Quantum Behavioral Language (QBL)
Written by Walter O. Krawec

Introduction

Quantum Behavioral Language (QBL) is a simple programming language meant to encapsulate the functionality of the Quantum Decision Maker (QDM). It was tested on Windows 7, Mac OS X, and Linux. It is an extension of wok::Interp (included) and allows a programmer to use the QDM quickly in a Lisp/Scheme style syntax. See Interpreter.pdf for instructions on syntax, and basic commands available. This document lists those new commands included with QBL to access the QDM.

Building

A pre-built Windows executable is included. To build QBL on Windows, use the included MS VC++ 2010 project. For other platforms, use the included Makefile.

QBL may be extended to support new functionality by adding new commands to the interpreter. This is done in two ways: if you are running on Windows, you may use the load_module command to load a specially designed command library DLL file (see Interpreter.pdf). Otherwise, you must include the library and rebuild QBL.

To create a new command library, inherit from the wok::CommandLibrary class (WOK/interpreter.h) and overwriting the virtual functions:

int requiredVersion() : returns the minimum version of wok::Interpreter required. As of writing this document (March 15, 2013), the latest version is 1.

bool evaluate(wok::Text& text, wok::interp::Store& store, wok::interp::Object& output, wok::interp::Interpreter* interpreter, std::vector <wok::interp::CommandLibrary*> &library, bool globalMode) : will be called whenever QBL does not recognize a command. Call "text[0]" to get the command name – if this is not a command specific to this library, you must return false (so that other libraries may be searched). Otherwise, process the command by calling interpreter->evaluate(text[arg], store, objectOutput, library, globalMode). "arg" should be an integer 1, 2, … specifying the argument number of your command. objectOutput should be a wok::interp::Object type which will store the value of the command text[arg]. See Interpreter.pdf and QBL.cpp for more information. "output" should store the value of text[0] applied with arguments text[1], text[2], …

bool newUserType(const std::string& type, const std::string& name, wok::interp::Store* store) : called if "new" was called with an unknown data type. Return "false" if this type is not handled by this library (so others can be searched); otherwise create a new data type of the specified type and name, add it to the store, and return true. See QBL.h.

Usage

Typically, one will create QBL programs as a text file and run them using the "file" command (see Interpreter.pdf). A QDM requires a Delta space and an Action space. We will first describe how to construct an action space. An action space consists of a list of exclusive actions; to construct this list, first create a new qbl::qdm::ActionList type:

```
(new qbl::qdm::ActionList actions.0)
```

Our QDM considers actions as points on an integer lattice. Create the points using the ActionList constructor:

```
(set actions.0
        (qbl::qdm::ActionList
                (qbl::qdm::Action (qbl::qdm::Point 0) forward)
                (qbl::qdm::Action (qbl::qdm::Point -1) left)
                (qbl::qdm::Action (qbl::qdm::Point 1) right)
        )
)
```

This creates three actions "forward", "left", and "right". "forward" is at point "0", left at point "-1", right at point "1". Thus this action space consists of a one dimensional line. To use more dimensions, simply list the coordinates in sequence; e.g. (qbl::qdm::Point p1 p2 p3 …). One may change these points later or add new action points thus changing the behavior of the QDM. Essentially, every exclusive action set is given its own K dimensional integer lattice modulo N. K is set above by adding more coordinates to qbl::qdm::Point. N will be set later. When first run, the state of the QDM will be at (N/2, N/2, … , N/2) (that is, in the center of the integer lattice). During operation, this position will shift to other points (most likely a superposition of points). Finally, after a certain number of iterations, a measurement is

made and the particle is found on one of the points of the integer lattice. The action point defined above which is closest to the measured point is the action decided upon.

Create a new ActionList for each exclusive action space (giving each list a distinct name – e.g. actions.1 etc.).

For each ActionList we may construct an ActionSpace (these are separated so as to allow the easy changing of ActionList points while in operation). Create a new qbl::qdm::ActionSpace type for each list of exclusive actions:

(new qbl::qdm::ActionSpace A.0)

And set the action space appropriately. The most common settings are:

```
(qbl::qdm::set_action_space movementMap A.0  -1 0 1)
(qbl::qdm::set_action_space points A.0 actions.0)
(qbl::qdm::set_action_space gridSize A.0 7)
(qbl::qdm::set_action_space resetDelta A.0 1)
```

The "movementMap" setting sets the movement map of this action space. Here allow the particle to move -1, 0 and 1. For multidimensional systems, this will allow the particle to move in all combinations of these three directions for each coordinate (e.g. (0,0), (-1,0), (1,1), etc.).

"points" specifies an ActionList type describing the actual points of the actions for this exclusive action set. "gridSize" specifies how large the integer lattice should be (N from the previous example; note that each exclusive action set has its own value for K and N). Here it is of size 7. For multi-dimensional action spaces, the gridSize argument specifies the size in a single dimension (so if the points of "actions.0" were two dimensional, this would create a 7x7 grid or 49 points).

"resetDelta" specifies whether or not the delta space should be reset between runs. If yes (1 as in this example), it is sent to the center position according to "movementMap" (0 in this case). If no (0), it is left alone after a measurement (creating a quantum memory).

Now create a new qbl::QDM type and instantiate it:

```
(new qbl::QDM mainQDM)
(qbl::qdm::new mainQDM)
```

And add the action spaces to this QDM:

(qbl::qdm::add_action_space mainQDM A.0)

And finally, finalize the space (after this command you cannot add new exclusive action sets):

(qbl::qdm::finalize mainQDM 1)

The “1” argument specifies a quantum memory dimension of 1 (that is, no auxiliary memory); change this to “2” or higher to create a memory (or set resetDelta to 0).

Note: calling finalize may take time as the TOP matrix is computed at this point.

The QDM is now setup and ready either to be trained, or to load state operators and run. Typically, one would train state operators in a separate program, then load them later to be run. Training a set of state operators is done by describing the probability of choosing an action given a certain state operator (or combination of state operators).

To train, first add the sensors you wish to learn:

```
(qbl::add_sensor QDM sensor_name1)
(qbl::add_sensor QDM sensor_name2)
…
```

You may want to evolve sensors separately; only evolve them together if they interact with each other.

Next, create a new training set:

```
(new qbl::TrainingSet T)
```

Then add multiple rules to it as follows:

```
(set T (qbl::add_rule T
        (qbl::State repeat
                (qbl::SensorList sensor_name1_iter1  sensor_name2_iter1 …)
                (qbl::SensorList sensor_name1_iter2  sensor_name2_iter2 …)
                …)
        (qbl::dist_str QDM pr1 action_name11 action_name12 …
                pr2 action_name21 action_name 22 …
                …) ) )
```

The add_rule command requires a training set (in this case “T”), a “State” and a distribution. A state consists of a “mode” (repeat or single) and multiple sensors lists. Recall how the QDM works: On iteration “t”, a collection of state operators is applied to the decisional space. After this the TOP matrix is applied. After several such iterations (the number of iterations is called the reaction rate) a measurement is made of one or more action spaces and an action is decided upon. Following this, the measured action space is reset to the center position of the integer lattice. Thus, we are training a set of state operators such that, if applied in certain orders, results in an action being decided upon with a certain probability.

The training algorithm used, assumes a fixed reaction rate for all action sets. This allows us to calculate the probability of choosing any action very quickly however, if differing reaction rates are used, it loses this information. In the future we hope to improve on this algorithm.

For training, a qbl::State object (which can be a variable type created earlier using "new") is a "mode" followed by an arbitrary number (at least one though) of qbl::SensorList types (which may also be a variable type created earlier). A qbl::SensorList is simply an arbitrary list of sensor names (string names). On iteration 0, the very first list of sensors is applied (in order left to right –order matters in quantum operators) followed by TOP. On iteration 1, the next list of sensors is applied followed by the TOP and so on. If the reaction rate is reached, the fitness is calculated based on the specified distribution and the process starts over at 0 for the next rule. If the list of qbl::SensorList types is smaller than the reaction rate then, if mode = "repeat", the process starts over again with the first qbl::SensorList. If mode = "single", then no additional sensor operators are applied – just the TOP until a measurement is made.

A distribution requires a QDM object and, following this, a list of probabilities and actions. Pr1, pr2, … are doubles, action_name11, … are strings specifying actions created earlier in a qbl::qdm::ActionList. The rule above says that if, on iteration 0, the operators "sensor_name1_iter1" followed by "sensor_name_iter2", … are applied and if, on iteration 1 "sensor_name2_iter2"… are applied and so on until a measurement is made (if the reaction rate is larger than the number of sensor lists, the algorithm will start over at iter1 since "repeat" mode is used), the actions "action_name11 AND action_name12, …" should be chosen with probability p1; likewise for action_name21 AND action_name 22 … (probability p2) and so on. Note that, the sum of the probabilities need not be one. If the listed actions are in the same exclusive action set, the sum must be no larger than 1 (less than 1 is acceptable if not all actions in an exclusive action set are listed, this implies you "don't care" about the other actions). If the listed actions come from multiple action sets, the sum may be larger than 1 (for instance, an operator may decide to move left with probability 1 and chirp with probability 1, thus summing to 2).

Add multiple rules in this manner.

When finished, call the qbl::train command:

qbl::train QDM T NUMBER_OF_ITERATIONS REACTION_TIME START_POPULATION MAX_POPULATION

If using Windows, you may break out of the training at any time by pressing 'q'. Otherwise, the training stops after NUMBER_OF_ITERATIONS.

After training, save the state operators:

qbl::save QDM ("filename.dat")

Note that, in wok::Interp, it is best to enclose strings in parenthesis and quotes ("STRING").

It is usually a good idea to evolve, separately, state operators which are independent of each other. This is due to the complexity of the learning problem. See QDM.cpp/h for the training code which uses a GA to evolve a population of state operators. Here is a simple training example:

```
begin
        (new qbl::QDM QDM)
        (qbl::qdm::new QDM)
```

```
(new qbl::qdm::ActionList actions.0)
(set actions.0
        (qbl::qdm::ActionList
                (qbl::qdm::Action (qbl::qdm::Point -2) left)
                (qbl::qdm::Action (qbl::qdm::Point 0) forward)
                (qbl::qdm::Action (qbl::qdm::Point 2) right)
        )
)

(new qbl::qdm::ActionSpace A.0)

(qbl::qdm::set_action_space movementMap A.0 -1 0 1)
(qbl::qdm::set_action_space points A.0 actions.0)
(qbl::qdm::set_action_space gridSize A.0 9)

(qbl::qdm::add_action_space QDM A.0)
(qbl::qdm::finalize QDM 1)

(nop qbl::qdm::shell (get QDM))
(new qbl::TrainingSet T)

(NOP
        Add three sensors called left_ir, no_ir, and right_ir
)

(qbl::add_sensor QDM left_ir)
(qbl::add_sensor QDM no_ir)
(qbl::add_sensor QDM right_ir)

(NOP
        Simple rules to specify "choose right with probability 1 if left_ir is applied on each
        iteration until measurement"
        and so on.
)

(set T (qbl::add_rule T (qbl::State repeat (qbl::SensorList left_ir))
        (qbl::dist_str QDM 1.0 right) ) )
(set T (qbl::add_rule T (qbl::State repeat (qbl::SensorList no_ir))
        (qbl::dist_str QDM 1.0 forward) ) )
(set T (qbl::add_rule T (qbl::State repeat (qbl::SensorList right_ir))
        (qbl::dist_str QDM 1.0 left) ) )

(NOP
        Run the actual training algorithm for 10000 iterations, a start population of 100
        a max population of 210 and a reaction time of 5
)
(qbl::train QDM T 10000 5 100 210)
```

```
(NOP
        The above routines try to set the left_ir, no_ir, and right_ir sensors operators so that
        they conform to the rules listed in T.  We can use them now right away, or save them
        for later use.  We save them now to be used by another program (see qbl_run_*.txt)
)
(qbl::save (get QDM) ("samples/sensors/qbl_sensors.dat"))

(NOP
        Make sure to free the memory used by the QDM when finished.
)
(qbl::qdm::free (get QDM))
```

After training, you may run a QDM as follows. First, after the setup stage before (after the call to qbl::qdm::finalize), load your sensor operators:

qbl::load ("filename.dat")

Note: do not call qbl::add_sensor as qbl::load will do this automatically. Sensors are applied using their string names defined earlier during the training stage.
Note: you may call qbl::load multiple times for different sensor files – the sensors will be loaded cumulatively. So if you evolve sensors separately in different files, just call qbl::load for each data file.

Recall the operation of the QDM:

- Repeat:
    1. Apply appropriate sensor operators
    2. Apply TOP
    3. t = t + 1
    4. If reaction_rate(i) | t for some i :
        - Measure exclusive action set "i" and decide on an action
        - Reset measured action set to (N/2, N/2,…, N/2)
        - Reset Delta space to center (if resetDelta was 1, otherwise leave alone in a possible superposition)

Recall that each exclusive action set had its own value of K (the dimension of the integer lattice) and N (the grid size) specified by the user. The reaction_rate of any particular action set is user defined though generally N/2 (to avoid overflow though this is not required). Thus, each action set can have its own reaction rate and so decide on one action while still "thinking" about another. This is the reason for the index "i" in step 4 – this index runs through all exclusive action sets seeing if one is ready to make a decision. If so, a decision is made (based on the action point set earlier closest to the measured point) and the space is reset for the next iteration.

In QBL, the above process is done as follows:

- Apply sensor operators: qbl::qdm::apply_op QDM_NAME SENSOR_NAME

- Apply TOP: qbl::qdm::think QDM_NAME
- Decide: qbl::qdm::decide_str QDM_NAME ACTION_SET_INDEX

Here is a simple example including the setup, of a QDM with one action set (which is two dimensional) and that has five actions: left, forward, right, backward, stop (as points on the two dimensional integer lattice, these actions are resp. (-2,0), (0,2), (2,0), (0,-2), and (0,0)) (note the “robot” commands are not QBL commands):

```
begin
	(new qbl::QDM QDM)
	(qbl::qdm::new QDM)

	(new qbl::qdm::ActionList actions.0)
	(set actions.0
		(qbl::qdm::ActionList
			(qbl::qdm::Action (qbl::qdm::Point -2 0) left)
			(qbl::qdm::Action (qbl::qdm::Point 0 0) stop)
			(qbl::qdm::Action (qbl::qdm::Point 2 0) right)
			(qbl::qdm::Action (qbl::qdm::Point 0 2) forward)
			(qbl::qdm::Action (qbl::qdm::Point 0 -2) backward)
		)
	)

	(new qbl::qdm::ActionSpace A.0)

	(qbl::qdm::set_action_space movementMap A.0 -1 0 1)

	(qbl::qdm::set_action_space points A.0 actions.0)
	(qbl::qdm::set_action_space gridSize A.0 7)
	(qbl::qdm::set_action_space resetDelta A.0 0)

	(qbl::qdm::add_action_space QDM A.0)
	(qbl::qdm::finalize QDM 1)

	(qbl::load (get QDM) ("programs/sensors/qbl_follow_stop.dat"))
	(qbl::load (get QDM) ("programs/sensors/qbl_follow_forward.dat"))
	(qbl::load (get QDM) ("programs/sensors/qbl_follow_right.dat"))
	(qbl::load (get QDM) ("programs/sensors/qbl_follow_backward.dat"))
	(qbl::load (get QDM) ("programs/sensors/qbl_follow_left.dat"))

	(qbl::qdm::prepare QDM)

	(new integer time.0)
	(set time.0 0)

	(while 1
		(NOP
```

```
            Apply correct operators here based on state of agent.  For example:
            (qbl::qdm::apply_op QDM left)
        NOP)
        (qbl::qdm::think QDM)
        (set time.0 (+ time.0 1))

        (NOP
            Wait for four iterations: a reaction rate of floor(7/2 + 1)
            if time.0 is less than this, do nothing; otherwise make a decision.
        NOP)

        (if (< time.0 4)
            (NOP)
            (begin
                (set time.0 0)
                (case (qbl::qdm::decide_str QDM 0)
                    stop (begin
                        (robot::move_command robot1 STOP)
                        (NOP use "begin" block to add more than 1 command))
                    forward (robot::move_command robot1 FORWARD)
                    right (robot::move_command robot1 RIGHT)
                    backward (robot::move_command robot1 BACKWARD)
                    left (robot::move_command robot1 LEFT)
                )
            )
        )
)

(qbl::qdm::free (get QDM))
```

See the included samples for more examples of programs.

Q-Prog

QBL allows a user to easily access the functionality of our Quantum Decision Maker (QDM). The QDM in turn, relies on Q-Prog for all quantum computations. Each time you call qbl::qdm::new QDM_NAME, that "QDM_NAME" is given a new, private instance of Q-Prog. You may access this instance of Q-Prog using the command: qbl::qdm::shell QDM_NAME. See QProg.pdf for instructions on how to use this language.

Extending QBL

QBL may be easily extended with new functionality. See bin/samples/CommandLibrary for an example. Essentially, you must write a C++ class inheriting the wok::interp::CommandLibrary class (QBL itself is actually a command library for wok::Interp). Following this, you must add the library to QBL. If using Windows, it is possible to construct a DLL file to load at runtime (see an example in Simple Robot Sim

2010 – not included, but available for download at walterkrawec.org). Otherwise, you must rebuild QBL. If your new CommandLibrary class is called "NewLibrary1", you may do incorporate this library by editing the main.cpp file and adding the lines:

```
NewLibrary1 *lib = new NewLibrary1;
extras->push_back(lib);
```

After the vector "extras" is defined and before QBL.shell() or QBL.evaluate() is called. Any command supported by "NewLibrary1" will be accessible within a QBL program. Be sure to delete "lib" after shell or evaluate is called before the program itself ends.

See bin/samples/CommandLibrary for a simple example which you may use as a template for other command libraries. See QBL.cpp/QBL.h for a more complicated example of creating a CommandLibrary.

Embedding QBL in C++

The last section showed how to write stand-alone QBL programs however you may also embed QBL's functionality in a C++ program. This is shown in the QFlock demo (a separate download).

First, include "QBL.h" in your program (you will also have to link with QBL.cpp, QDM.cpp, QProg.cpp, and SimpleAlgebra.cpp). Next, create an instance of the qbl::QBL class; for our example call it "QBL":

```
qbl::QBL *QBL = new qbl::QBL;
```

You may then evaluate QBL commands as follows:

```
QBL->evaluate("COMMAND")
```

Or, if the output of the command is required:

```
wok::interp::Object output;

QBL->evaluate("COMMAND", output);
```

Then use wok::interp::ObjectHelper to get the output value (e.g. if the output is an integer, call wok::interp::ObjectHelper::getInteger(output)).

COMMAND may be any QBL, wok::Interp, or "extra" command (including the "file" command to run a separate script). If you wish to incorporate your own command library (see Interpreter.pdf and bin/samples/CommandLibrary for an example extension), you may do so as follows:

```
NewCommandLibrary1 *lib1 = new NewCommandLibrary1;
NewCommandLibrarType2 *lib2 = new NewCommandLibraryType2;
std::vector <wok::interp::CommandLibrary*> extras;
extras.push_back(lib1);
extras.push_back(lib2);
QBL->evaluate("COMMAND", &extras);
```

QBL->evaluate(“COMMAND”, output, &extras);

In this manner, you can custom-fit QBL to suite your own application’s needs (e.g. add a robot’s motor control command library).