RL-Glue 3.0 Overview

Adam White :: awhite@cs.ualberta.ca

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:** help convert existing and environments work with RL-Glue
3. **Plugging agents and environments together:** how to write experiment programs
4. **What function do I use to:** quick function reference for RL-Glue

In September 2008, the RL-Glue Project was split into two projects: RL-Glue and RL-Glue Extensions.

RL-Glue now only includes the RL-Glue interface and plugs for direct-compile C/C++ agents, environments and experiment programs.

The RL-Glue Extensions Project contains codecs that allow C/C++, Java, Python, Matlab, and Lisp agents/environments/experiments to use RL-Glue. This multi-language support was previously bundled with RL-Glue. The reason for the split was partially to separate the technical details of using RL-Glue with a particular language from the high level overview of what RL-Glue does.

This document is the high-level overview document: it contains contains **NO** implementation specific technical details for writing programs.

Please refer to the RL-Glue technical manual and manuals for specific codecs for language specific details on how to implement agents, environments and experiment programs.

1.2 How to Use This Document

This document as been divided to reflect the purposes 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 environment and agent programs compatible with RL-Glue we recommend sections 3.1 and 4.1. Sections 3.1 and 4.1 describe only the mandatory functions that RL-Glue environments and agents must implement. Sections 3.2 and 4.2 describe advanced environment and agent functions. To learn about experiment programs and how they interact with RL-Glue see Section 5. For quick function reference see Section 6. Frequently asked questions can be found in Section 8. A summary of and explanations for all changes from RL-Glue 2.X to RL-Glue 3.0 can be found in Section 7.

RL-Glue uses naming conventions and definitions from Sutton and Barto's text: "Reinforcement Learning: An Introduction". This text is available for free online: `http://www.cs.ualberta.ca/~sutton/book/the-book.html`.

2 RL-Glue Concepts

A large part of studying and researching reinforcement learning is experimentation. When you write an agent, you should ensure the agent makes exploratory moves to discover more about the world. Similarly, it is important that you are able to "explore" new algorithms and ideas without having to create the experiment code for each test. One of the goals of RL-Glue is to simplify and accelerate the process of writing an experiment so that many ideas can be easily tested.

In machine learning research, 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 using and comparing agents and environments from diverse sources. A common problem for

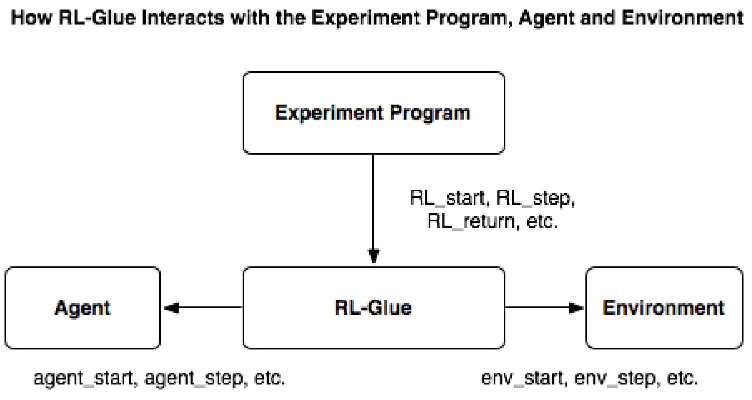
How RL-Glue Interacts with the Experiment Program, Agent and Environment
Experiment Program
RL_start, RL_step, RL_return, etc.
Agent
RL-Glue
Environment
agent_start, agent_step, etc.
env_start, env_step, etc.

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

researchers arises when they try compare their work with previously published results.

Before RL-Glue, the solution was often to reverse engineer code for the experiment based on the results and (often incomplete) implementation descriptions that had been published. Even code was released to the public, it was often still a challenge to understand adapt the original code. Now, you can make the necessary RL-Glue agent/environment/experiment programs available to the public such that another experimenter can reproduce your original experiment and easily experiment with their own code to compare performance. Several recent reinforcement competitions, at NIPS and ICML have used RL-Glue for benchmarking participant submissions, 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 the reinforcement learning community 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 without having to continually rewrite the connecting code for these pieces. An experiment program is, very simply, code stating how many times to run an agent in an environment and what data should be extracted this interaction. 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 1 is a diagram which shows how function calls work in RL-Glue.

The Experiment Program contains the “main function” which will make all of the 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 can never interact with the agent or environment directly: all contact goes through the RL-Glue interface. 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.

2.1 Agents, Environments and Experiment Programs

Understanding the semantics we ascribe to *agents*, *environments*, and *experiments* is a fundamental part of understanding RL-Glue.

In RL-Glue, the *agent* is both the learning algorithm and the decision maker. The agent decides which action to take at every step.

The *environment* is responsible for storing all the relevant details of the world, or problem of your experiment. The environment generates the observations/states/perceptions that are provided to the agent, and also determines the transition dynamics and rewards.

The experiment is the intermediary which (through RL-Glue) controls all communication between the agent and environment. This structured separation is by design, division of the agent and environment both helps create modularized code and captures our intuitions about how much the agent and environment should "know" about each other.

The experiment program will be familiar to anyone who create reinforcement learning experiments. 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 6) to assist in writing an experiment program.

3 RL-Glue Environment Programs

The environment represents everything outside the agent's direct control. For example, in a tabular grid world, the environment determines the state space, the obstacles, the rewards, start states, termination conditions and the state transitions. In a robotic task, the environment would also include the robot's body, because the agent does not have complete, deterministic control over its motors. The environment is basically everything that is not the agent.

In RL-Glue, the environment is defined by a set of parameterized functions that the RL-Glue interface queries on behalf of the experiment program. These functions define what the environment does before an experiment begins, at the beginning of an episode, on every remaining step of the episode and after the experiment is completed. The following sections describe the basic requirements of an RL-Glue environment and present a complete list of all environment functions.

3.1 Essential Components Of A RL-Glue Environment

Every RL-Glue environment must implement a number of functions. The most important functions are `env_start` and `env_step`.

3.1.1 Observation and Action Encoding

In RL-Glue, Observations, Actions, and some other types are represented by structures that are any combination of:

- list of discrete numbers (`int`)
- list of continuous numbers (`double`)
- list of ASCII characters

We have found that most action an observation types can easily be captured with this structure. In a grid world, for example, the action can be an `int` list of length 1 (with valid values 0-3), corresponding to (N,S,E,W) and the observation can also be an `int` list of length 1 that maps to the agent's current state label (which is also the state of the environment, in this case). In a problem like Mountain Car, the actions are discrete (0-2) and the observation is the car's position and velocity (both real numbers). The action can be an `int` list of length 1 and the observation can be a `double` list of length two. Different implementation languages will use different languages constructors to encode observations and actions: please refer to the codec specific manual for your programming language of choice for more details.

3.1.2 Environment Start

The `env_start` function is very simple. The function takes no input and simply returns an observation. The `env_start` function is the first function called at the beginning of an episode; `env_start` chooses the initial state of the environment and returns the corresponding observation. For example, the following pseudocode 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
```

3.1.3 Environment Step

The other essential piece of a RL-Glue environment is the `env_step` function. The `env_step` function must take an action, as input, and return a observation, a reward and a termination flag. In most reinforcement learning 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 would be a valid `env_step` function:

```
1. env_step(action) -> reward, observation, flag
2.      newState = updateState(action, state)
3.      flag = isTerminal(newState)
3.      reward = calculate reward for netState
4.      set observation equal to newState
5.      state = 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 fill in two functions and you have a valid RL-Glue environment. In later sections we will discuss advanced environment functions and how these additional functions can be used to write more complex experiment programs.

3.2 Additional Components Of A RL-Glue Environment

So far we have only scratched the surface of what you can do with RL-Glue environments. Additional environment functions can be used 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_spec` language that is used, in RL-Glue, to encode basic information about environments.

3.2.1 Task Specification Language (`task_spec`)

In an effort to provide the agent writer with simple and concise information about the environment, a `task_spec` is passed from the environment, through RL-Glue, to the agent. The environment's init function (`env_init`) encodes information about the problem in an ASCII string. The string is then passed to the agent's init function (`agent_init`). This information can also be used to check if an agent and environment pair are compatible. The agent is responsible for parsing any relevant information out of the `task_spec` in the `agent_init` function.

More specifically, the `task_spec` string encodes a version number, the number of observation action dimensions, the types of observations and actions, the ranges of the observations and actions and the min and max reward values.

The `task_spec` is constantly evolving to match the state-of-art of learning algorithms and tasks being solved in reinforcement learning research; we expect that the `task_spec` will evolve much faster than the main RL-Glue protocol. To prevent this document from becoming quickly outdated, we have separated the `task_spec` documentation from the main RL-Glue documentation. Please see the online `task_spec` documentation for details about different `task_spec` versions.

3.2.2 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 at the beginning and end of 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` functions might initialize the tabular state variable to zero and allocate a *numStates X numStates* state transition array. The `env_init` function can optionally define a `task_spec` string. The `env_init` function must return a string, but it may return an empty string if the `task_spec` is not required for your experiment.

The `env_cleanup` function usually deallocates or frees anything allocated in `env_init`.

3.2.3 Environment Message

Once in a while you will find yourself wishing for the ability to add custom environment functions to RL-Glue that can be called by your experiment program. The `env_message` function allows you basically add your own functionality to the RL-Glue spec. You can send a string message to the environment and it can respond with a string. For example: you could make random starting states a parameter of your environment by using a message to toggle that property:

```
env_message(inMessage) -> outMessage
     if inMessage == "turnOffRandomStarts"
          randStarts = false
     end
     if inMessage == "turnOnRandomStarts"
          randStarts = true
     end
return ""
```

3.2.4 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 abstract data type as the observations and actions; the `state_key` must be some combinations of lists of: `int`, `double`, and `character`. The `state_key` can be as simple as an integer state label, or some high dimensional array, depending on the environment. The env_set_state takes a `state_key` as input and simply restores the environment to the state encoded in the key.

Using these functions usually takes the agent out of the normal *flow* of a learning experiment, because the agent's state transitions are no longer following the full sequence that the environment would generate. Instead, the experiment program is able to help the agent jump around through the state space.

The get and set state functions are meant to be used during a single learning experiment not across runs.

For example, in an experiment program, you might want to do something like the following:

```
1. initialize RL-Glue
2. until observation is interesting do next step of episode
3. get state key for interesting observation from environment and store in stateKey
4. for 100 steps
5.      set environment state to stateKey (the interesting observation)
6.      generate a sequence of experience
```

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

3.2.5 Environment Get and Set Random Seed

It can be useful to record and set the seed of the environment's 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 to 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 RL-Glue Agent Programs

An RL-Glue agent can be as simple as a program the returns a random number on every step or a more advanced algorithm that learns a model of the reward and transition functions of environment while maximizing reward. The agent program is a decision maker first and foremost: it must return an action when queried by RL-Glue. Many RL-Glue agents, however, learn the best action by learning from the sequence of observations, actions and rewards during an episode. Agent programs, like environments, are completely defined by a set of functions you must implement. RL-Glue calls these agent functions during an experiment, as directed by the experiment program. Whether you are writing a random agent or a learning agent you usually only need to implement a few functions. This section covers the basic requirements of an RL-Glue agent and describes a number of optional agent functions.

4.1 Essential Components Of A RL-Glue Agent

An agent program is fully compatible with RL-Glue if initializes the action type and implements three functions: agent_start, `agent_step` and `agent_end`.

4.1.1 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. Just like observations actions can be any combination of a list of `int`, `double`, and `character` values.

4.1.2 Agent Start

The `agent_start` function selects the first action at the beginning of an episode based on the first observation from 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.      lastObservation=observation
3.      for each action i
4.           if highest valued action Q(observation,i)
5.           then store i as chosenAction
6. return chosenAction
```

4.1.3 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's 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:

```
1. agent_step(reward, observation)-> action
2.      newAction = egreedy(observation)
3.      QofOld = Q(lastObservation,lastAction)
4.      QofNew = Q(observation,newAction)
5.      Q(lastObservation,lastAction) = QofOld + alpha*[reward + gamma*QofNew - QofOld]
6.      lastObservation = observation
7.      lastAction = newAction
```

```
8.      set chosenAction equal to newAction
9. return chosenAction
```

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.

4.1.4 Agent End

In an episodic 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 also performs any other end-of-episode routines, such as clearing eligibility traces. If the environment is non-episodic RL-Glue will never call `agent_end`.

Continuing with the SARSA example:

```
1. agent_end (reward)
2.      QofOld = Q(lastObservation,lastAction)
3.      Q(lastObservation,lastAction) = QofOld + alpha*[reward - QofOld]
```

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.

4.2 Additional Components Of A RL-Glue Agent

You now can construct a basic RL-Glue agent. RL-Glue agents, like environments, can be made more useful with the addition of a few optional agent functions. This section describes how to use initialization, cleanup and generic message functions.

4.2.1 Agent Initialization and Cleanup

Agent programs, like environments, often need to allocate and free various data structures. The `agent_init` and agent_cleanup functions are called at the beginning and end of a learning experiment, respectively. The `agent_init` function receives the `task_spec` string as input. The `agent_init` function usually parses the `task_spec` and stores various information encoded in the string. For example, after parsing the Task Specification, the agent_init function can then initialize the value function array to the size state space using the number of states from the Task Specification. Remember the task_spec is not required and could just be an empty string.

4.2.2 Agent Message

The agent_message function is used 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. People have created whole protocols that use agent_message to set agent learning parameters, query their value functions, and more.

Here is a quick example of how you could query the current values of some parameters of an agent:

```
agent_message(inMessage) -> outMessage
      if inMessage == "getCurrentStepSize"
           return alpha
      end
      if inMessage == "getCurrentExplorationRate"
           return epsilon
      end
 return ""
```

5 RL-Glue Experiment Programs

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 the sequence of agent-environment interactions (steps) c) extract and analyze experimental data d) end the experiment and clean up. Only the RL-Glue interface functions can be called by the experiment program. No agent or environment functions can be directly accessed by the experiment program.

5.1 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.     steps=steps+1
8. 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 designer access to all the data produced during an episode; however, it is often more convenient 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 (ie terminal flag from the `env_step` function is set to true).

The `RL_step` function allows the experiment program to record/sum/average the reward at each step, but the `RL_episode` function runs many (perhaps millions) of steps without before returning control to the experiment program. The `RL_return` and `RL_num_steps` functions allow the experiment program to retrieve the cumulative reward and the number of steps used during the episode. Specifically, `RL_return` returns the sum of rewards accumulated during the current or most recently completed episode (it is reset to zero at the start of every episode). The `RL_num_steps` returns the number of steps elapsed during the current or most recently completed episode (also reset to zero). The function reference in Section 6 provides pseudo code for each of the RL-Glue interface functions.

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.

5.2 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 accessed through calls to the RL-Glue interface from the 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
```

5.2.1 Training/Testing Phase Experiment

We can now produce more advanced experiment programs that might be used in typical 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()
```

This experiment program has two phases. During the exploration phase (lines 3-5) the agent is allowed to interact with the environment without any penalty: the experiment does not measure the reward or number of steps taken during the exploration phase. The experiment program then informs the agent that the training phase is over (line 5). The agent then (presumably) stops learning so its policy can be evaluated on the same environment for 100 episodes (lines 6-8). The evaluation phase records the agents performance by measuring and the average number of steps the agent takes during each episode. Many results in the reinforcement learning literature are collected in a similar fashion.

Feel free to combine, mix and match the various RL-Glue interface functions. You will find that, these functions allow to write powerful experiment programs that are easy to read and understand.

6 Command and Function Reference

Once your comfortable with the basics and you have written a few agents and environments you may often find yourself wondering “I need to do **X**, what function should I use?”. You may also wonder what is the “intended purpose” of a particular RL-Glue function, when you are reviewing someone else’s agent or environment code. This section provides a complete listing of all agent, environment and RL-Glue interface functions for quick reference.

6.1 Agent Functions

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

```
agent_init(task_specification)
```

This function will be called first, even before `agent_start`. The `task_spec` 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_spec` strings is found here. In `agent_init` , information about the environment is extracted from the `task_spec` and then used to set up any necessary resources (for example, initialize the value function).

```
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( 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(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: `RL_episode` cutoff before entering a terminal state) `agent_end` is NOT called.

```
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_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.

NOTE: Unlike the other functions, `agent_message` can be called at any time: including before `agent_init` and after `agent_cleanup`.

6.2 Environment Functions

Every environment must implement 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 call these functions.

```
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_spec` which adheres to the `task_spec` language. A `task_spec` stores information regarding the observation and action space, as well as whether the task is episodic or continuous.

```
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(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_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 that state. 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(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() -> random_seed_key
```

Causes the environment to 'save' the state of the random number generator such that it could be restored upon presentation of `random_seed_key`.

```
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. Usually this function would only be called with a value previously generated by `env_get_random_seed`. `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()
```

This can be used to release any allocated resources. It will be called once for every call to `env_init`.

```
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.

NOTE: Unlike the other functions, `env_message` can be called at any time: including before `env_init` and after `env_cleanup`.

6.3 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_spec` to the agent. This should be called at the beginning of every trial.

```
RL_start() --> o0, a0
    o = env_start()
    a = agent_start(o)
    nextAction = a
return o,a
```

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

```
RL_step() --> rt, ot, terminal, at
    r,o,terminal = env_step(nextAction)
    if terminal == true
         agent_end(r)
         return r, o,terminal
    else
         a = agent_step(r, o)
         nextAction = 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) --> terminal
     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)
                return 0
           else
                r, o, a = RL_step()
                list = list + [r, o, a]
                num_steps = num_steps + 1

     agent_end(r)
return 1
```

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.

7 Changes from RL-Glue 2.x

Version 3.0 of RL-Glue represents a large series of updates with the intention of bringing RL-Glue from a University of Alberta side project to an open source project generally useful to the global reinforcement learning community. As part of that process, we made some pretty big changes.

7.1 The Codec Split

The codecs (including the C/C++ codec) has been split from the main RL-Glue project. Each codec package is now more independently, and they each may offer something different to the user. However, as always, they layer on RL-Glue. The root page for all of the codecs is:
http://glue.rl-community.org/Home/codecs

As usual, you don't need to change your code depending on how it will be used. The code for an agent, environment, or experiment is identical no matter if you will run it using sockets or directly compiled together. The only difference is what library you link against.

7.1.1 RL-Glue Project

The idea of the RL-Glue project is that it will very very rarely change. We've made most of the changes on our wishlist, so RL-Glue can become a library that is standard and reliable for doing reinforcement learning experiments. The goal is that it is always there, and it always just works.

This project is written entirely in C and can be linked from C or C++ code.

When you download and install the RL-Glue project, you get three artifacts:

rl_glue executable socket server The server for running socket-based experiments.

librlglue A C library that can be linked against for creating executables where the agent, environment, and experiment program are all written in C or C++. These experiments have virtually no rl-glue calling overhead.

Agent/Environment/Experiment Headers Four header files: RL_glue.h Agent_common.h, Environment_common.h, and RL_common.h.

The way that they should be included in your agents/environments/experiments is like this:

```
<rlglue/RL_common.h>             /* Datastructures                                      */
<rlglue/RL_glue.h>               /* Experiment (RL_) functions for experiments
                                  (includes RL_common)                                  */
<rlglue/Agent_common.h>          /* Agent (agent_) functions (includes RL_common)       */
<rlglue/Environment_common.h>    /* Environment (env_) functions (includes RL_common)   */
<rlglue/Environment_common.h>    /* Environment (env_) functions (includes RL_common)   */
<rlglue/utils/C/RLStruct_util.h> /* Handy utility functions for copying/initing structs*/
```

Generally, each of agent/env/experiment should only have to include one of these files. You'll probably never include RL_common.h, but it is needed by the others.

7.1.2 RL-Glue-Extensions Project :: C/C++ Codec

The C/C++ codec gives you libraries that can be used to build stand-alone socket-based agents, environment, and experiments. The C Codec is in the rl-glue-ext project and is expected to change more frequently than the main RL-Glue project.

This project is written entirely in C and can be linked from C or C++ code.

The artifacts of the C Codec are:

librlagent Library give agents what they need to connect to the rl_glue executable server over sockets.

librlenvironment Library give environments what they need to connect to the rl_glue executable server over sockets.

librlexperiment Library give experiments what they need to connect to the rl_glue executable server over sockets.

7.2 Build Changes

We're not manually writing Makefiles anymore! We've moved both RL-Glue and the C/C++ Codec to a GNU autotools system. You can build these projects using the following standard Linux/Unix procedure now:

```
>$ ./configure
>$ make
>$ sudo make install
```

7.3 API Changes

All of the API changes have been included in all of the codecs.

7.3.1 RL_Freeze and Agent_Freeze

The freeze function were the first in a long series of "special" functions that some people wanted RL-Glue to support. The long term solution to special methods is the messaging system, `RL_agent_message` and RL_env_message. With the messaging system you can create any protocol you want between your experiment and agent or experiment and environment.

So, to reduce the clutter and kruft (cruft?) of the API, we've removed Freeze. How do you unfreeze anyways?

7.3.2 RL_Episode

Csaba Szepesvri made the request at some point that `RL_Episode` should let you know whether it ended because the time step limit expired, or because the episode ended normally. We now return the value of the terminal flag from the last `env_step` of the episode. If the flag is set to 1, then the episode terminated normally, if not, the timeout ended the episode.

7.3.3 RL_Init

It made sense to us that `RL_init` should return the task spec, in case the experiment program wants to know it, make a note of it, etc. The RL_glue specification has now been updated to handle this.

7.4 Type Changes

All of the Type changes have been included in all of the codecs.

7.4.1 Typedefs

This is a big one. We revamped all of the type names for C/C++. We made them all lower case, and added "_t" to them to identify them as types. This should reduce confusion so there is no more code like:

```
Observation observation;
```

Instead it'll be:

```
observation_t observation;
```

We think the latter is easier to read.

This is a pain to change if you have a lot of existing code, so we've made it easy to transition. There is a file you can include which will define all the old, ugly type names. This is good news because it means you can migrate to the new type names at your leisure.

```
#include <rlglue/legacy_types.h>
```

You can find all the old and new type names here:
http://code.google.com/p/rl-glue/source/browse/trunk/src/rlglue/legacy_types.h

7.4.2 charArray!

Some people have found the interface of abstract types that are arrays of `double` and `int` a little bit too constricting. We've added a third array, this time of `char`. Now people can push strings and char arrays of anything they want through observations, action, `state_key` types, etc.

The rl_abstract_type_t now looks like:

```
typedef struct
{
    unsigned int numInts;
    unsigned int numDoubles;
    unsigned int numChars;
    int* intArray;
    double* doubleArray;
    char* charArray;
} rl_abstract_type_t;
```

Keep in mind that charArray is an array of characters. It is not necessarily null terminated. We don't enforce null termination. Remember, 3 chars takes up 3 array spots, but "123" takes up 4 (' 0' at the end).

If you do the following, bad things will probably happen if the char array is not null terminated:

```
printf("My char array is %s\n",observation.charArray);
```

8 Frequently Asked Questions

8.1 Where did the task spec parsers go?

The `task_spec` language has always been a bit of a sensitive topic, and writing good parsers for the task spec has never been easy. We're currently considering slightly changing the task spec format, to make it easier to read and easier to parse. After that, we'll need (**Chance to Contribute!**) a parser for each codec. We won't need to change the main RL-Glue project, but we will need to re-release all of the codec software. So, we're waiting on deciding on changes to the language and implementing the parsers.

8.2 Can I write my agents in < insert language here >

Yes! Maybe. Writing agents/environments/experiments in different languages require there to be a codec for that language. As of writing, there are codecs for C/C++, Java, Matlab, Python, and Lisp. Check out the codecs project for more information:
http://glue.rl-community.org/Home/codecs

8.3 Does Rl-Glue support multi-agent reinforcement learning?

No. RL-Glue is designed for single agent reinforcement learning. At present we are not planning a multi-agent extension of RL-Glue. We envision that this would be a separate project with a different audience and different objectives.

8.4 Why isn't the RL-Glue interface object oriented?

RL-Glue is meant to be a low level protocol for connecting agents, environments, and experiments. These interactions can easily be described by the simple, flat, functions calls of RL-Glue. We don't feel that it is useful to overcomplicate things in that respect.

However, there is no reason that an implementation of an agent or environment shouldn't be designed using an object-oriented approach. In fact, many of the contributors to this project have their own object-oriented libraries of agents that they use with RL-Glue.

Some might argue that it makes sense to create a C/C++ or Java codec that supports an OO design directly. This would not be hard, it's just a matter of someone interested picking up the project and doing it. Personally, we've found it easy enough to write a small bridge between the existing codecs and our personal OO hierarchies.

8.5 What languages can I write my agent and environment programs in?

As of RL-Glue 3.0, C, C++, Java, Matlab and Python are supported. We soon hope to have Lisp added.

8.6 What does Observation mean? Why does the RL-Glue not pass around "states"?

Observation is a more general term, to which the concept of state, and state of say mountain car is a subset. Observation can be an array of doubles or an int or whatever. This is controlled in the common types file for each agent-environment pair.

8.7 Where is the "environmental state" stored in RL-Glue? In other systems, such as CLSquare, the old state is passed to the environment step function.

The environment in RL-Glue is responsible for keeping track of the current "state" and computing the next "state" given an action. Old state need not be passed. The state stays within the environment. The next_state method in CLSquare is basically the same as `env_step` in RL-Glue.

8.8 Can RL-Glue handle sampling the same trajectory a number of times consecutively?

This can be done in RL-Glue using a couple of optional functions. Using env_get_state one can get a key for current environmental state. Then use env_set_state to initialize the environment to a particular state given the key. Using `env_get_random_seed` the user can acquire the random seed used by the environment, and reset it using `env_set_random_seed` in the same fashion as getting and setting state described above.

8.9 How can my agent distinguish between training (e.g. exploration) and testing phases?

To freeze an agent use the `RL_agent_message` and `agent_message` functions. There is no standard RL function for freezing, unfreezing or checking. RL_agent_freeze has been depreciated.

8.10 Why is there no RL_freeze, RL_unfreeze or RL_frozen?

Answer: These three functions can easily be implemented through `RL_agent_message` and agent_message. There is no standard RL function for unfreezing or checking if the agent is frozen. To avoid the RL-Glue interface becoming bloated, we are trying to avoid adding too many redundant functions.

8.11 Some combinations of environments and agents did not seem to work since they assume a different specification (states specified as an array, single dimension, etc). For example, the agents I added do not work with problems with single dimensions, since I added for-loops which assume an array specification of the observations and actions.

Although we would all like to see it, it is not the current state of the art to have a single agent that solves many problems. Most write agents for particular tasks. However, we expect that an agent that works with multi-dimensional continuous state should be able to solve mountain car and say cart-pole. The RL-Glue fully supports this, with some setup work required in the `agent_init` function. That said, its not likely that an agent made for a multi dimensional task should work with a tabular agent. Reasonable subclasses exist and the workshop organizers have done a good job illustrating them in the benchmark announcement. Take a look at SarsaAgent.cpp. This should work with any tabular problem. A similar multi dimension continuous agent could be easily extended form TileAgent.cpp.

Document Information

`Revision Number: $Rev: 864 $`

```
Last Updated By: $Author: brian@tannerpages.com $
Last Updated   : $Date: 2008-09-30 20:50:49 -0600 (Tue, 30 Sep 2008) $
$URL: https://rl-glue.googlecode.com/svn/trunk/docs/TechnicalManual.tex $
```