GRL: a generic C++ reinforcement learning library

Wouter Caarls <mailto:wouter@caarls.org>

June 11th, 2015

1 Introduction

GRL is a C++ reinforcement learning library that aims to easily allow evaluating different algorithms through a declarative configuration interface.

2 Directory structure

```
.
|-- base                      Base library
|   |-- include               Header files
|   `-- src                   Source files
|       |-- agents            Agents (fixed, black box, td)
|       |-- discretizers      Action discretizers
|       |-- environments      Environments (pendulum, cart-pole)
|       |-- experiments       Experiments (online, batch)
|       |-- policies          Control policies (PID, Q-based)
|       |-- predictors        Value function predictors (SARSA, AC)
|       |-- projectors        State projectors (tile coding, fourier)
|       |-- representations   Representations (linear, ann)
|       |-- samplers          Action samplers (greedy, e-greedy)
|       |-- traces            Elibility traces (accumulating, replacing)
|       `-- visualizations    Visualizations (value function, policy)
|-- addons                    Optional modules
|   |-- cma                   CMA-ES black-box optimizer
|   |-- gl                    OpenGL-based visualizations
|   |-- glut                  GLUT-based visualizer
|   |-- llr                   Locally linear regression representation
|   |-- matlab                Matlab interoperability
|   |-- muscod                Muscod interoperability
|   |-- odesim                Open Dynamics Engine environment
|   |-- rbdl                  Rigid Body Dynamics Library dynamics
|   `-- ros                   ROS interoperability
```

```
|-- bin                       Python binaries (configurator)
|-- externals                 Imported external library code
|-- cfg                       Sample configurations
|-- share                     Misc files
|-- tests                     Unit tests
|-- CMakeLists.txt            CMake instructions to build everything
`-- grl.cmake                 CMake helper functions
```

3 Prerequisites

GRL requires some libraries in order to compile. Which ones exactly depends on which agents and environments you would like to build, but the full list is

- Git
- GCC (including g++)
- Boost (for shared_ptr)
- Eigen
- GLUT
- QT4 (including the OpenGL bindings)
- TinyXML
- MuParser
- ODE, the Open Dynamics Engine
- Python (including Tkinter and the yaml reader)

On Ubuntu 14.04, these may be installed with the following command:

```
wcaarls@vbox:~$ git cmake g++ libboost-dev1 libeigen3-dev1 \
libgl1-mesa-dev-lts-utopic freeglut3-dev1 libqt4-opengl-dev \
libtinyxml-dev libmuparser-dev libode-dev1 python-yaml python-tk1 \
```

4 Building

GRL may be built with or without ROS's catkin. When building with, simply merge `grl.rosinstall` with your catkin workspace

```
wcaarls@vbox:~$ mkdir indigo_ws
wcaarls@vbox:~$ cd indigo_ws
wcaarls@vbox:~/indigo_ws$ rosws init src /opt/ros/indigo
wcaarls@vbox:~/indigo_ws$ cd src
wcaarls@vbox:~/indigo_ws/src$ rosws merge /path/to/grl.rosinstall
```

```
wcaarls@vbox:~/indigo_ws/src$ rosws up
wcaarls@vbox:~/indigo_ws/src$ cd ..
wcaarls@vbox:~/indigo_ws$ catkin_make
```

Otherwise, follow the standard CMake steps of (in the `grl` directory)

```
wcaarls@vbox:~/src/mprl$ mkdir build
wcaarls@vbox:~/src/mprl$ cd build
wcaarls@vbox:~/src/mprl/build$ cmake ..
-- The C compiler identification is GNU 4.8.2
...
wcaarls@vbox:~/src/mprl/build$ make
Scanning dependencies of target yaml-cpp
...
```

5 Build environment

The whole `grl` system is built as a single package, with the exception of `mprl_msgs`. This is done to facilitate building inside and outside catkin. There is one `CMakeLists.txt` that is used in both cases. The ROS interoperability is selectively built based on whether `cmake` was invoked by `catkin_make` or not.

Modules are built by calling their respective `build.cmake` scripts, which is done by `grl_build_library`. The include directory is set automatically, as is an `SRC` variable pointing to the library's source directory.

The build system has a simplistic dependency management scheme through `grl_link_libraries`. This calls the `link.cmake` files of the libraries on which the current library depends. Typically they will add some `target_link_libraries` and add upstream dependencies. `grl_link_libraries` also automatically adds the upstream library's include directory.

6 Class structure

Most classes in grl derive from `Configurable`, a base class that standardizes configuration such that the object hierarchy may be constructed declaratively in a configuration file. Directly beneath `Configurable` are the abstract base classes defining the operation of various parts of the reinforcement learning environment, being:

`Agent` RL-GLUE[1] style agent interface, receiving observations in an episodic manner and returning actions.

`Discretizer` Provides a list of discrete points spanning a continuous space.

`Environment` RL-GLUE style environment interface, receiving actions and returning observations.

[1] `http://http://glue.rl-community.org`

`Experiment` Top-level interface, which typically calls the agent and environment in the correct manner, but may in general implement any experiment.

`Optimizer` Black-box optimization of control policies, suggesting policies and acting on their cumulative reward.

`Policy` Basic control policy that implements the state-action mapping.

`Predictor` Basic reinforcement learning interface that uses transitions to predict a value function or model.

`Projector` Projects an observation onto a feature vector, represented as a `Projection`.

`Representation` Basic supervised learning interface that uses samples to approximate a function. As such, it generally supports reading, writing and updating of any vector-to-vector mapping.

`Sampler` (Stochastically) chooses an item from a vector of (generally unnormalized) values.

`Trace` Stores a trace of projections with associated eligibilities that can be iterated over.

`Visualization` Draws on the screen to visualize some aspect of the learning process.

`Visualizer` Keeps track of visualizations and provides the interface to the graphics subsystem.

Each abstract base class is generally implemented in various concrete classes, with or without additional hierarchy. A typical example of the information flow between the various classes can be seen in Figure 1, which depicts the standard TD control setting.

6.1 Configuration

Each `Configurable` subclass must define its type and a short description using the `TYPEINFO` macro:

```
class OnlineLearningExperiment : public Experiment
{
  public:
    TYPEINFO("experiment/online_learning", "Interactive learning experiment")

  /* ... */
};
```

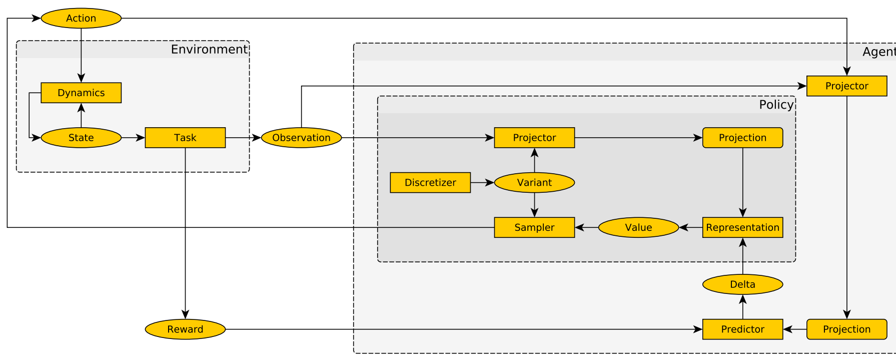

Figure 1: Information flow diagram for regular TD control. Rectangles (and dashed rectangles) are `Configurable` objects, while the others are the data passed between them.

This textual description of the type is used to facilitate user configuration by limiting the selection of parameter values, as well as enforcing the type hierarchy. In general, the textual description should follow the C++ class hierarchy, but this is not obligatory.

The basic `Configurable` interface has three important functions:

6.1.1 `request`

```
virtual void request(ConfigurationRequest *config);
```

`request` is called by the configurator to find out which parameters the object requires to be set, and which parameters it exports for other objects to use. To do this, it should extend the given `ConfigurationRequest` by pushing configuration request parameters (`CRP`s). A basic `CRP` has the following signature:

```
CRP(string name, string desc, TYPE value)
```

where TYPE is one of int, double, Vector, or string. For example:

```
config->push_back(CRP("steps", "Number of steps per learning run", steps_));
config->push_back(CRP("output", "Output base filename", output_));
```

The `value` argument is used both to determine the type of the parameter and the default value suggested by the configurator. `request` may also be called while the program is running, in which case it is expected to return the current value of all parameters.

To use other `Configurable` objects as parameters, use

```
CRP(string name, string type, string desc, Configurable *value)
```

The extra `type` field restricts which `Configurable` objects may be used to configure this parameter. Only objects whose `TYPEINFO` starts with the given `type` are eligible. For example:

```
config->push_back(CRP("policy", "policy/parameterized",

                      "Control policy prototype", policy_));
```

restricts the `"policy"` parameter to classes derived from `ParameterizedPolicy`. Note that this extra type hierarchy is related to, but not derived from the actual class hierarchy. Care must therefore be taken in the correct usage of `TYPEINFO`.

Some parameters are not requested, but rather *provided* by an object. In that case. These have the following signature:

```
CRP(string name, string type, string desc, CRP::Provided)
```

Examples of provided parameters are the number of observation dimensions (provided by `Tasks`) or the current system state (provided by some `Environments`).

6.1.2 `configure`

```
virtual void configure(Configuration *config);
```

`configure` is called after all parameters (including other `Configurable` objects) have been initialized. The parameter values may be accessed using mapping syntax (`config["parameter"]`). Note that `Configurable` objects are passed as void pointers and must still be cast to their actual class:

```
steps_ = config["steps"];
output_ = config["output"].str();
policy_ = (ParameterizedPolicy*)config["policy"].ptr();
```

Note the use of `.str()` and `.ptr()` for strings and objects, respectively. Provided parameters should be written to the configuration instead of read, like so:

```
config.set("state", state_);
```

6.1.3 `reconfigure`

```
virtual void reconfigure(const Configuration *config);
```

Some parameters may be defined as reconfigurable by appending `CRP::Online` to the respective `CRP` signature. In the case of a reconfiguration, `reconfigure` will be called with the new values of those parameters in `config`. `reconfigure` may also be used for general messaging, equivalent to RL-GLUE's `message` calls. In that case, it is often helpful to reconfigure all objects in the object hierarchy, which can be done using

```
void Configurable::walk(const Configuration &config);
```

Examples are resetting the hierarchy for a new run (`config["action"] = "reset"`) or saving the current state of all memories (`config["action"] = "save"`). In the latter case, `Configurable::path()` may be used to determine an object's location in the object hierarchy.

6.2 Roles

While using the configurator, the user often has to select previously defined objects as the value of certain parameters. If all such previously defined objects are presented as possibilities, the list would quickly grow very large. To make setting these parameters easier, a class may have various *roles* while providing the same interface. In that case, only previously defined objects with a role that starts with the requested role are valid choices.

An example is a `Representation`, which may represent a state-value function, action-value function, control policy or model. Each has a different number of inputs and outputs, and chosing the wrong representation will result in mismatches. An object requesting a `Representation` may therefore request a certain role. For example:

```
config->push_back(CRP("representation", "representation.value/action",

                      "Q-value representation", representation_));
```

requests any representation that represents action-values. A newly defined `representation` will do, of course, but from the previously defined ones only the ones with the right role are eligible.

The same strategy is used for basic types, for example:

```
config->push_back(CRP("outputs", "int.action_dims",
                      "Number of outputs", outputs_, CRP::System));
```

make sure the only suggested previously defined values for the `"outputs"` parameter are ones with the `"action_dims"` role. As an added convenience, if the parameter is defined as a *system parameter* (`CRP::System`), meaning that the choice is not free but rather defined by the structure of the configuration, and only a single value was previously defined, that value is automatically used.

The role that needs to be requested may depend on the role of the requesting object itself. In that case, the following signature for `request` should be used:

```
virtual void request(const std::string &role, ConfigurationRequest *config);
```

7 Configurator

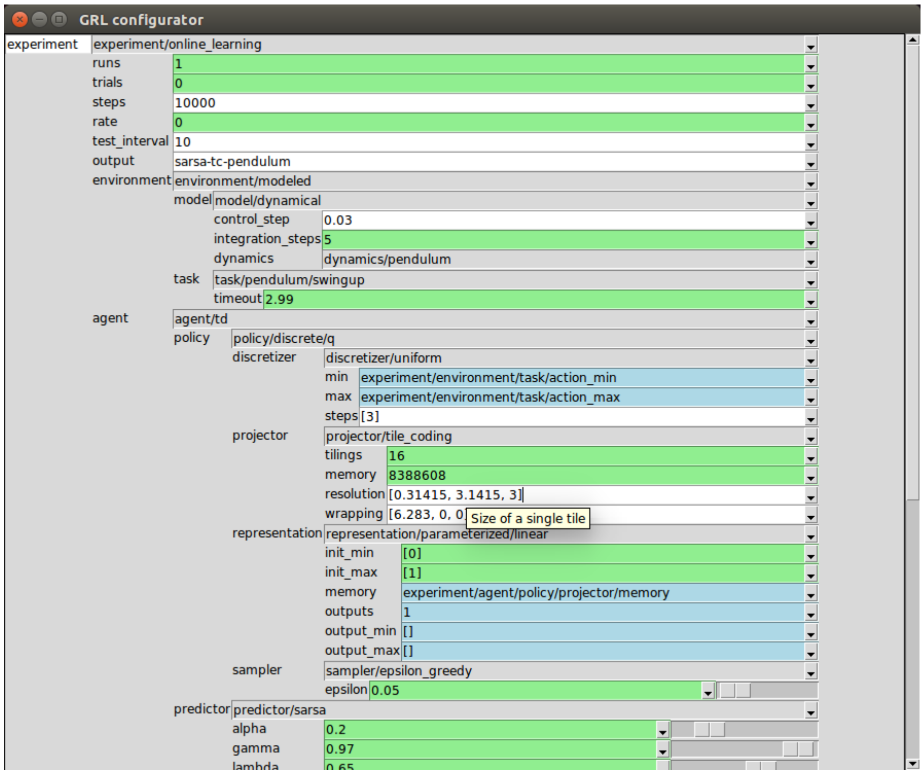

Figure 2: Python configurator user interface

8 Matlab interface

If Matlab is installed (and can be found on the path), a MEX interfaces for the agents and environments is built. If you want to use these, make sure that you're building with a compatible compiler, both by setting the `CC` and `CXX` variables in your call to `cmake` and by correctly configuring `mex`.

8.1 Environments

To initialize an environment, call

```
>> spec = grl_env('cfg/matlab/pendulum_swingup.yaml');
```

Where the argument specifies a configuration file that has a top-level 'environment' tag. `spec` gives some information about the environment, such as number of dimensions, minimum and maximum values, etc. Next, retrieve the first observation of an episode with

```
>> o = grl_env('start');
```

where `o` is the observation from the environment. All following steps should be called using

```
>> [o, r, t] = grl_env('step', a);
```

where `a` is the action suggested by the agent, `r` is the reward given by the environment and `t` signals termination of the episode. If `t` is 2, the episode ended in an absorbing state. When all episodes are done, exit cleanly with

```
>> grl_env('fini');
```

8.2 Agents

To initialize the agent, use

```
>> grl_agent('init', 'cfg/matlab/sarsa.yaml');
```

Where the argument specifies a configuration file that has a top-level 'agent' tag. Next, give the first observation of an episode with

```
>> a = grl_agent('start', o);
```

where `o` is the observation from the environment and `a` is the action suggested by the agent. All following steps should be called using

```
>> a = grl_agent('step', r, o);
```

where `r` is the reward given by the environment. To signal the end of an episode (absorbing state), use

```
>> a = grl_agent('end', r);
```

To end an episode without an absorbing state, simply start a new one. To exit cleanly after all epsiodes are finished (which also allows you to reinitialize the agent with different options), call

```
>> grl_agent('fini');
```