RL-Glue 3.0 Docs

Adam White

September 30, 2008

Contents

1 Introduction

1.1 Purpose of Document

This document has been laid out so that it can be used for:

1. **RL-Glue what?** learning about RL-Glue at an abstract level
2. **Compatibility:** making existing C/C++ agents and environments work with RL-Glue
3. **Plugging agents and environments together:** how to write an experiment programs
4. **Coding things up:** implementing C/C++ agents, environments and experiments
5. **RL-Glue manual:** as a complete reference Guide for RL-Glue

Recently (September 08) the RL-Glue Project has been split into the RL-Glue Project and RL-Glue Extensions Project. The RL-Glue Project now only includes the RL-Glue interface and plugs for C/C++ agents, environments and experiment programs. The RL-Glue Extensions Project contains codecs for Java, Python and Matlab. The Extensions Project contains the multi-language support code that used to be part of RL-Glue. As such, this document only contains technical details for writing C/C++ programs. See the codec documentation for language specific details on how to implement agents, environments and experiment programs in Java, Python and Matlab.

1.2 How to Use This Document

This document as been subdivided to reflect the five purposed described above. To learn about the major components of RL-Glue and a description of how those components interact see Section 2. To learn how to make an environment and agent programs compatible with RL-Glue we recommend sections 4.1, 4.3, 5.1 and 5.3. Sections 4.1 and 5.1 describe only the mandatory functions that RL-Glue environments and agents must implement. Sections 4.3 and 5.3 describe how to implement environments and agents in C and C++. To learn about experiment programs and how they interact with RL-Glue see Section 6. For all command reference and advanced queries see advanced environment functions in Section 4.2, advanced agent functions in Section 5.2 and the command and function reference in Section 8.

Error messages, FAQ and glossary can be found in sections 7,9 and 10.

2 RL-Glue Concepts

A large part of studying and researching reinforcement learning is experimentation. When writing an agent, ensuring it makes exploratory moves to discover the world is important to achieving an optimal policy. It is similarly important that experimenters are easily able to "explore" new algorithms and ideas without the prohibitive cost of writing the necesary experimentation code. One of the functions of RL-Glue is to simplify and speed up the process of writing an experiment so that every idea can be tested.

Another important part of learning is evaluation and improvement. With agents, learning is often a cycle of policy and representation evaluation and improvement. An agent will roam the world and gain more information about the states it encounters. The agent then uses this new data to evaluate how accurate the value function was and adjusts its behaviour (policy) based on the new value function. Similiarily, in research and development it is important to look at other work being done in the field, compare your own performance and then improve. One goal for RL-Glue is to provide a consistent tool for running and comparing varied agents and environments from diverse sources. A common problem for reinforcement learning researchers arrises when an experimenter wishes to compare their own work with previously established results. Pre-RL-Glue, the solution was often to reverse engineer code for the experiment based on the results and environment/agent descriptions provided in papers. When an author provided their own source, there was still the issue of deciphering the original code and piecing in a new agent or environment. With RL-Glue, an author can make the necessary RL-Glue agent/environment/experiment programs available to the public (or use a public copy of an environment from our library [1] such that another author can rerun the original experiment and easily plug in their own code to compare performance. Competitions for agents using RL-Glue have been run at the NIPS conference in recent years, further exemplifying the utility of RL-Glue to the research community.

RL-Glue is both a set of ideas and standards, as well as a software implementation. In theory, RL-Glue is a protocol for all RL researchers to follow. Having this very simple standard of necessary functions facilitates the exchange and comparison of agents and environments without limiting their abilities. As software, RL-Glue is functionally a test harness to "plug in" agents, environments and experiment programs (previously titled benchmarks) without having to continually rewrite the connecting code for these pieces. An experiment program is some very simple code stating how many times to run an agent in an environment and what data should be extracted from the agent

[1] `http://rlai.cs.ualberta.ca/RLR/index.html`

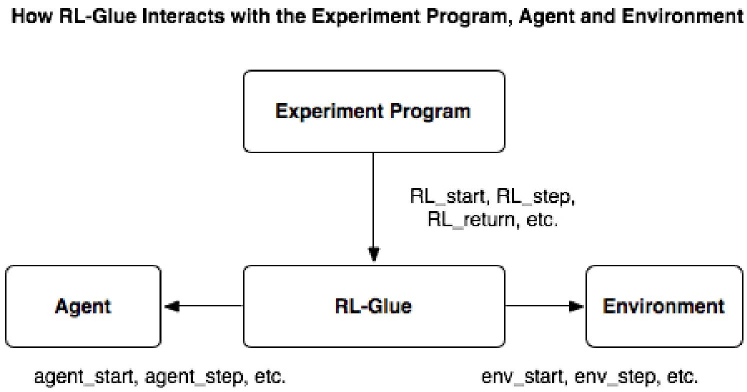

Figure 1: The RL-Glue Standard. Arrows indicate function call direction.

and environment. Provided the agent, environment, and experiment program follow the RL-Glue protocol by implementing the few necessary functions, they can easily be plugged in with the RL-Glue code to have an experiment running quite effortlessly. Figure 2 is a diagram which shows how function calls work in RL-Glue.

The Experiment Program contains the "main function" which will then make all of it's requests for information through RL-Glue. These requests are usually related to setting up, starting and running the experiment and then gathering data about the agent's performance. The experiment program should never have access to the agent or environment directly, all contact should always go through the RL-Glue interface first. There is also no direct contact between the agent and the environment. Any information the agent or environment returns is passed through RL-Glue to the module which needs it. With the newest version of RL-Glue, the agent and environment and experiment can even be written in different languages and/or compiled on different computers and be run over the internet. In this case, the arrows can be thought of as the sockets which connect the pieces of RL-Glue over the internet.

2.1 Agents, Environments and Experiment Programs

Understanding agents and environments is a fundamental part of understanding reinforcement learning. A more detailed explanation of reinforcement learning, and its definitions for agents and environments, can be obtained from Reinforcement Learning: An Introduction (Sutton, Barto. 1998). The agent, when distilled to basics, is both the learning algorithm and the decision maker. For RL-Glue's purpose, the agent only needs to decide which action to take at every step. The environment should store all the relevant details of the world of your experiment. This should include knowledge of the current state representation (the observation) as well as some method for determining state transitions and rewards. When writing code for an experiment, RL-Glue has been structured to place separation between the agent and environment. This division of the agent and environment both helps create modularized code and is theoretically desirable.

The experiment program is familiar to anyone who has written code to run an reinforcement learning experiment on their own. Akin to the typical main function in many reinforcement learning experiments, an RL-Glue experiment program is a control loop which runs the agent through the environment **x** number of times, perhaps doing **y** trials of these **x** episodes, all the while gathering data about how efficiently the agent has behaved or how quickly it has learned. RL-Glue provides several functions (Section 8) to assist in writing an experiment program.

When writing an agent , environment or experiment program for RL-Glue it is necessary to adhere to the interfaces (the RL-Glue Protocol) set out in Section 8.

3 Procedures

3.1 How to Build RL-Glue

3.2 How to Run a Learning Experiment

3.3 Getting Agent and Environment Programs

3.4 Using Codecs and Multi-language Support

4 Writing an Environment

4.1 Essential Components Of A RL-Glue Environment

Every RL-Glue environment must define and initialize the action and observation types (rewards are always real-valued scalars) and define the env_start and env_step functions.

4.1.1 Observation and Action Encoding

The representation of the observation and the action must be one of the following:

- an integer
- a double
- a character
- array of integers
- array of doubles
- array of characters

Although this seems restrictive, in practice it should be easy to represent many observation and action types in this format. In a gridworld, for example, the action can be a single integer with values 0-3 (for N,S,E,W) and the observation can also be an integer for the agent's grid position (which is also the state of the environment, in this case). In Mountain Car, the actions are discrete (0-2) and the observation is the cars position and velocity (both real numbers). The action can be represented with and integer and the observation can be a two dimensional double array.

Details on how to encode observation and action types in C and C++ is described in Section 4.3.

4.1.2 Start and Step Functions

Writing env_start is very simple. The function takes no input and simply returns an observation. The env_start function signifies the beginning of an episode; env_start chooses the initial state of the environment and returns the corresponding observation. For example, the following pseudo code selects a random start state for a grid world and returns the observation:

1. **env_start**→ observation
2. state = rand()*num_states
3. set observation equal to state
4. **return** observation

The sections describing the language specific details for environments will explain line 3.

The final essential piece of an RL-Glue environment is the env_step function. The env_step function must take an action as input and produce a observation, reward and a termination flag as output. In most RL problems the env_step function updates the internal state of the environment, tests for end of episode and returns the new observation of state and current reward. In other words, step function encodes the state transition and reward functions. Keeping with the grid world example, the following step function would be a valid env_step function:

1. **env_step**(Action a) → reward, observation, flag
2. newState = updateState(a, oldState)
3. flag = isTerminal()
3. reward = -1
4. set observation equal to newState
5. oldState = newState
6. **return** reward, observation, flag

Here we assume the existence of a state update function and an isTerminal function that checks if the current state is a terminal state.

So thats it. Just define the types of actions and observations and write two functions and you have a valid RL-Glue compatible environment. In later sections we will go over advanced environment functions and describe the syntactic details of actually coding an environment in several different programming languages.

4.2 Additional Components Of A RL-Glue Environment

So far we have only scratched the surface of what kinds of environments you can write in RL-Glue. Additional environment function can be written to initialize data structures, get and set the state of the environment, get and set the random seed and send generic string messages to the environment. Before we go on describing these functions it is useful to understand the task specification language that is used in RL-Glue to encode basic information about environments.

4.2.1 Task Specification Language

In an effort to provide the agent writer with simple and concise information about the environment a Task_specification is passed from the environment, through the interface, to the agent. The environment's init method (env_init) encodes information about the problem in a ASCII string. The string is then passed to the agent's init method (agent_init). This information can also be used to check that the agent and environment are suitable for each other. A few example Task_specifications are provided below.

The agent is responsible for parsing any relevant information out of the Task_specification in the init method. A generic Task_specification parsing function is provided with RL-Glue 2.0 for all C/C++ users. This simple parser will return a structure containing the information encoded in the Task_specification, such as the Observation and Action dimensions, arrays of Observation and Action variable ranges, and arrays of Observation and Action variable types. More information about the parser and the parsed task spec struct can be found here.

The Task_specification is stored as a string with the following format:

> "V:E:O:A:R"

For example, this is a sample task_specification provided as one of the examples below:

> "2:e:1_[i]_[0,N-1]:1_[i]_[0,3]:[-1,0]"

The V corresponds to the version number of the task specification language. E corresponds to the type of task being solved. It has a character value of 'e' if the task is episodic and 'c' if the task is continuing. O and A correspond to Observation and Action information respectively. Finally, the R corresponds to the range of rewards for the task. Within each of O, A and R a range can be provided, however if the values are unknown or infinite in magnitude, two special input values have been defined.

The format of O and A are identical. We will describe the form of O only. O contains three components, separated by underscore characters ("_") :

> #dimensions_dimensionTypes_dimensionRanges

#dimensions is an integer value specifying the number of dimensions in the Observation space.

dimensionTypes is a list specifying the type of each dimension variable. The dimensionTypes list is composed of #dimensions components separated by comma characters (",") within square brackets ([x1,x2,x3,..., xn] where xi represents the ith value). Each comma-separated value in the list describes the type of values assigned to each Observation variable in the environment. In general, Observation variables can have one of the following 2 types:

'i' - integer value

'f' - float value

Thus a dimensionTypes list corresponding to an Observation space with 1 dimension has the following form:

[a] where a is an element of ['i','f']

So a dimensionTypes list with one integer value would be: [i]

An Observation space with 2 dimensions would have a dimensionTypes with the following form:

[a,b] where a and b are elements of ['i','f']

indicating the value type of the first (a) and second (b) Observation variables. Thus a three dimensional Observation with one float, integer dimension and another float dimension would have the following dimensionTypes:

"[f,i,f]"

The dimensionRanges is a list specifying the range of each dimension variable in the Observation space. The dimensionRanges is composed of #dimensions components separated by underscore characters. Each dimensionRanges component specifies the upper and lower bound of values for each Observation variable. If the bounds are unknown or unspecified, you can leave an empty space in the place of a value. If the bounds are positive or negative infinity, you can use inf or -inf to represent your range. These can be used in combination. For example one valid range could be an unknown lower bound and infinite upper bound, or a lower bound of -inf and an upper bound of 1. You can be as precise (though you must be accurate) as you wish. A dimensionRanges corresponding to an Observation space with a single dimension variable would have the following form:

[O1MIN, O1MAX]

So a dimensionRanges list for binary dimension varaible would be: [0,1] A dimensionRanges list for variable with no upper or lower bound unspecified would be: [] or [,] (both are valid). A dimensionRanges with one or two unbounded value can take on a value of inf or -inf. Eg [0, inf] or [-inf,1] or [-inf,inf].

An Observation space with 2 dimensions would have a dimensionRanges with the following form:

[O1MIN, O1MAX]_[O2MIN, O2MAX]

indicating the minimal and maximal values of Observation variables O1 and O2 respectively. This definition can be then trivially extended to Observation spaces with N dimensions.

NOTE: the dimensionRanges of an Observation space with 1 or more unbounded values may not be representable in this way. An unbounded value has no minimal or maximal range. Thus, we simply do not specify the range in the dimensionRanges for any Observation variables with unbounded values. For example, consider a problem with 3 Observation dimensions. The first and third Observation variables have interval values and the second has unbounded ratio value. The corresponding dimensionRanges for this problem is encoded as:

[O1MIN, O1MAX]_[,]_[O3MIN, O3MAX]

indicating the minimal and maximal values of Observation variables O1 and O3.

The format of A (Action space information) is identical to that of O (Observation space information) and thus the definitions above hold for Action spaces.

Lastly the R (Reward space information) is merely a range specifier. By the Reward Hypothesis there is only ever one reward signal (which in RL-Glue is always a floating point number) so the #dimensions and dimensionType information becomes meaningless. The reward range can again be specified to be unknown or infinite in the same manner as the Observation ranges. A rewardRange follows the following form:

[rewardMin, rewardMax]

In the case of a reward with rewards -1 or 0 the rewardRange would appear as such: [-1,0]. If no lower bound was known and the upper bound was positive infinity, the rewardRange would appear as such: [,-inf]

4.2.2 Example Task_Specification

Consider a simple gridworld with Actions North, South, East and West and a single dimension Observation of grid position. If we encode actions as 0, 1 ,2 ,3 and position as an integer between 0 and N-1, we get the following Task_specification:

"2:e:1_[i]_[0,N-1]:1_[i]_[0,3]:[-1,0] "

This Task_specification provides the following information:

- RL-Glue version 2.0 supported
- the task is episodic
- Observation space has one dimension
- the Observation variable has integer values (discrete state)

- range of Observation variable is 0 to N-1
- Action space has one dimension
- the Action variable has integer values (discrete actions: tabular)
- range of Action variable is 0 to 3
- range of the rewards is -1 to 0

4.2.3 Environment initialization and cleanup

Most environments need to store an internal state representation and therefore many environment programs you write will need to allocate and deallocate data structures before and after a learning experiment. The env_init function allocates any global data structures and variables that will be accessed by the start and step functions. For example, the env_init method might initialize the tabular state variable to zero and allocate a *numStates X numStates* state transition array. The env_cleanup function usually deallocates or frees anything allocated in env_init.

4.2.4 Environment Message

Once in a while you will find yourself wishing there was an environment function that was part of RL-Glue. The env_message function allows you to exactly this: you can send a string message to the environment and it can respond with a string. For example to disable random starting states:

1. env_message(inMessage) → outMessage
2.   **if** inMessage == "turnOffRandomStarts"
3.   randStarts = false
4.   **end**
5. **return** "1"

4.2.5 Environment Get and Set Sate

It is often necessary to record the state of the environment and then reset the environment to a particular state to evaluate learning performance. For example, one might want to repeatedly evaluate the performance of a stochastic policy starting from a finite set of interesting states. The env_get_state_key function returns a state_key that can be used to restore the environment to the current state when the key was created. The state_key is the same data type as the observations and actions; the state_key must be an int, double, character, int array, double array or character array. The env_set_state takes a state_key as input and simply restores the environment to the state encoded in the key. The state_key can be as simple as a integer state label, in the case of a gridworld, or a some function of an array, in the case of a simulated robot navigation task.

The get and set state functions are meant to be used during a single learning experiments not across runs. For example, the following experiment is valid:

1. initialize RL-Glue
2. **until** observation “cliff” **do** next step of episode
3. get state key form environment and store in stateKey
4. **for** 100 steps
5. set environment state to stateKey
6. run an episode

Storing a state key in a file and then using that state key in a different experiment program or later time is outside the intended usage of the env_get_state and env_set_state functions. It might work with some RL-Glue environments and not others.

4.2.6 Environment Get and Set Random Seed

It can be useful to record and set the seed of the random number generator. Although the random seed can be well thought of as part of the state of the environment there may be times when it is more convenient access the random seed only with env_get_random_seed and env_set_random_seed functions. The environment get and set random functions return and take as input a random_seed_key, like the state_key for setting states.

Like the get and set state functions, env_get_random_seed and env_set_random_seed functions should not be used to store and set the random seed between distinct runs of RL-Glue.

4.3 C/C++ Environments

what files to include
declaring, initializing and using the obs and action types
special considerations
sample environment

4.4 Java, Python and Matlab Environments

get the codec and its docs

5 Writing an Agent

5.1 Essential Components Of A RL-Glue Agent

An agent program is fully compatible with RL-Glue if it implements three functions: agent_start, agent_step and agent_end.

5.1.1 Action Types

The three agent functions take observations and rewards as input and return actions. The observations and rewards are created by the environment, so the agent program needs to only read their values. The actions, however, must be defined by the agent.

5.1.2 Agent Start

The agent_start function selects the first action at the beginning of an episode based on the first observation of the environment. The agent_start function does not receive a reward as input; agent_start usually contains no learning update code. For example, the following function selects the first action based on the current value function estimate:

1. agent_start (observation) → action
2.   **for** each action i
3.   **if** highest valued action (V(observation,i))
4.   **then** store i as maxAction
5.   **set** action equal to newAction
6. **return** action

The following sections will describe how to perform step 5 in several different programming languages.

5.1.3 Agent Step

The agent_step function encodes the heart of the agents' learning algorithm and action selection mechanism. At a minimum the step function must return an action every time it is called. In most learning agents, the step function queries the agent programs action selection function and performs a learning update based on the input observation and reward. The following agent_step function does a SARSA update on a tabular value function Q:

agent_step(reward, observation) → action
    newAction = egreedy(observation)
    QofOld = Q(oldState,oldAction)
    QofNew = Q(observation,newAction)
    Q(oldState,oldAction) = QofOld + α[reward + γQofNew - QofOld]
    oldState = observation
    oldAction = newAction
    set action equal to newAction
return action

Notice that the agent program must explicitly store the observation and action from the previous time step. RL-Glue does not make the history of actions, observations and rewards available to the agent or environment.

5.1.4 Agent End

In episode task the environment enters a terminal state that ends the episode. RL-Glue responds to the end of an episode by calling the agent_end function; passing the reward produced on the last transition to the agent and signaling the end of the current episode. The agent_end function usually performs a final learning update based on the last transition and any other end-of-episode routines, such as clearing eligibility traces. If the environment is non-episodic RL-Glue will not query agent_end.

The agent_end function does not receive the final observation from the environment. In many learning problems this is of no consequence because the agent does not make a decision in the terminal state. If, however, the agent were learning a model of the environment, information about the final transition would be important. In this case, it is recommended that the environment be augmented with a terminal state that has a reward of zero on the transition into it. This choice was made to keep the RL-Glue interface as minimal and light-weight as possible.

5.2 Additional Components Of A RL-Glue Agent

5.2.1 Agent initialization and cleanup

Agent programs, like environments, often need to allocate and free various data structures. The agent_init and agent_cleanup function do not take any input or return an data and are called at the beginning and end of a learning experiment, respectively.

5.2.2 Agent Message

The agent_message function isused to send an arbitrary string message to the agent program. This function can be used to change agent parameters, notify the agent that the exploration phase is over, and request the name of the agent program, for example.

5.3 C/C++ Agents

5.4 Java, Python, and Matlab Agents

get the right codec

6 Writing an Experiment

Usually the shortest and easiest part of writing your first learning experiment is writing the experiment program. The experiment program has no interface to implement and is mostly comprised of calls to the already existing RL-Glue functions. The experiment program has four main duties: a) start the experiment b) specify how many times to run the experiment c) extract data and possibly analyze d) end the experiment and clean up. One thing to note is that it is only the RL-Glue interface functions available to the experiment program. No agent or environment implemented functions should be directly accessed by the experiment program.

6.1 Basic Experiment Programs

At a minimum the experiment program must call RL_init and RL_cleanup and execute several time steps of agent-environment interaction. The following pseudo code represents a simple experiment program.

1. RL_init()
2. RL_start()
3. steps=0
4. terminal=false
5. **while** steps $<$ 100 **and not** terminal
6.   terminal,reward,observation,action = RL_step()
7. RL_cleanup()

This experiment program initializes the agent and environment (RL_init), calls the start functions of the agent and environment (RL_start) and then executes a 100 or less step episode.

The RL_step function calls the env_step function passing it the most recent agent action (in this case from agent_start). The env_step function returns the new observation, reward and terminal flag. If the flag **is not** set the agent_step function is called with the new observation and reward as input arguments. The action returned by agent_step is stored by RL-Glue until the next call to RL_step. If the flag **is** set, the agent_end function is called with the reward as input. This process continues until either the flag is set or 100 steps are completed.

Using the RL_step function gives the experiment program designer access to all the data produced during an episode; however, it is often more convient to use the RL_episode function when step-level control is not needed. Lines 5 and 6, in the above experiment program, can be replaced by a single call to RL_episode(100). If the input to RL_episode is zero, control will return to the experiment program if and only if the environment enters a terminal state.

The RL_step function allows the experiment program to record/sum/average the reward, but the RL_episode function returns no values. The RL_return and RL_num_steps functions allow the experiment program to obtain reward and number of steps taken. Specifically, RL_return returns the sum of rewards accumulated during the current or most recently completed episode. The

RL_num_steps returns the number of steps elapsed during the current or most recently completed episode.

Putting these new functions together we can write a more useful experiment program:

1. RL_init()
2. theReturn = 0
3. **for** 1 = 1:100
4.   RL_episode(1000)
5.   theReturn += RL_return()
6. **Print** theReturn/100
7. RL_cleanup()

The above experiment program runs 100 episodes, each with max length 1000, and computes the average cumulative reward per episode.

6.2 Advanced Experiment Programs

As you know from previous sections (about agent and environment programs) there are several optional agent and environment functions that provide more advanced control. These functions are access through calls to the RL-Glue interface from an experiment program. For example, to send a message to the agent use RL_agent_message. To get the state key from the environment use RL_get_state. The pattern is simple to follow:

- RL_set_state() → env_set_state()
- RL_get_random_seed() → env_get_random_seed()
- RL_set_random_seed() → env_set_random_seed()
- RL_env_message() → env_message

We can now produce more advanced experiment programs that would be used in reinforcement learning research:

1. RL_init()
2. numSteps = 0
3. **for** 1 = 1:1000
4.   RL_episode(1000)
5. RL_agent_message("freezeAgentPolicy")
6. **for** 1 = 1:100
7.   RL_episode(1000)
8.   numSteps += RL_num_steps()
9. **Print** numSteps/100
10. RL_cleanup()

6.3 C/C++ Experiments

6.4 Java, Python, Matlab and Experiments

7 Error Messages

8 Command and Function Reference

8.1 Flags, Environment Parameters and Input Arguments

8.2 Agent Functions

Every agent must define all of the following routines. Note these functions are only accessed by the RL-Glue. Experiment programs should not try to bypass the Glue and directly access these functions.

agent_start: agent_start(first_observation) → first_action
Given the first_observation (the observation of the agent in the start state) the agent must then return the action it wishes to perform. This is called once if the task is continuing, else it happens at the beginning of each episode.

agent_step: agent_step(reward, observation) → action
This is the most important function of the agent. Given the reward garnered by the agent's previous action, and the resulting observation, choose the next action to take. Any learning (policy improvement) should be done through this function.

agent_end: agent_end(reward)
If the agent is in an episodic environment, this function will be called after the terminal state is entered. This allows for any final learning updates. If the episode is terminated prematurely (ie a benchmark cutoff before entering a terminal state) agent_end is NOT called.

agent_init: agent_init(task_specification)
This function will be called first, even before agent_start. The task_specification is a description of important experiment information, including but not exclusive to a description of the state and action space. The RL-Glue standard for writing task_specification strings is found here. In agent_init, information about the environment is extracted from the task_specification and then used to set up any necessary resources (for example, initialize the value function to a prelearning state).

agent_cleanup: agent_cleanup()
This function is called at the end of a run/trial and can be used to free any resources which may have allocated in agent_init. Calls to agent_cleanup should be in a one to one ratio with the calls to agent_init.

agent_freeze: agent_freeze()
Signals to the agent that training has ended. Requests that the agent freeze its current policy and value function (ie: stops learning and exploration).

agent_message:agent_message(input_message) → output_message
The agent_message function is a jack of all trades and master of none. Having no particular functionality, it is up to the user to determine what agent_message should implement. If there is any information which needs to be passed in or out of the agent, this message should do it. For example, if it is desirable that an agent's learning parameters be tweaked mid experiment, the author could establish an input string that triggers this action. Likewise, if the author wished to extract a representation of the value function, they could establish an input string which would cause agent_message to return the desired information.

8.3 Environment Functions

Every environment must define all of the following routines. Note these functions are only accessed by the RL-Glue. Experiment programs should not try to bypass the Glue and directly access these functions.

env_start: env_start() → first_observation
For a continuing task this is done once. For an episodic task, this is done at the beginning of each episode. Env_start assembles a first_observation given the agent is in the start state. Note the start state cannot also be a terminal state.

env_step: env_step(action) → reward, observation, terminal
Complete one step in the environment. Take the action passed in and determine what the reward and next state are for that transition.

env_init: env_init() → task_specification
This routine will be called exactly once for each trial/run. This function is an ideal place to initialize all environment information and allocate any resources required to represent the environment. It must return a task_specification which adheres to the task specification language. A task_specification stores information regarding the observation and action space, as well as whether the task is episodic or continuous.

env_get_state: env_get_state() → state_key
The state_key is a compact representation of the current state of the environment such that at any point in the future, provided with the state_key, the environment could return to thatstate. Note that this does not include the agent's value function, it is merely restoring the details of the environment. For example, in a static grid world this would be as simple as the position of the agent.

env_set_state: env_set_state(state_key)
Given the state_key, the environment should return to it's exact formation when the state_key was

obtained.

env_get_random_seed: env_get_random_seed() → random_seed_key
Saves the random seed object used by the environment such that it can be restored upon presentation of random_seed_key.

env_set_random_seed: env_set_random_seed(random_seed_key)
Sets the random seed used by the environment. Typically it is advantageous for the experiment program to control the randomness of the environment. Env_set_random_seed can be used in conjunction with env_set state to save and restore a random_seed such that the environment will behave exactly the same way it has previously when it was in this state and given the same actions.

env_cleanup: env_cleanup()
This can be used to release any allocated resources. It will be called once for every call to env_init.

env_message:env_message(input_string) → output_string
Similar to agent_message, this function allows for any message passing to the environment required by the experiment program. This may be used to modify the environment mid experiment. Any information that needs to passed in or out of the environment can be handled by this function.

8.4 Interface Routines Provided by the RL-Glue

The following built in RL-Glue functions are provided primarily for the use of the experiment program writers. Using these functions, the experiment program gains access to the corresponding environment and agent functions. The implementation of these routines are to be standard across all RL-Glue users. To ensure agents/environments/experiment programs can be exchanged between authors with no changes necessary, users should not change the RL-Glue interface code provided.

To understand the following, it is helpful to think of an episode as consisting of sequences of observations, actions, and rewards that are indexed by time-step as follows:

o0, a0, r1, o1, a1, r2, o2, a2, ..., rT, terminal_observation

where the episode lasts T time steps (T may be infinite) and terminal_observation is a special, designated observation signaling the end of the episode.

RL_init()
    agent_init(env_init())

This initializes everything, passing the environment's task_specification to the agent. This should be called at the beginning of every trial.

RL_start() → o0, a0
    global upcoming_action

```
    o = env_start()
    a = agent_start(o)
    upcoming_action = a
return o,a
```

Do the first step of a run or episode. The action is saved in upcoming_action so that it can be used on the next step.

```
RL_step() → rt, ot, terminal, at
    global upcoming_action
    r,o,terminal = env_step(upcoming_action)
    if terminal == true
        agent_end(r)
        return r, o,terminal
    else
        a = agent_step(r, o)
        upcoming_action = a
    return r, o, terminal, a
```

Take one step. RL_step uses the saved action and saves the returned action for the next step. The action returned from one call must be used in the next, so it is better to handle this implicitly so that the user doesn't have to keep track of the action. If the end-of-episode observation occurs, then no action is returned.

```
RL_episode(steps)
    num_steps = 0
    o, a = RL_start()
    num_steps = num_steps + 1
    list = [o, a]
    while o ≠ terminal_observation
        if(steps ≠ 0 and num_steps ≥ steps)
            end
        else
            r, o, a = RL_step()
            list = list + [r, o, a]
            num_steps = num_steps + 1
    agent_end(r)
```

Do one episode until a termination observation occurs or until steps steps have elapsed, whichever comes first. As you might imagine, this is done by calling RL_start, then RL_step until the terminal observation occurs. If steps is set to 0, it is taken to be the case where there is no limitation on the number of steps taken and RL_episode will continue until a termination observation occurs. If no terminal observation is reached before num_steps is reached, the agent does not call agent_end, it simply stops.

RL_return() → return

Return the cumulative total reward of the current or just completed episode. The collection of all the rewards received in an episode (the return) is done within RL_return however, any discounting of rewards must be done inside the environment or agent.

RL_num_steps() → num_steps

Return the number of steps elapsed in the current or just completed episode.

RL_cleanup()
    env_cleanup()
    agent_cleanup()

Provides an opportunity to reclaim resources allocated by RL_init.

RL_set_state(State_key)
    env_set_state(State_key)

Provides an opportunity to reset the state (see env_set_state for details).

RL_set_random_seed(Random_seed_key)
    env_set_random_seed(Random_seed_key)

Provides an opportunity to reset the random seed key (see env_set_random_seed for details).

RL_get_state() → State_key
    return env_get_state()

Provides an opportunity to extract the state key from the environment (see env_get_state for details).

RL_get_random_seed() → Random_seed_key
    return env_get_random_seed()

Provides an opportunity to extract the random seed key from the environment (see env_get_random_seed for details).

RL_agent_message(input_message_string) → output_message_string
    return agent_message(input_message_string)

This message passes the input string to the agent and returns the reply string given by the agent. See agent_message for more details.

RL_env_message(input_message_string) → output_message_string
    return env_message(input_message_string)

This message passes the input string to the environment and returns the reply string given by the environment. See env_message for more details.

9 Frequently Asked Questions

10 Glossary