the human-playable speed of 16 FPS, it can still take days for an agent to achieve near optimal behavior. At 64 FPS, Xpilot is not taking full advantage of the power of any modern day computer, which runs at low capacity even with multiple Xpilot servers and agents running simultaneously. For this research, a genetic algorithm evolving agents for combat play was tested at a sequence of increasing FPS to experimentally determine the realistic break point for increased speeds in Xpilot. The intent is to determine if 64 FPS is the top speed where an agent can learn with acceptable degradation.

In addition to exploring the effects of FPS to find the maximum feasible rate for use with a genetic algorithm, this will also help in the development of a Punctuated Anytime Learning system for real-time learning in Xpilot. The maximum acceptable speed for Punctuated Anytime Learning can be higher than for a typical genetic algorithm since it is designed to compensate for discrepancies in the simulation. Any oddities that arise from an increased FPS can be worked through just as any other discrepancies that might arise during learning. Determining the max FPS usable for Punctuated Anytime Learning will assist in the development of such a system by allowing the system to learn faster and as a result perform better. For the purposes of testing the various FPS, the genetic algorithm will be tested at higher FPS. Once the agent has optimized a control program for each of the tested FPS, all agents will be compared at the standard game speed of 16 FPS.

A. Experiment

In order to determine the best FPS for learning, a standard genetic algorithm was used to do learning at different speeds. The genetic algorithm optimized parameters in the rules for an expert system used to control an Xpilot combat agent. These parameters were shown previously in Figure 5. Code for this genetic algorithm and its corresponding Xpilot agent are shown in Appendices A and B. This type of learning was shown to be successful in the past (Parker and Parker, Evolving parameters for Xpilot combat agents) with the model agent running at 64 FPS. To calculate the fitness for a given control program, each agent was allowed to engage in combat for two minutes on an Xpilot server against a robust hand-coded expert agent named Sel. During its time in battle, the learning agent would receive one point of fitness for every frame it was alive and gain 1000 points of fitness for every time it killed its opponent. In addition, whenever it died, it would lose 20 seconds off of its total time.

The genetic algorithm was run on various Xpilot servers set at varying frames per second, starting at 16 FPS up through 1024 FPS. Each agent was allowed to learn for 115 generations. Once learning was completed, the best agents from the 50th and 100th generations were individually taken and placed against Sel while running on a 16 FPS server. This was done to determine if the results stayed consistent when playing on a normal server after learning. If the agents only appear to learn less while running at higher frame rates but still perform equally as well as those that learned at slower frame rates when placed in matches at the same FPS, then Xpilot would only have the appearance of a degradation of quality at higher frame rates. On the other hand, if those agents that appear to learn poorly at higher frame rates also perform poorly when placed in lower frames per seconds, then it is clearer that the higher frame rates have a negative effect on the quality of learning.

B. Results

Five test runs using five randomly generated populations at each of the tested frames per second were run for a total of 115 generations each. Tests included agents running at frames per seconds ranging from 16 through 1024. Figure 7 shows the fitness of the agents over time as they evolved at their respective frames per seconds. Based on the data, it can be observed that there was a great discrepancy in the quality of learning as the FPS changed. As the FPS increased, the amount learned by the agents drastically decreased. For the agents running at 16 FPS, their fitnesses grew at an average of 18.6 points per generation. Meanwhile, the agents running at 1024 FPS actually lost fitness by the end, shrinking at a rate of 0.213 fitness per generation on average as can be observed on the graph. The agents learning at 128 FPS and above all

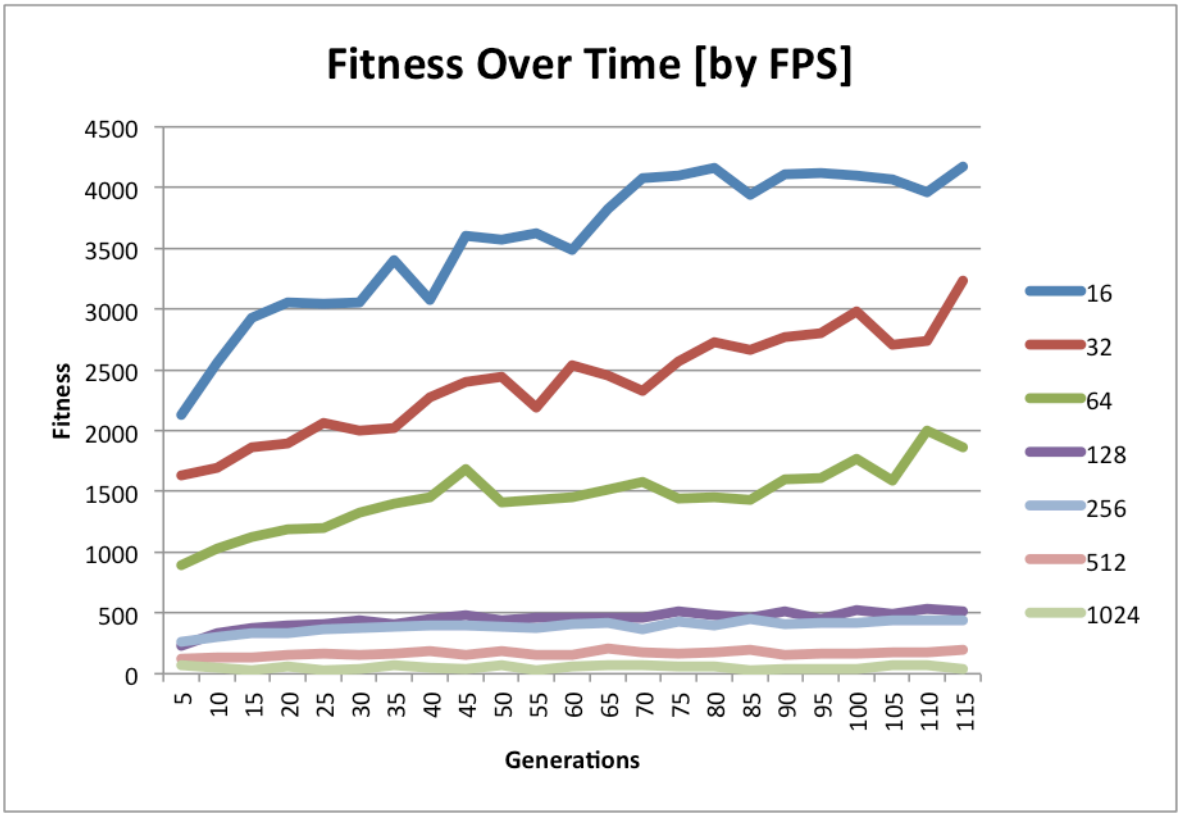

Figure 7 Growth curves for the GA learning with the game running at varying frames per second. The average of the population for five tests at each speed are shown.

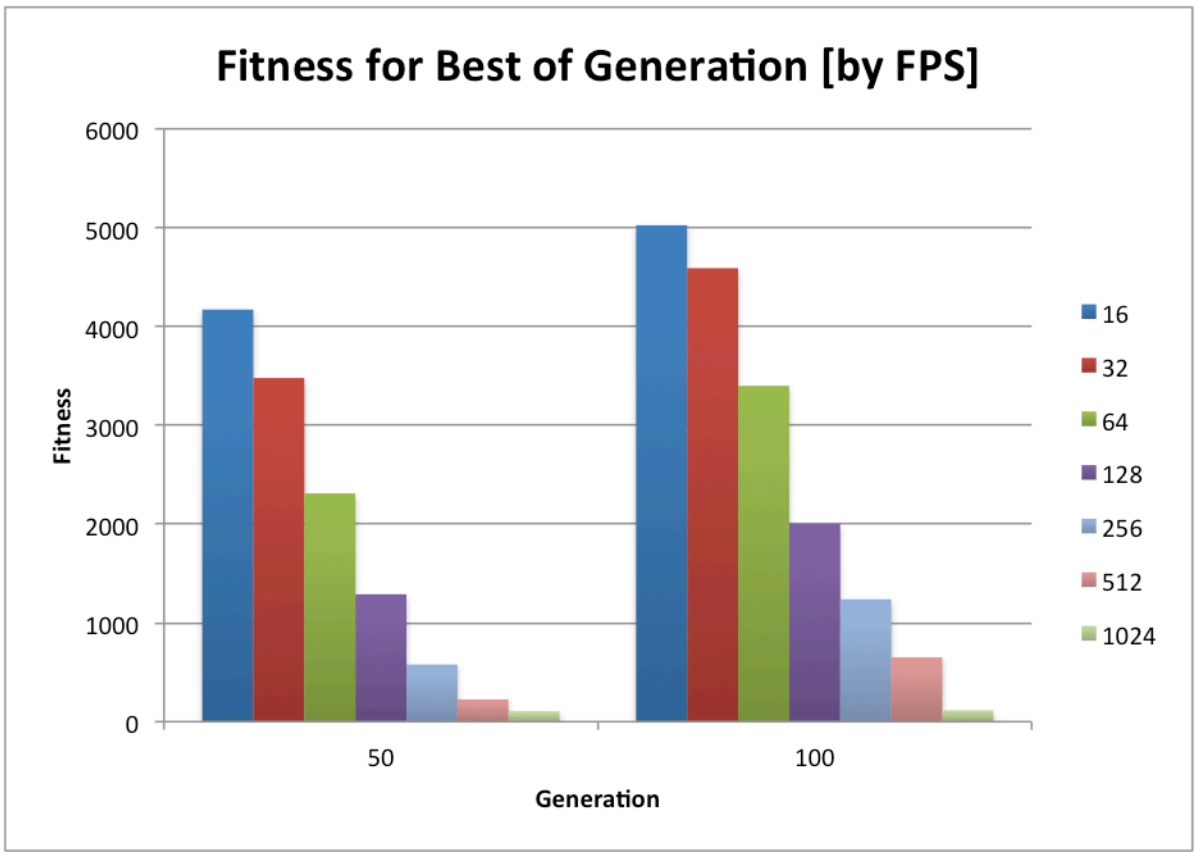

Figure 8 The average of the best individuals produced by the genetic algorithm after 50 and 100 generations. Each column represents the average of five agents, each of which were evaluated 30 times for a total of 150 data points.

performed a minimal amount of learning.

In addition, observations of the agents' behaviors were made from 16 to 64 FPS. They appeared normal and behaved intelligently. At speeds above 64 FPS, it appeared that agents were incapable of reacting properly to the events occurring around them. They would have a higher chance of a running directly into walls they would otherwise have avoided when running at slower speeds. If an enemy agent were to come into a learning agent's vicinity, it would be less likely to fire and often not seem to register that anything was different. At the highest speeds, the agent was effectively incapable of reacting to anything at all. It would spend most of its time thrusting directly into the nearest wall immediately upon appearing.

To further analyze the data, the best agents from each trial were analyzed after 50 and 100 generations. Each agent was evaluated in a test environment running at 16 FPS, regardless of the FPS the agent learned at, in the same way their fitnesses were evaluated during learning. Each agent from the five different trials was tested in 30 separate trials, yielding a total of 150 trials for each FPS learning speed at each of the two generations, 50 and 100. Figure 8 shows the averaged fitnesses of these trials. This data shows similar trends to that of the average fitnesses learned by the agents, with the agents learning at 16 FPS performed the best while the agents performing at 1024 FPS were clearly the worst. However, the best individual tests shown in Figure 8 show that the 128 through 512 FPS learning speeds may have merit. Even though they show less effective learning than the slower speeds, they still show some level of improvement. There is notable improvement from 50 to 100 generations in all cases, except at 1024 FPS. It's clear that a GA running at this speed produces little or no improvement in the Xpilot controller.

The above empirical data can also be confirmed by observation. At the beginning of the learning process for the lower FPS agents, the agents would regularly make obvious mistakes in combat. For example, some control programs would thrust while turning to avoid a wall and as a result run directly into the wall rather than avoid it or simply not turn sharply enough to avoid a bullet. However, by the end of the learning process, these mistakes were correct in the majority of the learned control programs. They exhibited intelligent behavior that was capable of defeating their opponent in the majority of situations. Meanwhile, the agents running at higher FPS generally displayed unintelligent behavior. For example, during the learning process, they would often thrust directly into the wall immediately upon spawning or aim incorrectly at opponents while firing. Even after learning was completed, these traits still remained strong in the majority, if not all, of the population.

Another observation that can be made from this graph (Figure 8) is a comparison of fitnesses from differing learning speeds at 50 and 100 generations. One can compare 16 FPS after 50 generations with 32 FPS after 100 generations, as the learning will take about the same time. With this in mind, consider 16 FPS at 50 generations versus 32 FPS at 100 generations. Since the fitness of 32 FPS at 100 generations is higher than 16 FPS at 50 generations, it can be concluded that 32 FPS is a better learning speed than 16 FPS. Now consider 32 FPS at 50 generations and 64 FPS at 100 generations. These fitnesses are close to the same with 32 FPS slightly better than 64 FPS. Considering 64 FPS at 50 generations and 128 FPS at 100 generations shows that 64 FPS has the clear advantage.

C. Conclusions

Based on the collected data, it can be deduced that learning at a considerably higher FPS has a significant negative impact on the quality of learning produced. That said, running at 16 FPS is not necessarily the ideal solution either. Agents that learned at 16 FPS had the best results per generation, but those that learned at 32 FPS had the best results over time, since they were effectively learning at twice the speed. In Xpilot, learning algorithms often produce a large variation in their results due to the nature of the environment. As a result, even though the average fitness within the population run at 32 FPS is noticeably lower than that of the population run at 16 FPS, the average of the best agents is quite comparable. Given the increased of speed of learning due to the faster FPS, it makes sense to use 32 FPS for learning. In addition, since the 64 FPS versus 32 FPS results are nearly equal, 64 FPS is also a reasonable choice for learning. Speeds at 128 FPS and above are not recommended for standard GA learning.

Another conclusion that can be drawn from the data is what range of FPS is acceptable and useful for learning in PAL. When determining an appropriate speed for the simulation server,

the ideal FPS would still learn but need not be adequate if learning on its own. Running Xpilot at 64 FPS would be a safe bet, but higher speeds are also possible. Agents are still able to learn and improve over time, but when compared to the 16 to 64 FPS agents, do not learn nearly as well or as fast. Although running PAL at 64 FPS would be sure to yield good results, depending on the goals of the research, running the simulation at FPS between 128 and 512 would also be acceptable given that they still improve and learn over time. They show a larger disconnect from the ideal FPS of 16 and as a result would not learn effectively on their own. However, if combined with PAL they could serve well as the simulation server speed. Any agent running at or above 1024 FPS, though, is ineffective. Algorithms run at this speed show neither intelligence nor improvement over time.

These tests will help in determining the FPS that we can use to test the limits of the Punctuated Anytime Learning system. In future work, PAL will be tested with the simulation running at 512 FPS. The higher the FPS the system is capable of learning at, the better it will be at improving game agents during play and dealing with inaccuracies in robot models for actual robot real-time learning.

Fitness Biasing Applied to Xpilot-AI

Based on the results presented in the previous section, 128 FPS was determined to be the fastest FPS that showed consistent aspects of continued learning. Despite the degradation in behavior due to high FPS, it was selected for testing the application of Fitness Biasing to Xpilot-AI. In past experiments using standard genetic algorithms, tests were run primarily at 64 FPS as that was the fastest Xpilot could be run at without losing too much in the way of performance.

Here, an attempt is made to learn at these faster speeds using Fitness Biasing to help compensate for the discrepancies in the system.

A. Fitness Biasing versus Co-Evolution of Model Parameters in Xpilot-AI

In the field of evolutionary robotics, Co-Evolution of Model Parameters is a powerful system capable of fast and efficient learning with demonstrated results (Parker, Punctuated Anytime Learning for Hexapod Gait Generation). Fitness Biasing, though also useful in evolutionary computation, has characteristics that make it not as effective as the Co-Evolution of Model Parameters for many problems. In Punctuated Anytime Learning, every punctuated generation involves runs on the actual agent as opposed to just on the learning system's simulation. When using a Fitness Biasing-based Punctuated Anytime Learning system, the entire population is tested on the actual agent. With the Co-Evolution of Model Parameters, on the other hand, only three agents are tested. When dealing with a robot that is hundreds of times slower than a simulation and is subject to malfunctions when overused, any excess use of the robot should be avoided. However, Co-Evolution of Model Parameters requires that an evolvable model of the of the robot be produced. This is not always possible, so sometimes the use of Fitness Biasing is the only option. In a video game where the degradation in the system is due to a high frame rate, an evolvable model would be hard to produce and since multiple runs result in minimal cost, Fitness Biasing is a good choice. Running an entire population on the actual agent in Xpilot, even at a slow speed of 16 FPS, is not prohibitively slow. There is also no physical cost; an Xpilot agent has no physical moving parts that are capable of breaking or wearing down with time. It is an unchanging entity.

B. Experiment

To run Punctuated Anytime Learning, the previously created genetic algorithm-agent was modified into two forms: one to act as the simulation and one to act as the agent. The simulation would run just as the normal genetic algorithm did with two key additions: while calculating fitnesses, it would bias them as previously described. In addition, every 15 generations it would connect to the agent to test all the individuals in the genetic algorithm population to calculate new biases. While doing this, the genetic algorithm would halt itself and discontinue learning until the new biases were received. The modified genetic algorithm code can be found in Appendix C.

The client running the actual agent would run separately but simultaneously with the genetic algorithm. The client continued to play at all times, whether or not it was currently running tests for the learning system. It had a communication system that ran indefinitely, waiting for the learning system to signal it with a new population to test (code for this can be found in Appendix D). When a new population was communicated to the actual agent for testing, it would run each chromosome of the population on the agent, record their fitnesses, calculate their biases, and then send the new information back to the learning system. At this point it would continue to play using the best of the chromosomes for its controller.

Upon receiving results from the agent, the learning system would overwrite its own collection of biases with the new information and continue learning in simulation until another 15 generations had passed. This process is repeated until stopped by the researcher, or some predetermined stopping point has been reached (i.e. stop after a certain number of generations).

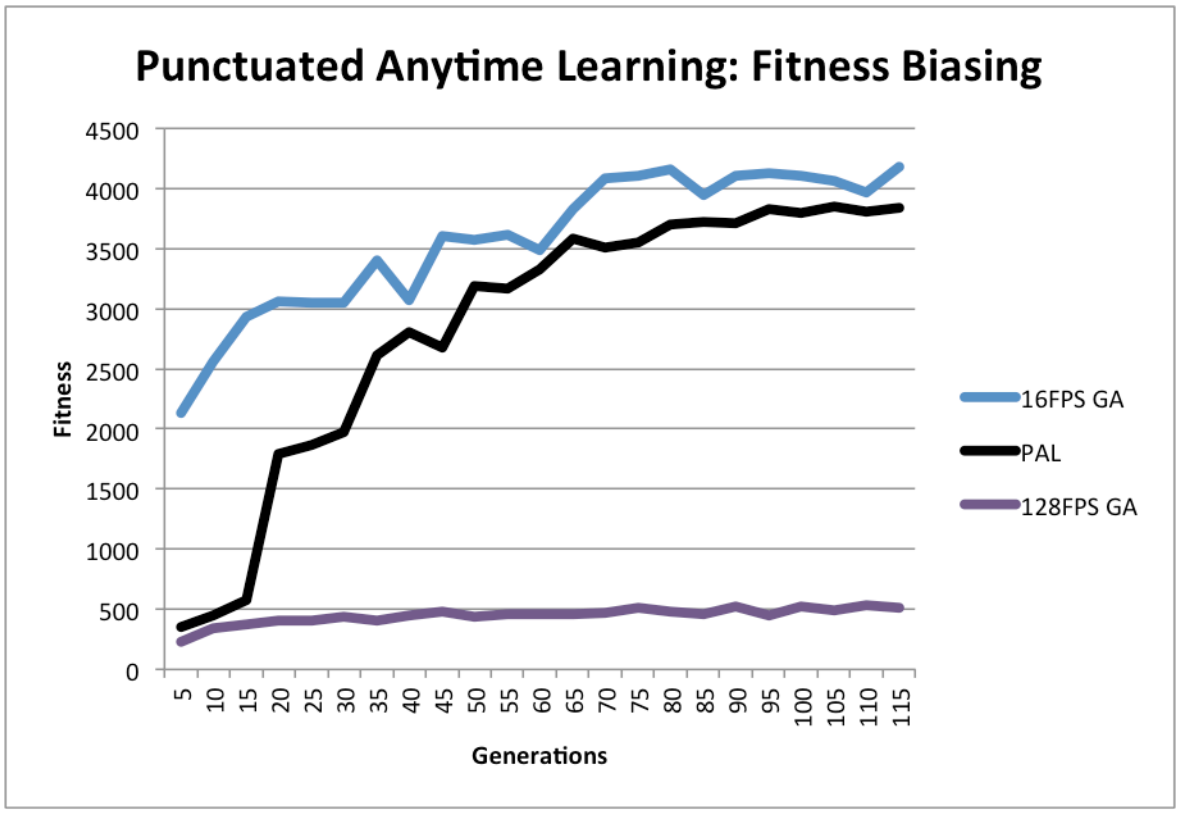

Figure 9 Fitness biased agent compared to a standard GA run at both 16 FPS and at 128 FPS

C. Results

Five tests runs using five randomly generated populations were run for 115 generations each. Based on this data, it is clear that the fitness biasing produced favorable results. The system tested the current population on the agent to generate new biases every 15 generations. Every 5 generations, the average fitness of the population was recorded. Figure 9 shows the average fitness over time as the population is evolving. Based on this data, it can be observed that there was great improvement in the fitness of the agent throughout the learning period. In the beginning, the fitness averaged at approximately 350 and rose to an average of approximately 3,840 by the end of testing.

Of additional importance is a comparison of controller quality related to the amount of time spent playing on the agent. In the genetic algorithm controlled agents, the entire game is

played live against competitors in order to learn in real time. After five generations, its fitness averaged at about 2,100. In the fitness-biased agent, though it had learned over the course of 35 generations in our testing, only two of these generations were spent on the agent playing against Sel. After these two generations, the fitness biasing system's average fitness is already higher than that of the fifth generation genetic algorithm-controlled agent and is approximately equal to the tenth generation. This is an important factor for real-time learning because it is learning faster per games played against its opponent.

The above empirical data can also be confirmed by observation. At the start of the learning process, the agent regularly made clear mistakes in combat. For example, some control programs would not turn sharply enough to avoid a bullet or a wall, not thrust to avoid a bullet, or aim incorrectly at its opponent when firing. However, towards the end, these mistakes were corrected in the majority of the learned control programs. The agent would adeptly dodge enemy bullets while responding quicker and firing back with greater accuracy. This increase in optimal behavior can be attributed to the learning that took place as a result of the Fitness Biasing system. By the end of training, all five trials produced individuals with intelligent behavior that were able to consistently beat Sel both in fitness and by the scoring system Xpilot uses to measure player kills and deaths.

D. Conclusions

Fitness Biasing when applied to Xpilot-AI agents can be used to evolve exceptional controllers that are highly competitive when facing opponents. The agents show the ability to learn effective combat behavior that appears complex to an observer. Throughout testing, the fitness biasing system showed a clear improvement over the quality of learning reached by other

agents, especially in the speed at which the learning took place. It has been shown to be a viable platform for effective learning.

Live Fitness Biasing Experimentation

One of the primary arguments in favor of Punctuated Anytime Learning as the primary learning system for this research is its ability to adapt in real-time to changing environments. In the video game industry, in order for a computer-controlled bot to be considered truly intelligent, it must be able to react and improve based on the skills of the human players it is challenging. Likewise, in the field of evolutionary robotics, the mark of a quality learning system is one that is capable of handling unexpected environmental changes or degradations in capabilities of the robot without issue. Fitness Biasing is ideal for creating this behavior because of the frequency with which the system's biases are updated. If an environmental change occurs, it does not take long for the system to recognize this and update the fitnesses' biases accordingly. The following experiment demonstrates this behavior and how well the fitness biasing implementation performs in such a scenario.

A. Experiment

The purpose of this experiment is to compare the quality of learning in a fitness biasing system to that of a standard genetic algorithm-based agent while encountering changes in the behavior of their opponents. A genetic algorithm running on a simulation at 128 FPS was allowed to learn for 100 generations with Sel, the standard opponent used for learning in this research. After 100 generations, learning had tapered off and was negligible, indicating that a near-optimal solution had probably been reached. This experiment starts at the 100^{th} generation with the agent still competing against Sel. After 15 generations, Sel was removed from the game

and another agent, named Morton 5GA6 was introduced to the combat. Morton 5GA6 is the result of ongoing research by Parker. It was developed using a genetic algorithm to learn the parameters for a rule-based system as described in this paper, but the rule-based system was different. After 15 generations of combat, the agent's opponent was switched back to Sel. Fifteen generations later, it was placed in combat against an agent named Morton, a simplistic rule-based bot that was not the result of learning, and finally after 15 more generations the agent fought Sel one last time.

The Fitness Biasing system was run through the same set of opponents for the same periods of time. Its learning system was also run at 128 FPS to maintain consistency, and throughout the test the learning system's opponent in the simulation was always Sel, regardless of which opponent the actual agent was facing. Using the same model opponent throughout testing, regardless of what the actual agent is experiencing, allows the system to demonstrate the effectiveness of Fitness Biasing for the purposes of learning even with an incorrect model. To insure that both the fitness biasing agent and the genetic algorithm agent started at the same point, the population from the 100^{th} generation of the genetic algorithm was stored and subsequently loaded into the fitness biasing system. Starting these tests with the two systems using the same population helps demonstrate the differences in learning capabilities of these systems as they are forced to react to changes in their opponents. It also insures that they are both starting at the same point – neither system is given an advantage over the other due to prior learning. Both will have learned an equal amount at the start of these trials as a result. The fitness biasing system started by testing the current population on the actual agent to attain a set of valid biases and repeated this process every 15 generations for the duration of the trials. The result of this is that

every time a new opponent was introduced, new biases were calculated for the fitness biasing system.

B. Results

Fifteen tests using fifteen randomly generated populations were run for 100 generations each in the genetic algorithm. Fitness data was recorded every fifth generation of learning. Each of these fifteen populations were then trained against each of the five agents (Sel, Morton 5GA6, Sel, Morton, and Sel) in succession. The same 15 populations used to train the genetic algorithm were also used for the fitness biasing system. Figure 10 shows the averages of the best agents from each of the 15 trials. Based on this data, it is clear that the fitness biasing system is favorable compared to the genetic algorithm-based agent. By the 100th generation, the genetic

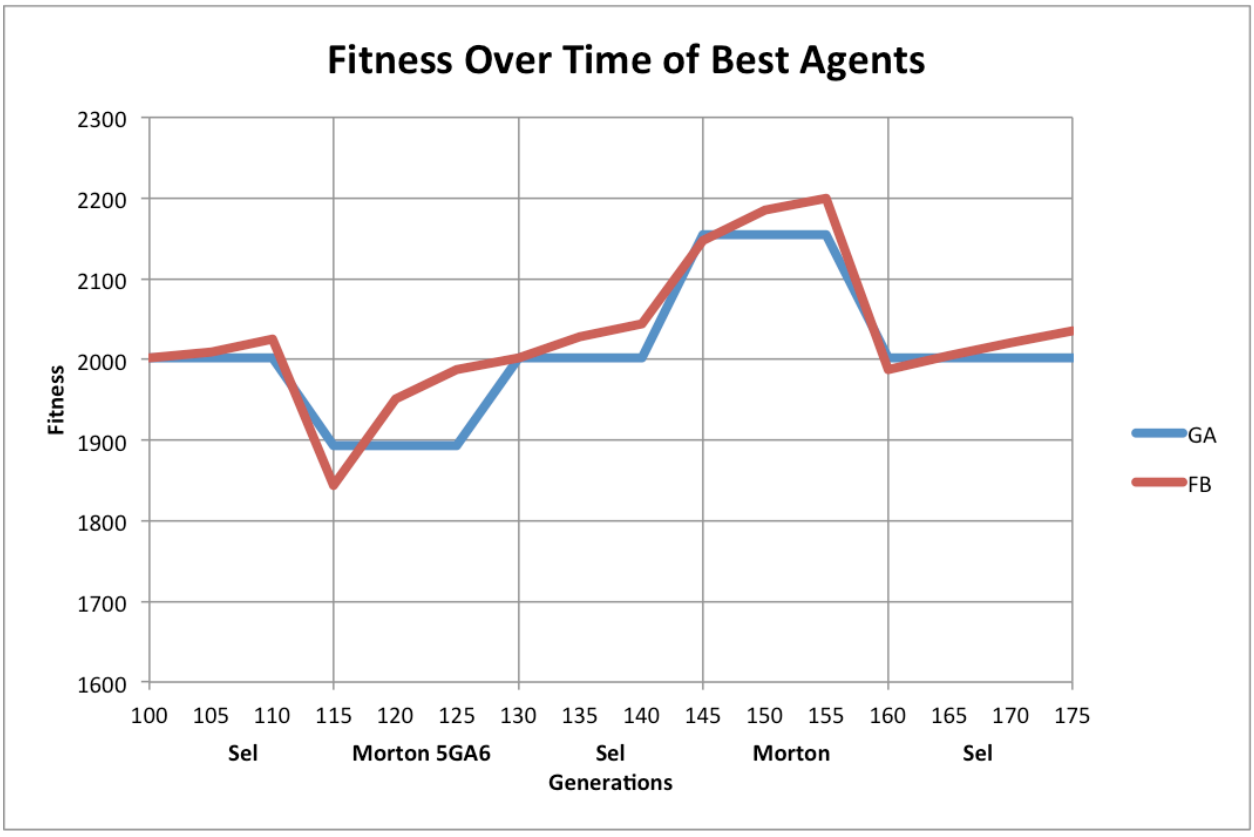

Figure 10 Fitness biased against compared to a standard genetic algorithm-based agent in trials against the following five agents in succession: Sel, Morton 5GA6, Sel, Morton, Sel

algorithm was effectively unable to continue learning. Its fitness remained static simply because it had already learned as much as it was capable of learning. The Fitness Biasing agent, on the other hand, showed steady improvement, in particular whenever a change in opponent occurred. Though the standard genetic algorithm often started at or above the fitness of the Fitness Biasing agent upon facing a new opponent, it did not improve. The fitness biasing system, though, after re-calculating its biases to match the new opponent showed constant improvement. By the end of each 15 generation period, the fitness biasing system was clearly performing better overall.

C. Conclusions

This experiment clearly shows that fitness biasing is capable of both effective learning, as also shown in the previous section, and in adaptive learning. While the genetic algorithm-based agent struggled with a more challenging opponent and failed to improve on any one particular opponent, the fitness biasing system largely improved on each opponent shortly after the opponent was introduced to combat. Further still, even while facing an incorrect model the system continued to learn once the biases had been updated for each opponent.

Conclusions

In this research, three primary experiments have been performed that serve to improve our understanding of the Xpilot-AI combat game and learning environment as well to demonstrate the effectiveness of Punctuated Anytime Learning. Each experiment was successful and conclusive, while also opening doors to new and future work that builds on the work shown here. The primary contributions are the determination of what the optimal game speeds are for learning in Xpilot, the demonstration that Fitness Biasing is a viable means for evolving combat

control programs, and the execution of experiments that show it is capable of effective and efficient real-time learning in changing environments.

In the past, research in Xpilot-AI has been performed at a game speed of 64 FPS. This was done in order to speed up the rate of learning in the game but was never justified through experimentation. Until now, at no point has work been done to analyze the effects of game speed on genetic algorithm-based learning in Xpilot controllers. This is important knowledge since high speeds are needed for efficient learning, but result in detrimental effects on the quality of learning. Based on the results of this work, it has been shown that while 64 FPS is a reasonable choice for learning, it may not be the best choice. Although it shows consistent improvement in learning without large negative effects, it does suffer degradation due to information loss during server-client communication. Results show that learning in Xpilot at 32 FPS performed marginally better in the same time period, despite the decrease in gameplay speed. This experiment has the added benefit of determining an optimal speed for a learning system based on Punctuated Anytime Learning. It was shown that agents learning at 128 FPS still showed signs of continued learning throughout the learning process. Though there were also significant issues with the quality of learning while running at that speed, it was not enough to prevent it from being usable. This proved to be quite useful because a Punctuated Anytime Learning system is particularly applicable when the learning system's simulation is not perfect.

Using this information, an agent was implemented using Fitness Biasing, a subset of Punctuated Anytime Learning, as its learning system. Results demonstrated a strong learning curve that was able to outmatch all other systems in speed. At its highest fitness, after 115 generations in the learning system, the Fitness Biasing system was nearly matching the fitness of a genetic algorithm-based agent running at 16 FPS, which would take significantly longer. Given

that agents learning at 16 FPS were shown to be the most effective learners, this shows a clear ability for a Fitness Biasing system to learn effectively. This is also interesting because, though the learning system had run 115 generations, in reality only seven of those generations had been performed on the actual agent, given that the periodic bias updating generations only occur every 15 generations. The 15 generations run on the learning system takes place at highly increased speed over that of the agent running at 16 FPS. By analyzing how long each algorithm took to get to their respective fitnesses, after the 7 punctuated fitness biasing generations, the system was already outmatching the fitnesses being produced by the genetic algorithm-based agent running at 16 FPS.

Further still, once Fitness Biasing had been shown to be a feasible system for learning, experiments were conducted to test its ability to adapt to changing environments. When introduced to new agents, both ones that were more and less advanced than the previous agent, it not only adapted to the change quickly but showed continued learning in response to each new agent as well. While the genetic algorithm remained static and did not continue learning, the Fitness Biasing agent constantly improved. This shows that the agent is capable of real-time learning, in large part due to the time with which the agent reacted to its new opponents. One fitness biasing generation for each new opponent was enough to allow the system to return to its former fitness and remain competitive.

Due to these experiments, it is clear that Fitness Biasing and, as a result, Punctuated Anytime Learning is a powerful learning methodology in the Xpilot-AI game environment. Demonstrating that it works well in this environment, because of the complexity of Xpilot, implies that it will be successful in other environments as well. One other major application of such a system is in the field of evolutionary robotics. One of the most important aspects of an

intelligent autonomous system is the ability to react in real-time to changes in its environment. The real world is constantly changing and robots need to be able to adapt appropriately. Robots are often sent in to unknown or dangerous environments that are too dangerous to send in humans. It is in these situations that an effective real-time learning system would be most useful.

Consider the idea of space exploration. One of the largest applications for the field of robotics is going places where, at this time, no human can. What if we could, rather than sending one or two robots to a new planet, as is the case with the Mars Rover, instead send dozens of small robots? Punctuated Anytime Learning is highly useful in such a scenario because the bulk of the computing required for learning takes place off-line. The robot does not need the expensive computing equipment received for learning; all it needs is a small computer capable of running the control program. The learning can take place elsewhere, on a computer in a safe environment. In the case of space exploration, the learning system could be as far away as a satellite orbiting the planet in space. While dozens of inexpensive and easier to build robots are scouring the surface, the satellite is perfectly safe as it continues learning for the robots on the ground. When changes occur in the environment on the planet, the satellite will be able to update the control programs present on the robots on the ground to help them adapt and survive longer. The applications of such a system are vast, both for simple areas like video game development and complex and far reaching areas like search and discovery in new environments.

A. Future Work

There are a number of different areas where this work can lead. In future work, the system will be tested against human opponents. Rather than having the actual agent in the Fitness Biasing system play against a computer controlled agent, such as Sel, imagine it running on a server hosted on the internet for anyone to join. The agent would be able to play against

countless opponents, each of which could have their own unique play style. It would prove to be a challenging and unique opportunity to test the robustness of the Fitness Biasing system while at the same time promoting a fun experiment for those who wish to get involved. If successful, it would only further reinforce the idea that this system would be useful if implemented in commercial video games. In addition, it would give further evidence that Punctuated Anytime Learning with Fitness Biasing is a viable system for the real-time learning of controllers of autonomous agents, both those in video games and those that are robots.

Appendix

A. Standard genetic algorithm used for optimizing Xpilot-AI learning speeds

```
1.  # Copyright 2010-2011, Connecticut College Computer Science
2.  """Classic genetic algorithm, generalized to work with any problem."""
3.
4.  __author__ = 'Phil Fritzsche <pfritzsche@gmail.com>'
5.
6.  import random
7.  import sys
8.  import time
9.
10. import gabot_timed
11. from gabot_timed import xpai
12.
13. sys.setcheckinterval(0)
14.
15. fps = len(sys.argv[1]) == 2 and '0%s' % sys.argv[1] or sys.argv[1]
16.
17. OUTPUT_POP_FILE = 'fps_trials/%s/output_pop_' % fps
18. OUTPUT_BESTS_FILE = 'fps_trials/%s/output_best_chroms' % fps
19. OUTPUT_FIT_FILE = 'fps_trials/%s/output_fitnesses' % fps
20. OUTPUT_EXT = '.txt'
21.
22.
23. class Struct(dict):
24.     """A dictionary whose values can also be accessed as attributes."""
25.     def __init__(self, **kwargs):
26.         self.update(kwargs)
27.
28.     def __getattr__(self, name):
29.         """Retrieves the value for the given name, if it exists. Returns
30.         None if it does not."""
31.         if name in self:
```

```
32.             return self[name]
33.
34.     def __setattr__(self, name, value):
35.         """Sets the value for the given name to the given value."""
36.         self[name] = value
37.
38.
39. def avg(seq):
40.     """Calculates the average of a sequence."""
41.     return sum(seq) / len(seq)
42.
43.
44. def comp(x, y):
45.     """Compares two tuples based on the second item in each. Returns 1 if
46.     x > y, 0 if x == y, and -1 if x < y."""
47.     if x[1] > y[1]:
48.         return 1
49.     elif x[1] == y[1]:
50.         return 0
51.     else:
52.         return -1
53.
54.
55. class GA(object):
56.     def __init__(self, mutate_rate, chrom_size=72, pop_size=128, pop_file='',
57.                  chrom_set=[0, 1]):
58.         """Public constructor."""
59.         self.mutate_rate = mutate_rate
60.         self.chrom_size = chrom_size
61.         self.pop_size = pop_size
62.         self.chrom_set = chrom_set
63.         gabot_timed.launch(fps)
64.         if not pop_file:
65.             self.pop = self.generate_init_pop()
66.         else:
67.             self.pop = self.read_pop(pop_file)
68.
69.     def learn(self):
70.         """Primary learning loop of the GA. Performs the following actions
71.         in an infinite loop:
72.
73.             1. Every 5 iterations, saves the current fitness information.
74.             2. Every 50 iterations, save the best chromosome and the current
75.                 population to a file.
76.             3. Calculates new fitnesses for every chromosome in the population.
77.             4. Stochastically generates a new population.
78.             5. Gets the current best chromosome."""
79.         iterations = 0
80.         self.best = Struct(fitness=0, chrom='')
81.         while True:
82.             iterations += 1
83.             if not iterations % 5:
84.                 self.save_fitness_information(iterations)
85.             if not iterations % 50:
86.                 self.save_best_and_pop(iterations)
87.             self.fitnesses = self.calc_fitnesses()
88.             self.pop = self.generate_new_pop()
89.             self.best = self.get_best_chrom()
90.
91.     def get_best_chrom(self):
92.         """Sorts the population by fitness and returns the best chromosome."""
```

```
93.         gen = zip(self.pop, self.fitnesses)
94.         gen.sort(comp)
95.         n = len(gen) - 1
96.         return Struct(chrom=gen[n][0], fitness=gen[n][1])
97.
98.     def generate_init_pop(self):
99.         """Generates an initial random population."""
100.                pop = []
101.                for i in range(self.pop_size):
102.                    chrom = ''
103.                    for j in range(self.chrom_size):
104.                        chrom += str(random.choice(self.chrom_set))
105.                    pop.append(chrom)
106.                return pop
107.
108.            def read_pop(self, filename):
109.                """Reads in a population from the specified file."""
110.                fin = open(filename, 'r')
111.                file_text = fin.read()
112.                return file_text.split()
113.
114.            def calc_fitnesses(self):
115.                """Calculates the fitness of every chromosome in the population."""
116.                fits = []
117.                for chrom in self.pop:
118.                    gabot_timed.set_reset(chrom)
119.                    #xpai.talk('waiting')
120.                    gabot_timed.is_done.wait()
121.                    #xpai.talk('done waiting')
122.                    fits.append(gabot_timed.get_fitness())
123.                return fits
124.
125.            def generate_new_pop(self):
126.                """Generates a new population of chromosomes by doing the following
127.                enough times to replace every chromosome in the current population:
128.
129.                    1. Chooses crossover point for the two parents.
130.                    2. Chooses two parents.
131.                    3. Mates the parents together.
132.                    4. [Possibly] mutates the children.
133.                    5. Adds the new child to the new population."""
134.                new_pop = []
135.                for i in range(self.pop_size):
136.                    cross = random.randrange(self.chrom_size)
137.                    parents = self.select_parents()
138.                    child = self.mate_parents(parents, cross)
139.                    mutated_child = self.mutate(child, self.mutate_rate)
140.                    new_pop.append(mutated_child)
141.                return new_pop
142.
143.            def select_parents(self, bin_count=10000):
144.                """Uses a roulette-wheel style selection to choose two new parents
145.                based on their fitnesses."""
146.                parents = Struct()
147.                partial_sum = 0
148.                total_fitness = sum(self.fitnesses)
149.                p_mom = random.randrange(total_fitness)
150.                p_dad = random.randrange(total_fitness)
151.                for chrom, fit in zip(self.pop, self.fitnesses):
152.                    partial_sum += fit
153.                    if not parents.mom and p_mom < partial_sum:
```

```
                        parents.mom = chrom
                    if not parents.dad and p_dad < partial_sum:
                        parents.dad = chrom
                    if parents.dad and parents.mom:
                        return parents

            def mate_parents(self, parents, cross):
                """Returns a new child resulting from the passed in parents."""
                return parents.mom[:cross] + parents.dad[cross:]

            def mutate(self, chrom, mutate_rate):
                """Goes through every bit of the child determining if mutation
                should occur. If so, performs the mutation and returns the new
                child when finished."""
                new_chrom = ''
                for i in range(len(chrom)):
                    if random.random() <= mutate_rate:
                        choices = list(set(self.chrom_set) - set(chrom[i]))
                        new_chrom += str(random.choice(choices))
                    else:
                        new_chrom += chrom[i]
                return new_chrom

            def save_fitness_information(self, its):
                """Saves the current fitness information to a file."""
                fout = open(OUTPUT_FIT_FILE + OUTPUT_EXT, 'a')
                fout.write('%d, %d, %.3f\n' % (
                    its, self.best.fitness, avg(self.fitnesses)))
                fout.close()

            def save_best_and_pop(self, its):
                """Saves the current best agent and population to a file."""
                fout = open(OUTPUT_BESTS_FILE + OUTPUT_EXT, 'a')
                fout.write('%d, %s\n' % (its, self.best.chrom))
                fout.close()

                fout = open(OUTPUT_POP_FILE + str(its) + OUTPUT_EXT, 'w')
                for chrom in self.pop:
                    fout.write('%s\n' % chrom)
                fout.close()

        def main():
            mutate_rate = 0.01
            ga = GA(mutate_rate)
            ga.learn()

        main()
```

B. Agent used in coordination with the genetic algorithm in appendix A

```
# Copyright 2010-2011, Connecticut College Computer Science
"""GA bot for use in PAL thesis research."""

__author__ = 'Phil Fritzsche <pfritzsche@gmail.com>'

```

```
6.  import math
7.  import sys
8.  import thread
9.  import threading
10. import time
11.
12. import xpai
13.
14. sys.setcheckinterval(0)
15.
16. MAX_TURN = 20
17.
18. #chromosome = '10010011110001100010110000001111110110100101010000000010010000011001111
    0'
19. chromosome = '00000000000000000000000000000000000000000000000000000000000000000000000000'

20.
21. class Struct(dict):
22.     """A dictionary whose values can also be accessed as attributes."""
23.     def __init__(self, **kwargs):
24.         self.update(kwargs)
25.
26.     def __getattr__(self, name):
27.         """Retrieves the value for the given name, if it exists. Returns
28.         None if it does not."""
29.         if name in self:
30.             return self[name]
31.
32.     def __setattr__(self, name, value):
33.         """Sets the value for the given name to the given value."""
34.         self[name] = value
35.
36.
37. def gene_to_num(gene):
38.    dec_num = 0
39.    for i in gene:
40.        dec_num = (dec_num << 1) | int(i)
41.    return dec_num
42.
43.
44. gene = Struct()
45. def load_gene(chromosome):
46.     xpai.talk(str(chromosome))
47.     global gene
48.     gene = Struct(
49.         span=gene_to_num(chromosome[0:4]),
50.         offset_inc=gene_to_num(chromosome[4:8]) + 1,
51.         samespread=gene_to_num(chromosome[8:12]),
52.         wall_span1=gene_to_num(chromosome[12:16]) * 4,
53.         wall_span2=gene_to_num(chromosome[16:20]) * 4,
54.         vd_bullet_dist=gene_to_num(chromosome[20:24]) * 4,
55.         close_wall_speed=gene_to_num(chromosome[24:28]) * 2,
56.         c_angle_before_thrust=gene_to_num(chromosome[28:32]) * 10,
57.         d_bullet_dist=(gene_to_num(chromosome[32:36]) * 4) + \
58.             (gene_to_num(chromosome[20:24]) * 4),
59.         medium_wall_speed=gene_to_num(chromosome[36:40]) * 4,
60.         m_angle_before_thrust=gene_to_num(chromosome[40:44]) * 10,
61.         wall_avoid_angle=gene_to_num(chromosome[44:48]),
62.         screen_thrust_speed=gene_to_num(chromosome[48:52]),
63.         radar_no_thrust_speed=gene_to_num(chromosome[52:56]),
64.         ship_error_to_shoot=gene_to_num(chromosome[56:60]) * 2,
```

```
        radar_error_to_shoot=gene_to_num(chromosome[60:64]) * 2,
        vd_dodge_bullet_angle=gene_to_num(chromosome[64:68]) * 10,
        d_dodge_bullet_angle=gene_to_num(chromosome[68:72]) * 10
    )


def wall_feeler(range, degree):
    """Checks for walls within the given range at the given degree from the
    ship's current position. Uses absolute scale for degrees."""
    delta_x = xpai.self_x() + range * math.cos(math.radians(degree))
    delta_y = xpai.self_y() + range * math.sin(math.radians(degree))
    res = xpai.wallbetween(xpai.self_x(), xpai.self_y(), delta_x, delta_y)
    return res == -1 and range or res


def is_shot_behind_wall(n, span):
    c1 = bool(wall_feeler(
        xpai.shot_dist(n), xpai.shot_xdir(n)) < xpai.shot_dist(n))
    c2 = bool(wall_feeler(
        xpai.shot_dist(n), xpai.angleadd(
            xpai.shot_xdir(n), span)) < xpai.shot_dist(n))
    c3 = bool(wall_feeler(
        xpai.shot_dist(n), xpai.angleadd(
            xpai.shot_xdir(n), -span)) < xpai.shot_dist(n))
    return c1 and c2 and c3


def is_ship_behind_wall(n):
    return bool(wall_feeler(
        xpai.ship_dist(n), xpai.ship_xdir(n)) < xpai.ship_dist(n))


def is_radar_ship_behind_wall(n):
    return bool(wall_feeler(
        xpai.radar_dist(n), xpai.radar_xdir(n)) < xpai.radar_dist(n))


       def wall_avoid_turn_dir_ship(n, offset_inc):
           range = xpai.ship_dist(n)
           degree = xpai.ship_xdir(n)
           return wall_avoid_turn_dir_helper(range, degree, 0, offset_inc)


       def wall_avoid_turn_dir_radar(n, offset_inc):
           range = xpai.radar_dist(n)
           degree = xpai.radar_xdir(n)
           return wall_avoid_turn_dir_helper(range, degree, 0, offset_inc)


       def wall_avoid_turn_dir_helper(rng, deg, os, os_inc):
           if os > 180:
               return -1000
           elif wall_feeler(rng, deg + os) == rng:
               return deg + os
           elif wall_feeler(rng, deg - os) == rng:
               return deg - os
           return wall_avoid_turn_dir_helper(rng, deg, os + os_inc, os_inc)


       def screen_enemy_num(n):
           """Returns the number of the nearest enemy on screen, or -1 if no
```

```
126.            enemy is currently on screen."""
127.            if xpai.ship_x(n) == -1:
128.                return -1
129.            elif xpai.teamplay() == 1 and xpai.self_team() != xpai.ship_team(n):
130.                return n
131.            return screen_enemy_num(n + 1)
132.
133.        def screen_enemy_num2(n, first):
134.            if first < 0:
135.                return -1
136.            elif xpai.ship_x(n) == -1:
137.                return -1
138.            elif (n != first and xpai.teamplay == 1 and
139.                  xpai.self_team() != xpai.ship_team(n)):
140.                return n
141.            return screen_enemy_num(n + 1)
142.
143.        def radar_enemy_num2(n, first):
144.            if first < 0:
145.                return -1
146.            elif xpai.radar_x(n) == -1:
147.                return -1
148.            elif n != first and xpai.radar_xdir(n) != -1:
149.                return n
150.            return radar_enemy_num(n + 1)
151.
152.
153.        def is_same(x, y, spread):
154.            return abs(x - y) <= spread
155.
156.
157.        def radar_enemy_num(n):
158.            """Returns the number of the nearest enemy on radar, or -1 if no
159.            enemy currently exists on radar."""
160.            if xpai.radar_x(n) == -1:
161.                return -1
162.            elif xpai.radar_xdir(n) != -1:
163.                return n
164.            return radar_enemy_num(n + 1)
165.
166.
167.        def change_heading(dir):
168.            """Turns the ship relative to its heading."""
169.            xpai.self_turn(xpai.anglediff(xpai.self_heading(), dir))
170.
171.
172.        current_score = xpai.self_score()
173.        reset_flag = False
174.        frames = 0
175.        kills = 0
176.        final_fitness = 0
177.        start_time = time.time()
178.        pre_life = True
179.        is_done = threading.Event()
180.
181.        def reset_all():
182.            global current_score
183.            global frames
184.            global kills
185.            global final_fitness
186.            global is_done
```

```
187.            global reset_flag
188.            global start_time
189.            current_score = xpai.self_score()
190.            frames = 0
191.            kills = 0
192.            final_fitness = 0
193.            start_time = time.time()
194.            is_done.clear()
195.            reset_flag = False
196.
197.        def ai_main():
198.            """Main function for the script; called once every frame of the game."""
199.            global is_done
200.            global pre_life
201.            if pre_life and not xpai.self_alive():
202.                return
203.            pre_life = False
204.
205.            global current_score
206.            if xpai.self_score() > current_score and not is_done.isSet():
207.                global kills
208.                kills += 1
209.            elif xpai.self_score() < current_score and not is_done.isSet():
210.                global start_time
211.                start_time -= 20
212.            current_score = xpai.self_score()
213.
214.            #print 'is_done: %r' % is_done
215.            #print 'final_fitness: %r' % final_fitness
216.            #print 'kills: %r' % kills
217.            #print 'frames: %r' % frames
218.            if xpai.self_alive():
219.                global frames
220.                frames += 1
221.
222.                ship_num = screen_enemy_num(0)
223.                ship_num2 = screen_enemy_num2(0, ship_num)
224.                radar_ship_num = radar_enemy_num(0)
225.                radar_ship_num2 = radar_enemy_num2(0, radar_ship_num)
226.
227.                wf_r1 = wall_feeler(600, xpai.angleadd(
228.                    xpai.self_track(), -gene.wall_span1))
229.                wf_l1 = wall_feeler(600, xpai.angleadd(
230.                    xpai.self_track(), gene.wall_span1))
231.                wf_r2 = wall_feeler(600, xpai.angleadd(
232.                    xpai.self_track(), -gene.wall_span2))
233.                wf_l2 = wall_feeler(600, xpai.angleadd(
234.                    xpai.self_track(), gene.wall_span2))
235.
236.                # Dodge bullet if very close
237.                #print 'alive'
238.                if (xpai.shot_alert(0) > -1 and
239.                    xpai.shot_alert(0) < gene.vd_bullet_dist and
240.                    not is_shot_behind_wall(0, gene.span)):
241.                    #print 'bullet vd'
242.
243.                    added_ang = xpai.angleadd(xpai.shot_idir(0),
244.                                              gene.vd_dodge_bullet_angle)
245.                    change_heading(added_ang)
246.                    xpai.self_thrust(1)
247.
```

```
248.                elif (xpai.shot_alert(1) > -1 and
249.                      xpai.shot_alert(1) < gene.vd_bullet_dist and
250.                      not is_shot_behind_wall(1, gene.span)):
251.                    #print 'bullet2 vd'
252.
253.                    added_ang = xpai.angleadd(xpai.shot_idir(1),
254.                                              gene.vd_dodge_bullet_angle)
255.                    change_heading(added_ang)
256.                    xpai.self_thrust(1)
257.
258.                # Two wall feelers are close
259.                elif (is_same(wf_r1, wf_l1, gene.samespread) and
260.                      wf_r1 < gene.close_wall_speed * xpai.self_vel() and
261.                      xpai.self_vel() > 1):
262.                    #print 'both wall feelers close'
263.
264.                    turn_amt = xpai.angleadd(180, xpai.self_track())
265.                    change_heading(turn_amt)
266.                    if (abs(xpai.anglediff(xpai.self_heading(), turn_amt)) <
267.                        gene.c_angle_before_thrust):
268.                        xpai.self_thrust(1)
269.
270.                # Right wall feeler is closer than the left
271.                elif (wf_r1 < wf_l1 and
272.                      wf_r1 < gene.close_wall_speed * xpai.self_vel() and
273.                      xpai.self_vel() > 1):
274.                    #print 'right wall feeler close'
275.
276.                    turn_amt = xpai.angleadd(180, xpai.angleadd(
277.                        -gene.wall_span1, xpai.self_track()))
278.                    change_heading(turn_amt)
279.                    if (abs(xpai.anglediff(xpai.self_heading(), turn_amt)) <
280.                        gene.c_angle_before_thrust):
281.                        xpai.self_thrust(1)
282.
283.                # Left wall feeler is closer than the right
284.                elif (wf_r1 > wf_l1 and
285.                      wf_l1 < gene.close_wall_speed * xpai.self_vel() and
286.                      xpai.self_vel() > 1):
287.                    #print 'left wall feeler close'
288.
289.                    turn_amt = xpai.angleadd(180, xpai.angleadd(
290.                        gene.wall_span1, xpai.self_track()))
291.                    change_heading(turn_amt)
292.                    if (abs(xpai.anglediff(xpai.self_heading(), turn_amt)) <
293.                        gene.c_angle_before_thrust):
294.                        xpai.self_thrust(1)
295.
296.                # Dodge bullet if close [but not very]
297.                elif (xpai.shot_alert(0) > -1 and
298.                      xpai.shot_alert(0) < gene.d_bullet_dist and
299.                      not is_shot_behind_wall(0, gene.span)):
300.                    #print 'bullet d'
301.
302.                    added_ang = xpai.angleadd(xpai.shot_idir(0),
303.                                              gene.d_dodge_bullet_angle)
304.                    change_heading(added_ang)
305.                    xpai.self_thrust(1)
306.
307.                elif (xpai.shot_alert(1) > -1 and
308.                      xpai.shot_alert(1) < gene.d_bullet_dist and
```

```
309.                     not is_shot_behind_wall(1, gene.span)):
310.                 #print 'bullet2 d'
311.
312.                 added_ang = xpai.angleadd(xpai.shot_idir(1),
313.                                           gene.d_dodge_bullet_angle)
314.                 change_heading(added_ang)
315.                 xpai.self_thrust(1)
316.
317.             # Two wall feelers are at medium distance
318.             elif (is_same(wf_r2, wf_l2, gene.samespread) and
319.                   wf_r2 < gene.medium_wall_speed * xpai.self_vel() and
320.                   xpai.self_vel() > 1):
321.                 #print 'both wall feelers medium'
322.
323.                 turn_amt = xpai.angleadd(180, xpai.self_track())
324.                 change_heading(turn_amt)
325.                 if (abs(xpai.anglediff(xpai.self_heading(), turn_amt)) <
326.                     gene.m_angle_before_thrust):
327.                     xpai.self_thrust(1)
328.
329.             # Right wall feeler is closer than the left
330.             elif (wf_r2 < wf_l2 and
331.                   wf_r2 < gene.medium_wall_speed * xpai.self_vel() and
332.                   xpai.self_vel() > 1):
333.                 #print 'right wall feeler medium'
334.
335.                 turn_amt = xpai.angleadd(180, xpai.angleadd(
336.                     -gene.wall_span2, xpai.self_track()))
337.                 change_heading(turn_amt)
338.                 if (abs(xpai.anglediff(xpai.self_heading(), turn_amt)) <
339.                     gene.m_angle_before_thrust):
340.                     xpai.self_thrust(1)
341.
342.             # Left wall feeler is closer than the right
343.             elif (wf_l2 < wf_r2 and
344.                   wf_l2 < gene.medium_wall_speed * xpai.self_vel() and
345.                   xpai.self_vel() > 1):
346.                 #print 'left wall feeler medium'
347.
348.                 turn_amt = xpai.angleadd(180, xpai.angleadd(
349.                     gene.wall_span2, xpai.self_track()))
350.                 change_heading(turn_amt)
351.                 if (abs(xpai.anglediff(xpai.self_heading(), turn_amt)) <
352.                     gene.m_angle_before_thrust):
353.                     xpai.self_thrust(1)
354.
355.             # Turn towards nearest enemy ship on screen
356.             elif ship_num > -1 and not is_ship_behind_wall(ship_num):
357.                 #print 'turn towards nearest enemy ship on screen'
358.                 change_heading(xpai.ship_aimdir(ship_num))
359.
360.             # Turn towards second nearest enemy ship on screen
361.             elif ship_num2 > -1 and not is_ship_behind_wall(ship_num2):
362.                 #print 'turn towards nearest enemy ship2 on screen'
363.                 change_heading(xpai.ship_aimdir(ship_num2))
364.
365.             # Turn, thrust towards nearest enemy
366.             elif (ship_num > -1 and
367.                   not wall_avoid_turn_dir_ship(ship_num, gene.offset_inc) == -
    1000):
368.                 #print 'turn and thrust towards nearest enemy'
```

```
369.                    turn_dir = wall_avoid_turn_dir_ship(ship_num, gene.offset_inc)
370.                    turn_amt = xpai.anglediff(xpai.self_heading(), turn_dir)
371.                    xpai.self_turn(turn_amt)
372.                    if (abs(turn_amt) < gene.wall_avoid_angle and
373.                        xpai.self_vel() < gene.screen_thrust_speed):
374.                        xpai.self_thrust(1)
375.
376.                # Turn, thrust towards nearest enemy
377.                elif (ship_num2 > -1 and
378.                      not wall_avoid_turn_dir_ship(ship_num2, gene.offset_inc) == -
    1000):
379.                    #print 'turn and thrust towards nearest enemy2'
380.                    turn_dir = wall_avoid_turn_dir_ship(ship_num2, gene.offset_inc)
381.                    turn_amt = xpai.anglediff(xpai.self_heading(), turn_dir)
382.                    xpai.self_turn(turn_amt)
383.                    if (abs(turn_amt) < gene.wall_avoid_angle and
384.                        xpai.self_vel() < gene.screen_thrust_speed):
385.                        xpai.self_thrust(1)
386.
387.                # Turn to radar ship, thrust if not going too fast. Otherwise, shoot.
388.                elif (radar_ship_num > -1 and
389.                      not is_radar_ship_behind_wall(radar_ship_num) and
390.                      xpai.self_vel() > gene.radar_no_thrust_speed):
391.                    #print 'turn to radar ship and thrust if not going too fast 1'
392.                    # enemy 0 on radar, no wall and currently moving fast
393.                    change_heading(xpai.radar_xdir(radar_ship_num))
394.
395.                elif (radar_ship_num > -1 and
396.                      not is_radar_ship_behind_wall(radar_ship_num)):
397.                    #print 'turn to radar ship and thrust if not going too fast 2'
398.                    # enemy 0 on radar, no wall, and not currently moving fast
399.                    change_heading(xpai.radar_xdir(radar_ship_num))
400.                    xpai.self_thrust(1)
401.
402.                elif (radar_ship_num2 > -1 and
403.                      not is_radar_ship_behind_wall(radar_ship_num2) and
404.                      xpai.self_vel() > gene.radar_no_thrust_speed):
405.                    #print 'turn to radar ship and thrust if not going too fast 3'
406.                    # enemy 1 is on radar, no wall, and currently moving fast
407.                    change_heading(radar_ship_num2)
408.
409.                elif (radar_ship_num2 > -1 and
410.                      not is_radar_ship_behind_wall(radar_ship_num2)):
411.                    #print 'turn to radar ship and thrust if not going too fast 4'
412.                    # enemy 1 on radar, no wall, and not currently moving fast
413.                    change_heading(xpai.radar_xdir(radar_ship_num2))
414.                    xpai.self_thrust(1)
415.
416.                elif (radar_ship_num > -1 and
417.                      wall_avoid_turn_dir_radar(radar_ship_num, gene.offset_inc) != -
    1000):
418.                    #print 'turn to radar ship and thrust if not going too fast 5'
419.                    # enemy 0 on radar and behind wall
420.                    turn_dir = wall_avoid_turn_dir_radar(radar_ship_num, gene.offset_inc
    )
421.                    turn_amt = xpai.anglediff(xpai.self_heading(), turn_dir)
422.                    xpai.self_turn(turn_amt)
423.                    if abs(turn_amt) < 5 and xpai.self_vel() < 20:
424.                        xpai.self_thrust(1)
425.
426.
```

```
427.                angle_to_ship_diff = xpai.anglediff(
428.                    xpai.self_heading(), xpai.ship_aimdir(ship_num))
429.                shoot_cond1 = bool(
430.                    ship_num > -1 and
431.                    abs(angle_to_ship_diff) < gene.ship_error_to_shoot and
432.                    not is_ship_behind_wall(ship_num))
433.                #print 'shoot cond1'
434.
435.                angle_to_ship2_diff = xpai.anglediff(
436.                    xpai.self_heading(), xpai.ship_aimdir(ship_num2))
437.                shoot_cond2 = bool(
438.                    ship_num2 > -1 and
439.                    abs(angle_to_ship2_diff) < gene.ship_error_to_shoot and
440.                    not is_ship_behind_wall(ship_num2))
441.                #print 'shoot cond2'
442.
443.                angle_to_rship_diff = xpai.anglediff(
444.                    xpai.self_heading(), xpai.radar_xdir(radar_ship_num))
445.                shoot_cond3 = bool(
446.                    radar_ship_num > -1 and
447.                    abs(angle_to_rship_diff) < gene.radar_error_to_shoot and
448.                    not is_radar_ship_behind_wall(radar_ship_num))
449.                #print 'shoot cond3'
450.
451.                angle_to_rship2_diff = xpai.anglediff(
452.                    xpai.self_heading(), xpai.radar_xdir(radar_ship_num2))
453.                shoot_cond4 = bool(
454.                    radar_ship_num2 > -1 and
455.                    abs(angle_to_rship2_diff) < gene.radar_error_to_shoot and
456.                    not is_radar_ship_behind_wall(radar_ship_num2))
457.                #print 'shoot cond4'
458.
459.                if shoot_cond1 or shoot_cond2 or shoot_cond3 or shoot_cond4:
460.                    #print 'shooting'
461.                    xpai.self_shoot(1)
462.                #print 'after shooting'
463.            elif reset_flag:
464.                reset_all()
465.                pre_life = True
466.            elif not is_done.isSet() and abs(time.time() - start_time) > 120:
467.                global final_fitness
468.                final_fitness = frames + (1000 * kills)
469.                pre_life = True
470.                is_done.set()
471.            else:
472.                xpai.talk(str(frames))
473.                xpai.talk(str(abs(time.time() - start_time)))
474.                pre_life = True
475.
476.        def set_reset(chrom):
477.            global reset_flag
478.            global is_done
479.            load_gene(chrom)
480.            reset_flag = True
481.            is_done.clear()
482.
483.        def get_fitness():
484.            xpai.talk(str(final_fitness))
485.            return final_fitness
486.
487.        def launch(fps):
```

```
488.            # Initialize XPilot
489.            xpai.set_AImain(ai_main)
490.            xpai.setmaxturn(MAX_TURN)
491.            xpai.setargs('-join localhost -port 45%s -name Expert' % fps)
492.            load_gene(chromosome)
493.            thread.start_new_thread(xpai.launch, ())
```

C. Fitness Biasing genetic algorithm and PAL meta controller

```
1.  # Copyright 2010-2011, Connecticut College Computer Science
2.  """PAL GA"""
3.
4.  __author__ = 'Phil Fritzsche <pfritzsche@gmail.com>'
5.
6.  import random
7.  import sys
8.  import time
9.
10. import gabot_timed
11. from gabot_timed import xpai
12.
13. sys.setcheckinterval(0)
14.
15. OUTPUT_POP_FILE = 'pal_results/output_pop_'
16. OUTPUT_BESTS_FILE = 'pal_results/output_best_chroms'
17. OUTPUT_FIT_FILE = 'pal_results/output_fitnesses'
18. OUTPUT_EXT = '.txt'
19.
20.
21. class Struct(dict):
22.     """A dictionary whose values can also be accessed as attributes."""
23.     def __init__(self, **kwargs):
24.         self.update(kwargs)
25.
26.     def __getattr__(self, name):
27.         """Retrieves the value for the given name, if it exists. Returns
28.         None if it does not."""
29.         if name in self:
30.             return self[name]
31.
32.     def __setattr__(self, name, value):
33.         """Sets the value for the given name to the given value."""
34.         self[name] = value
35.
36.
37. def avg(seq):
38.     """Calculates the average of a sequence."""
39.     return sum(seq) / len(seq)
40.
41.
42. def comp(x, y):
43.     """Compares two tuples based on the second item in each. Returns 1 if
44.     x > y, 0 if x == y, and -1 if x < y."""
45.     if x[1] > y[1]:
46.         return 1
```

```
47.     elif x[1] == y[1]:
48.         return 0
49.     else:
50.         return -1
51.
52.
53. class GA(object):
54.     def __init__(self, mutate_rate, chrom_size=72, pop_size=128, pop_file='',
55.                  chrom_set=[0, 1]):
56.         """Public constructor."""
57.         self.mutate_rate = mutate_rate
58.         self.chrom_size = chrom_size
59.         self.pop_size = pop_size
60.         self.chrom_set = chrom_set
61.         self.fitnesses = [0] * self.pop_size
62.         self.biases = [1] * self.pop_size
63.         gabot_timed.launch()
64.         if not pop_file:
65.             self.pop = self.generate_init_pop()
66.         else:
67.             self.pop = self.read_pop(pop_file)
68.
69.     def learn(self):
70.         """Primary learning loop of the GA. Performs the following actions
71.         in an infinite loop:
72.
73.             1. Every 5 iterations, saves the current fitness information.
74.             2. Every 50 iterations, save the best chromosome and the current
75.                 population to a file.
76.             3. Calculates new fitnesses for every chromosome in the population.
77.             4. Stochastically generates a new population.
78.             5. Gets the current best chromosome."""
79.         iterations = 0
80.         self.best = Struct(fitness=0, chrom='')
81.         while True:
82.             iterations += 1
83.             if not iterations % 5:
84.                 self.save_fitness_information(iterations)
85.             if not iterations % 15:
86.                 self.world_sync()
87.             if not iterations % 50:
88.                 self.save_best_and_pop(iterations)
89.             self.calc_fitnesses()
90.             self.pop, self.biases = self.generate_new_pop()
91.             self.best = self.get_best_chrom()
92.             info_file = open('DEBUG_DATA', 'a')
93.             info_file.write('ITERATIONS %s\n' % iterations)
94.             info_file.write('%s\n' % self.fitnesses)
95.             info_file.write('\n%s\n============================\n' % self.biases)
96.             info_file.close()
97.
98.     def world_sync(self):
99.         pop_out = open('population_out', 'w')
100.                for chrom in self.pop:
101.                    pop_out.write('%s\n' % chrom)
102.                pop_out.close()
103.
104.                fits_out = open('fitnesses_out', 'w')
105.                for fitness in self.fitnesses:
106.                    fits_out.write('%s\n' % fitness)
107.                fits_out.close()
```

```
108.
109.                 ok_file = open('to_world', 'w')
110.                 ok_file.write('READY')
111.                 ok_file.close()
112.
113.                 ok_file = open('from_world', 'r')
114.                 while not 'READY' in ok_file.read():
115.                     ok_file.close()
116.                     time.sleep(1)
117.                     ok_file = open('from_world', 'r')
118.
119.                 ok_file.close()
120.                 ok_file = open('from_world', 'w')
121.                 ok_file.close()
122.
123.                 fits_in_file = open('fitnesses_in', 'r')
124.                 self.fitnesses = [float(f) for f in fits_in_file.read().split()]
125.                 fits_in_file.close()
126.
127.                 biases_in_file = open('biases_in', 'r')
128.                 self.biases = [float(b) for b in biases_in_file.read().split()]
129.                 biases_in_file.close()
130.
131.                 info_file = open('DEBUG_DATA', 'a')
132.                 info_file.write('FROM WORLD\n')
133.                 info_file.write('%s\n' % self.fitnesses)
134.                 info_file.write('\n%s\n============================\n' % self.biases)
135.                 info_file.close()
136.
137.             def get_best_chrom(self):
138.                 """Sorts the population by fitness and returns the best chromosome."""
139.                 gen = zip(self.pop, self.fitnesses)
140.                 gen.sort(comp)
141.                 n = len(gen) - 1
142.                 return Struct(chrom=gen[n][0], fitness=gen[n][1])
143.
144.             def generate_init_pop(self):
145.                 """Generates an initial random population."""
146.                 pop = []
147.                 for i in range(self.pop_size):
148.                     chrom = ''
149.                     for j in range(self.chrom_size):
150.                         chrom += str(random.choice(self.chrom_set))
151.                     pop.append(chrom)
152.                 return pop
153.
154.             def read_pop(self, filename):
155.                 """Reads in a population from the specified file."""
156.                 fin = open(filename, 'r')
157.                 file_text = fin.read()
158.                 return file_text.split()
159.
160.             def calc_fitnesses(self):
161.                 """Calculates the fitness of every chromosome in the population."""
162.                 for i in range(self.pop_size):
163.                     gabot_timed.kills = 0
164.                     gabot_timed.frames = 0
165.                     gabot_timed.final_fitness = 0
166.                     gabot_timed.set_reset(self.pop[i])
167.                     #xpai.talk('waiting')
```

```
168.                     gabot_timed.is_done.wait()
169.                     #xpai.talk('done waiting')
170.                     self.fitnesses[i] = self.biases[i] * gabot_timed.get_fitness()
171.                     print self.biases[i]
172.                     print self.fitnesses[i]
173.
174.             def generate_new_pop(self):
175.                 """Generates a new population of chromosomes by doing the following
176.                 enough times to replace every chromosome in the current population:
177.
178.                 1. Chooses crossover point for the two parents.
179.                 2. Chooses two parents.
180.                 3. Mates the parents together.
181.                 4. [Possibly] mutates the children.
182.                 5. Adds the new child to the new population."""
183.                 new_pop = []
184.                 new_biases = []
185.                 for i in range(self.pop_size):
186.                     cross = random.randrange(self.chrom_size)
187.                     parents = self.select_parents()
188.                     child, bias = self.mate_parents(parents, cross)
189.                     mutated_child = self.mutate(child, self.mutate_rate)
190.                     new_pop.append(mutated_child)
191.                     new_biases.append(bias)
192.                 return new_pop, new_biases
193.
194.             def select_parents(self, bin_count=10000):
195.                 """Uses a roulette-wheel style selection to choose two new parents
196.                 based on their fitnesses."""
197.                 parents = Struct()
198.                 partial_sum = 0
199.                 total_fitness = int(sum(self.fitnesses))
200.                 p_mom = random.randrange(total_fitness)
201.                 p_dad = random.randrange(total_fitness)
202.                 for chrom, fit, bias in zip(self.pop, self.fitnesses, self.biases):
203.                     partial_sum += fit
204.                     if not parents.mom and p_mom < partial_sum:
205.                         parents.mom = chrom
206.                         parents.bias_mom = bias
207.                     if not parents.dad and p_dad < partial_sum:
208.                         parents.dad = chrom
209.                         parents.bias_dad = bias
210.                     if parents.dad and parents.mom:
211.                         return parents
212.
213.             def mate_parents(self, parents, cross):
214.                 """Returns a new child resulting from the passed in parents."""
215.                 new_bias = (parents.bias_mom + parents.bias_dad) / 2
216.                 return parents.mom[:cross] + parents.dad[cross:], new_bias
217.
218.             def mutate(self, chrom, mutate_rate):
219.                     """Goes through every bit of the child determining if mutation
220.                     should occur. If so, performs the mutation and returns the new
221.                     child when finished."""
222.                 new_chrom = ''
223.                 for i in range(len(chrom)):
224.                     if random.random() <= mutate_rate:
225.                         choices = list(set(self.chrom_set) - set(chrom[i]))
226.                         new_chrom += str(random.choice(choices))
227.                     else:
228.                         new_chrom += chrom[i]
```

```
229.                return new_chrom
230.
231.            def save_fitness_information(self, its):
232.                    """Saves the current fitness information to a file."""
233.                fout = open(OUTPUT_FIT_FILE + OUTPUT_EXT, 'a')
234.                fout.write('%d, %d, %.3f\n' % (
235.                    its, self.best.fitness, avg(self.fitnesses)))
236.                fout.close()
237.
238.            def save_best_and_pop(self, its):
239.                    """Saves the current best agent and population to a file."""
240.                fout = open(OUTPUT_BESTS_FILE + OUTPUT_EXT, 'a')
241.                fout.write('%d, %s\n' % (its, self.best.chrom))
242.                fout.close()
243.
244.                fout = open(OUTPUT_POP_FILE + str(its) + OUTPUT_EXT, 'w')
245.                for chrom in self.pop:
246.                    fout.write('%s\n' % chrom)
247.                fout.close()
248.
249.        def main():
250.            mutate_rate = 0.01
251.            ga = GA(mutate_rate)
252.            ga.learn()
253.
254.        main()
```

D. Communication script for PAL meta controller and non-simulation agent

```
1.  # Copyright 2010-2011, Connecticut College Computer Science
2.  """PAL GA"""
3.
4.  __author__ = 'Phil Fritzsche <pfritzsche@gmail.com>'
5.
6.  import time
7.
8.  import world_bot_timed
9.
10. world_bot_timed.launch()
11.
12. def handle_communication():
13.     while 1:
14.         ok_file = open('to_world', 'r')
15.         while not 'READY' in ok_file.read():
16.             ok_file.close()
17.             time.sleep(1)
18.             ok_file = open('to_world', 'r')
19.
20.         ok_file.close()
21.         ok_file = open('to_world', 'w')
22.         ok_file.close()
23.
24.         pop_file = open('population_out', 'r')
25.         pop = pop_file.read().split()
26.         pop_file.close()
```

```
27.
28.         fits_file = open('fitnesses_out', 'r')
29.         fits = [float(f) for f in fits_file.read().split()]
30.         fits_file.close()
31.
32.         new_fits = []
33.         biases = []
34.         for i in range(len(pop)):
35.             print '%s, %s' % (i, pop[i])
36.             world_bot_timed.set_reset(pop[i])
37.             world_bot_timed.is_done.wait()
38.             fitness = world_bot_timed.get_fitness()
39.             bias = fitness / fits[i]
40.             if bias < 1:
41.                 bias = 1
42.             elif bias > 2.5:
43.                 bias = 2.5
44.             biases.append(bias)
45.             new_fits.append(fitness)
46.
47.         fits_out_file = open('fitnesses_in', 'w')
48.         for fitness in new_fits:
49.             fits_out_file.write('%s\n' % fitness)
50.         fits_out_file.close()
51.
52.         biases_out_file = open('biases_in', 'w')
53.         for bias in biases:
54.             biases_out_file.write('%s\n' % bias)
55.         biases_out_file.close()
56.
57.         ok_file = open('from_world', 'w')
58.         ok_file.write('READY')
59.         ok_file.close()
60.
61. handle_communication()
```

Works Cited

XPilot Development. 2010. 29 April 2011 <http://www.xpilot.org/development/>.

Xpilot-AI. April 2011. April 2011 <http://xpilot-ai.org/>.

Yannakakis, Georgios N. and John Hallam. "Evolving opponents for interesting interactive computer games." Proceedings of the 8th International Conference on the Simulation of Adaptive Behavior. Los Angeles, CA, USA: SAB 2004, 2004. 499-508.

Allen, Martin, Kristen Dirmaier and Gary Parker. "Real-time AI in Xpilot using reinforcement learning." Proceedings of the 2010 World Automation Congress International Symposium on Intelligent Automation and Control. Kobe, Japan: ISIAC 2010, 2010. 1-6.

Baumgarten, Robin, Simon Colton and Mark Morris. "Combining AI methods for learning bots in a real-time strategy game." International Journal of Computer Games Technology 2009 (2009): 10.

Grefenstette, John J. and Connie Loggia Ramsey. "An approach to anytime learning." Proceedings of the Ninth International Conference on Machine Learning. Morgan Kaufmann, 1992. 189-195.

Hayes-Roth, Frederick. "Rule-based systems." Communications of the ACM September 1985.

Lucas, Simon M. "Estimating Learning Rates in Evolution and TDL: Results on a Simple Grid-World Problem." Proceedings of the 2010 IEEE Congress on Computational Intelligence in Games. Copenhagen, Denmark: CIG 2010, 2010. 372-379.

Parker, Gary and David Arroyo. "The Xpilot-AI Environment." Proceedings of the 2010 World Automation Congress International Symposium on Intelligent Automation and Control. Kobe, Japan: ISIAC 2010, 2010.

Parker, Gary and Karen J. Larochelle. "Punctuated Anytime Learning for Evolutionary Robotics." Proceedings of the World Automation Congress. WAC 2000, 2000. 268-273.

Parker, Gary and Matt Parker. "Using a queue genetic algorithm to evolve Xpilot control strategies on a distributed system." Proceedings of the 2006 IEEE Congress on Evolutionary Computation. Vancouver, BC, Canada: CEC 2006, 2006. 232-237.

—. "Evolving parameters for Xpilot combat agents." Proceedings of the 2007 IEEE Symposium on Computational Intelligence in Games. Honolulu, HI, USA: CIG 2007, 2007. 238-243.

—. "The evolution of multi-layer neural networks for the control of Xpilot agents." Proceedings of the 2007 IEEE Symposium on Computational Intelligence in Games. Honolulu, HI, USA: CIG 2007, 2007. 232-237.

Parker, Gary and Michael Probst. "Using evolutionary strategies for the real-time learning of controllers for autonomous agents in Xpilot-AI." Proceedings of the 2010 IEEE Congress on Evolutionary Computing. Barcelona, Spain: CEC 2010, 2010. 1-7.

Parker, Gary. "Punctuated Anytime Learning for Hexapod Gait Generation." Proceedings of the 2002 IEEE / RSJ International Conference on Intelligent Robots and Systems. EPFL, Switzerland: IROS 2002, 2002. 2664-2671.

—. "Punctuated Anytime Learning for Hexapod Gait Generation." Proceedings of the 2002 IEEE/RSJ International Conference on Intelligent Robots and Systems. EPFL, Switzerland: IROS 2002, 2002.

Poole, David, Alan Mackworth and Randy Goebel. Computational Intelligence: A Logical Approach. New York: Oxford University Press, 1998.

Priesterjahn, Steffen, et al. "Evolution of human-competitive agents in modern computer games." Proceedings of the 2006 IEEE Congress on Evolutionary Computation. Vancouver, BC, Canada: CEC 2006, 2006. 777-784.

Stabell, Bjørn and Ken Ronny Schouten. "The story of XPilot." Crossroads December 1996: 3-6.

Stanley, Kenneth O., et al. "Real-time evolution of neural networks in the NERO video game." AAAI'06 Proceedings of the 21st National Conference on Artificial intelligence. Ed. Anthony Cohn. Boston, MA, USA: AAAI 2006, 2006. 1671-1674.

Rule-based systems and identification trees. 1 May 2011 <http://ai-depot.com/Tutorial/RuleBased.html>.

Rosenblatt, Frank. The Perceptron: a perceiving and recognizing automaton. Cornell Aeronautical Laboratory. New York, 1957.