SkyAI: Highly Modularized Reinforcement Learning Library
— Concepts, Requirements, and Implementation —

Akihiko Yamaguchi and Tsukasa Ogasawara

***Abstract*— This paper introduces a software library of reinforcement learning (RL) methods, named SkyAI. SkyAI is a highly modularized RL library for real/simulated robots to learn behaviors. Our ultimate goal is to develop an artificial intelligence (AI) program with which the robots can learn to behave as their users' wish. In this paper, we describe the concepts, the requirements, and the current implementation of SkyAI. SkyAI provides two conflicting features: high execution-speed enough for real robot systems and high flexibility to design learning systems. We also demonstrate the applications to crawling tasks of both a humanoid robot in simulation and a real spider robot.**

I. Introduction

Designing a behavior by only its objective is essential for future robots, since this ability enables the end-users to teach their wish to the robots easily. Reinforcement learning (RL) is one of such technologies, thus, RL applications in robotics are of great interest. There are a lot of researches of RL applications to robotics [1]∼[8].

However, the RL methods require a lot of learning cost in a large domain. Here, the learning cost means both the computational cost and the sampling cost (e.g. falling down). The latter one is crucial for a real robot. What is worse is that the learning cost increases exponentially with the complexity of the task and the robot (e.g. the degree of freedom). As long as this problem is not solved, it is difficult to apply RL methods to realistic tasks.

Many researchers are tackling to the learning cost issue. The following strategies to improve the RL methods are attempted so far: dimension reduction [2], model utilization [3], hierarchical structure [4], [5], imitation of the others [6], and reusing already learned knowledge [7].

These researches are focusing on specific ideas or issues; the other aspects are often simplified. However, integrating these methods is important to develop a realistic learning-robot system. Thus, we develop SkyAI aiming to integrate the RL or the other machine-learning methods so that many developers of robotics (or the other fields) can design sophisticated learning-robot systems. We believe that solving the RL issues is an important step to create robot's intelligence.

In the rest of this paper, we describe an overview of SkyAI in section II. In section III, We mention about relation to the other works. In section IV, we describe some examples about developing modules, in order to demonstrate how easily the module developers can create a new module. In section V, we introduce the already implemented modules in SkyAI, and availability. In section VI, we demonstrate the applications to crawling tasks of both a humanoid robot in simulation and a real spider robot, and we conclude the paper in section VII.

Part of this work was supported by a Grant-in-Aid for JSPS, Japan Society for the Promotion of Science, Fellows (22·9030)

A. Yamaguchi (JSPS Research Fellow) and T. Ogasawara are with the Graduate School of Information Science, Nara Institute of Science and Technology, 8916-5, Takayama, Ikoma, Nara 630-0192, JAPAN {akihiko-y, ogasawar}@is.naist.jp

II. SkyAI Overview

This section describes the principal concepts of SkyAI, the requirements basing on the concepts, the solutions, and an overview of a system with SkyAI.

A. Principal Concepts

1) High modularization: The approach of the SkyAI is *modularization* of the RL or the other machine-learning algorithms. Modular architecture enables the following features:

High extensibility: Modular architecture makes it easy to create a new module by inheriting the other module. Adding new functions, or specializing some functions are realized by a little modification. Thus, the library can be highly extensible.

High reusability of implementation: Modular architecture can separate a task (problem) specific implementation into some modules. Typical examples are reward modules and low-level robot controller. In contrast, we can make generic, i.e. highly reusable, modules.

High reusability of learned knowledge: Modular architecture can also enhance the reusability of learned knowledge, such as a learned policy by an RL algorithm, a dynamics model, and a reward model.

2) High execution-speed and high flexibility: The SkyAI must be executed in high speed, and should be highly flexible. These are very important features to apply SkyAI to real robot systems. Real robot systems require a high-speed execution. On the other hand, we need a high flexibility like script languages. Generally, these two features are conflicting.

3) Developer friendly: Highly-modularized architecture has many benefits as mentioned above, however, it sometimes makes development difficult. SkyAI pursues a developer-friendly implementation to boost the participation of many researchers and developers.

B. Requirements and Solutions

1) Writing in a compiler language: To achieve the high execution-speed, SkyAI should be written in a compiler language. We select *C++*. In general, the C++ source code is compiled to an executable code whose execution speed is

almost the highest level. This is very suitable for real robot systems.

Each module is implemented as a class of C++. Generally, communication between classes is performed by member functions[1]. We basically use *call-by-reference* for the functions, which enables high-speed communication.

2) Once compiled, reconfigurable infinitely: Once C++ source code is compiled, it is difficult to modify its behavior. Thus, SkyAI *wraps the C++ class system* and *provides a script interface* so that the modular structure can be changed by the script after compiling the source code.

To change the modular structure dynamically, the member functions for the communication between classes are needed to be connected and disconnected. Thus, each member function is encapsulated as a *port* class. Each module can have any number of ports. Ports can be connected and disconnected at any time in execution, which enables to reconfigure the modular structure.

A script language is defined to provide an interface of modular manipulations during execution. Specifically, instantiating modules, connecting ports, and setting parameters of the modules (e.g. a learning rate) can be described in the script language.

3) Using standard preprocessor and compiler: To make a module program compatible with the modular architecture, the program should follow some rules and regulations of SkyAI. Some similar software platforms, such as [9], provide their own programs to generate system-compatible source code. However, such a system often makes modification complicated, which is not developer friendly.

In contrast, SkyAI provides some macros and templates[2] to support the developers to easily write system-compatible source code. The macros and the templates can be processed by a standard preprocessor and compiler; code generators are not required. Thus, it is easy for the developers to modify source code with the system-compatibility.

C. Overview of System with SkyAI

The center of a software using SkyAI is an agent class. The agent class manages whole module instances, and has a parser of the script language. The agent class is provided as generic one, that is, available in any applications. A user instantiates the agent class and calls the parser from the C++ source code (basically, in the main function).

Fig. 1 illustrates an example modular structure around an RL module. In an on-line learning system, there are several kinds of cycles, such as episode, action, and time step of low-level controller. The SkyAI modular architecture can handle any number of cycles as shown in Fig. 1.

The SkyAI architecture enables to modularize RL algorithms as generic ones. On the other hand, the user of SkyAI should make task or robot specific modules, such as a low-level robot controller. Thus, in order to apply SkyAI, (1) the user implements some task or robot specific modules, then

[1]Public member variables are also available, but, they can be replaced by so-called *accessors*.

[2]Strictly, template functions and template classes of C++.

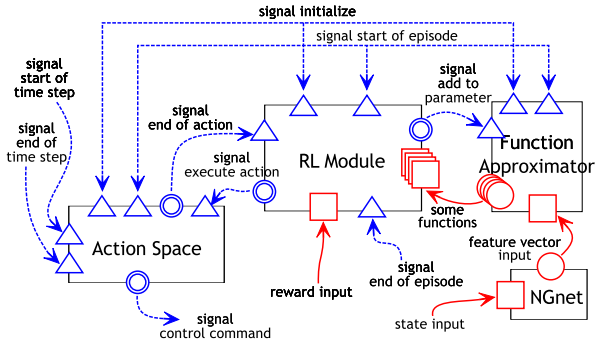

Fig. 1. Example modular structure around an RL module.

(2) the user builds them with the provided modules and the agent class, finally (3) the user writes a script for a specific task.

III. Relation to Other Works

There are similar works to develop libraries of RL or the other machine-learning methods. Compared to the libraries written in script languages, such as Python and MATLAB, SkyAI has a speed advantage. SkyAI is, therefore, considered to be more suitable for real robot systems.

Some libraries mainly written in a script language use a compiler language in crucial bottleneck parts. For example, PyBrain [10] is written in Python, but some parts are written in C/C++ which are referred from the Python code by using SWIG[3]. The reason that SkyAI does not use such a technology and defines a custom script language is (1) SkyAI provides a *composite module* architecture to compose some modules, and existing script languages are considered to be not suitable to define a composite module, and (2) SkyAI is using some features of C++ that are not supported by SWIG. However, it is considered to be possible and useful that we implement an interface of Python or the other script languages. The important fact is that SkyAI is completely written in C++ and provides an interface to manipulate the modular architecture.

The core architecture of SkyAI is a middleware. There are some middlewares for robotics, such as YARP[4], ROS[5], and OpenRTM[9]. A objective of SkyAI is to make a learning system by combining highly reusable modules. A highly reusable module generally becomes small, so SkyAI needs to prevent overhead of communication between modules. Thus, we implement each port of a SkyAI's module as a encapsulated function of call-by-reference, whose overhead may be smaller than that of the middlewares listed above.

IV. Development of Modules

In this section, we describe some examples about developing modules, in order to demonstrate how easily the module developers (including ordinary users) can create a new module.

[3]Simplified Wrapper and Interface Generator: `www.swig.org`

[4]Yet Another Robot Platform: `eris.liralab.it/yarp`

[5]Robot Operating System: `www.ros.org`

A. Creating a New Module

As described above, modules and ports are classes of C++. To make each module and port compatible with the agent class, base classes are defined for modules and ports. The developers/users need to inherit them and override some member functions. However, using the macros, these inheritances can be written in short. The following C++ code is an example of a module that simply accumulates a `double` variable.

```
class MDoubleAccumulator : public TModuleInterface
{
public:
  typedef MDoubleAccumulator  TThis;
  SKYAI_MODULE_NAMES(MDoubleAccumulator)

  MDoubleAccumulator(const std::string &instance_name)
    : TModuleInterface  (instance_name),
      sum_              (0.01),
      slot_reset        (*this),
      slot_add          (*this),
      out_sum           (*this)
    {
      add_slot_port (slot_reset);
      add_slot_port (slot_add  );
      add_out_port  (out_sum   );
    }

protected:

  double  sum_;

  MAKE_SLOT_PORT(slot_reset, void, (void), (), TThis);
  MAKE_SLOT_PORT(slot_add,
                 void, (const double &r), (r), TThis);
  MAKE_OUT_PORT(out_sum,
                 const double&, (void), (), TThis);

  virtual void slot_reset_exec (void)
    {
      sum_= 0.0;
    }
  virtual void slot_add_exec (const double &r)
    {
      sum_+= r;
    }
  virtual const double& out_sum_get (void) const
    {
      return sum_;
    }
};
SKYAI_ADD_MODULE(MRewardAccumulator)
```

Here, the super class `TModuleInterface` is the base class of every module. Its constructor takes an instance name as an argument. `SKYAI_MODULE_NAMES` macro generates basic member functions (including some overridings). `SKYAI_ADD_MODULE` macro generates code to register a module to a module manager. The module manager supports the agent class to instantiate a module from a text name. Through these steps, a module can be generated.

A port can be created in three steps. There are some sorts of ports, but they are a kind of function objects, namely, behave as a member function. In the first step, a port object is generated by a macro, such as `MAKE_SLOT_PORT`. Its arguments indicate a port name and a member function signature (return type and argument declarations). Then, we write two lines in the constructor of the module: initializer and the `add_slot_port` function. Finally, we define a member function whose signature should be the same as the one indicated to the macro. The created port can be connected to a corresponding port via the script language.

B. Parameter Handling Architecture

One of complications in C++ is to handle variables by a text. SkyAI also provides a way to handle the parameters of a module in the script language. For example, if we want to handle the following structure type in the script language,

```
struct TSomeConfigurations
{
  int                                X;
  double                             Y;
  std::vector<double>                VecData;
  std::vector<std::list<double> >    CmpData;
}
```

just define a `Register` member function and a constructor as:

```
TSomeConfigurations (TVariableMap &mmap)
  {
    Register(mmap);
  }
void Register (TVariableMap &mmap)
  {
    AddToVarMap(mmap, "X"      , X);
    AddToVarMap(mmap, "Y"      , Y);
    AddToVarMap(mmap, "VecData", VecData);
    AddToVarMap(mmap, "CmpData", CmpData);
  }
```

Then, we can assign the values in the script language as follows:

```
X = 12
Y = -2.1e+4
VecData = (1, 2.236, 3)

CmpData = {
    [] = (0.1, 0.2, 0.3)  // push back a list
    [] = (0.4, 0.5, 0.6)  // ditto
  }
```

`AddToVarMap` is a template function which can generate a variable handler for many types, even for template classes such as `std::vector`.

Thus, if a developer is familiar with C++, it is easy for the developer to make modules.

V. Implemented Modules and Availability

A. Implemented Modules

SkyAI already has about one hundred modules. Half of them are generated via the C++ template system. These modules include following important ones:

RL modules: As basic TD(λ)-learning algorithms, Peng's Q(λ)-learning [11] and Sarsa(λ) [12] are implemented. Also, fitted Q iteration [13] is implemented. These implementations are designed to be applicable with any action-value-function approximator modules that provide gradient w.r.t. their parameter vectors. As a hierarchical RL algorithm, Cohen's hierarchical RL (HRL) algorithm [14] is implemented.

Function approximator modules: Function approximators are used for the action value functions of the RL modules. A generic linear function approximator and wire-fitting [15] are implemented. The linear function approximator works with any basis-functions

module that converts a state input to a feature vector. Currently, normalized Gaussian network (NGnet) [16] is implemented.

Action space modules: These modules define action spaces for the RL modules. The most simple one is a *holder* that outputs a control input specified by an RL module during constant time. A module *discretizer* is defined to discretize the control input space; it works with the holder module. A discrete action space DCOB [1] is also modularized.

Utility modules: Some utility modules are defined, such as a data logger, a multiplier, an accumulator, a learning manager which manages episodes, and a timer which emits a signal in every specific interval.

Benchmarks: Currently, a two-dimensional maze task of a continuous state-action space and a crawling task of a simulation humanoid are implemented (same definitions as [1]). Also, as an off-the-shelf robot module, a Bioloid (made by ROBOTIS) controller is also provided. Its crawling task is defined without additional sensors, i.e. using only the sensors provided with the Bioloid.

B. Availability

The entire source code of the development version and the documentation are available from the website **`skyai.sourceforge.net`**, under the GNU General Public License. SkyAI depends on the Boost C++ libraries, and Octave (liboctave and its headers). Open Dynamics Engine (ODE) is needed to use a humanoid benchmark. SkyAI is developed on a Linux and is not built for other other platforms yet, such as Microsoft Windows and Mac OS. However, SkyAI is under active development, thus, will become available on the other platforms.

VI. Demonstrations

In this section, first, we show the results of the speed test, then demonstrate an application to a crawling task of a humanoid robot in simulation and a crawling task of an actual spider robot.

A. Speed of Modular Communication

The largest execution cost in the SkyAI system is the overhead of modular communication. Thus, we test the speed of modular communication. We implement an equivalent modular structure to the C++ code of Fig. 2, where `N1` and `N2` are constant integers. We make two modules, `MTest` as an equivalent module of `TTest`, and `MRepeater` for repeating (Fig. 3). By observing the execution time, we can know the overhead of modular communication compared to the calculation time in the `Step` function.

Table I shows the execution time in second (the mean of 100 execution). "C++ class" denotes using `TTest` (normal C++ class code), "No com" denotes using `MTest` (the SkyAI module) but no modular communication (just called the function of each port), and "Modular com" denotes using `MTest` and `MRepeater` in a SkyAI's manner. In the $\texttt{N1} = 10^8, \texttt{N2} = 10^0$ case, "Modular com" takes much more time than the others. The reason is that the cost of calculation in the `Step` function is quite small and almost the same as the overhead of the modular communication. In the other case, the execution time of the three are almost the same. From the results, we can make out that if a process of a port is very small, such as just a scalar calculation, the modular communication cost is relatively large, but for an usual process, such as update of an RL policy, the modular communication cost can be ignored. Thus, SkyAI achieves high-speed communication.

```
class TTest
{
public:
  void Init()  {a_=1;}
  void Step()
    {for(int i(0);i<N2;++i) {a_+=(a_%10==0)?2:1;}}
  void Print() {cerr<<a_<<endl;}
protected:
  int a_;
};
int main(int argc, char**argv)
{
  TTest test;
  test.Init();
  for (int i=0;i<N1;++i)  test.Step();
  test.Print();
  return 0;
}
```

Fig. 2. C++ code for testing the speed of modular communication.

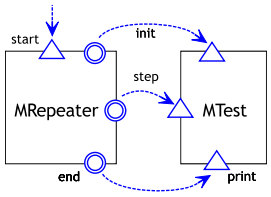

Fig. 3. Modular structure for testing speed.

TABLE I
Execution time (second)

N1	N2	C++ class	No com	Modular com
10^8	10^0	0.70	0.69	3.31
10^7	10^1	0.70	0.68	0.89
10^6	10^2	0.65	0.67	0.71
10^5	10^3	0.63	0.65	0.66

B. Demonstration of Humanoid Crawling Task in Simulation

We demonstrate an application to a crawling task of a simulation humanoid (Fig. 5). The task definition is the same as [1]. As an RL method, we apply DCOB and Q(λ)-learning from scratch.

The program is built with the basic modules and some task-specific modules. The task-specific modules are the `MHumanoidEnvironment` module which simulates the humanoid with ODE and receives the control command signal, and `MMotionLearningTask` module which calculates the reward of the crawling task by receiving the state of the humanoid. The reward is mainly given for the velocity of the center-of-mass of the body link. Then, we write a script to setup the learning; a part of it is shown in

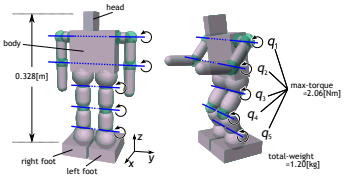

Fig. 5. Humanoid robot model whose DoF is constrained to five.

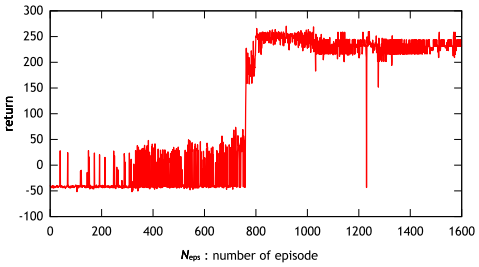

Fig. 6. Resulting learning curve of the crawling task of the humanoid robot in simulation. The curve shows the return per episode in a run.

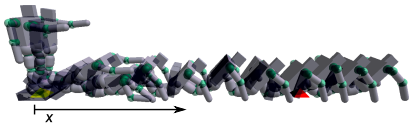

Fig. 7. Sequence of an acquired crawling motion.

```
module  MBasicLearningManager        , lmanager
module  MHumanoidEnvironment         , environment
module  MMotionLearningTask          , task
module  MUserEmittedTimer            , timer
module  MBasisFunctionsDCOBNGnet     , ngnet
module  MAVFLinearDiscAction         , avf_linear
module  MTDGenericFuncApprox_TDiscreteAction, behavior
module  MSimpleDataLogger2_TInt_TReal,logger_eps_return

module  MLinearFunctionRv            , action_converter
...

connect  behavior.signal_execute_action       ,
                dcob.slot_execute_action
connect  dcob.signal_execute_command          ,
                bftrans.slot_execute_action
connect  timer.signal_start_of_step_ud1       ,
                bftrans.slot_start_time_step
connect  bftrans.signal_execute_command       ,
                action_converter.slot_x
connect  action_converter.signal_y            ,
                environment.slot_execute_command_des_q

connect  environment.signal_system_reward     ,
                rwd_accumulator.slot_add
connect  task.signal_task_reward              ,
                rwd_accumulator.slot_add

connect  bftrans.signal_end_of_action         ,
                behavior.slot_finish_action
connect  behavior.signal_avf_add_to_parameter ,
                avf_linear.slot_add_to_parameter

connect  environment.signal_end_of_episode    ,
                behavior.slot_finish_episode
connect  task.signal_end_of_episode           ,
                behavior.slot_finish_episode
...

behavior.config={
    LearningAlgorithm = "laQLearning"
    UsingEligibilityTrace = true
    UsingReplacingTrace = true
    Alpha = 0.3
    AlphaDecreasingFactor = 0.002
    Gamma = 0.9
    Lambda = 0.9
  }
action_converter.config ={
    Factor ={
        resize(17, 5)
        [0]=  ( 0, 0, 0, 0, 0)
        [1]=  ( 1, 0, 0, 0, 0)
        [2]=  ( 0, 0, 0, 0, 0)
        [3]=  ( 0, 1, 0, 0, 0)
        [4]=  ( 1, 0, 0, 0, 0)
        [5]=  ( 0, 0, 0, 0, 0)
        [6]=  ( 0, 1, 0, 0, 0)
        [7]=  ( 0, 0, 0, 0, 0)
        [8]=  ( 0, 0, 1, 0, 0)
        [9]=  ( 0, 0, 0, 1, 0)
        [10]= ( 0, 0, 0, 0, 1)
        [11]= ( 0, 0, 0, 0, 0)
        [12]= ( 0, 0, 0, 0, 0)
        [13]= ( 0, 0, 1, 0, 0)
        [14]= ( 0, 0, 0, 1, 0)
        [15]= ( 0, 0, 0, 0, 1)
        [16]= ( 0, 0, 0, 0, 0)
      }
  }
...
```

Fig. 4. A part of the script for the crawling task of the humanoid robot in simulation.

Fig. 4. The script mainly consists of three parts: instantiating modules, connecting modules, and setting parameters. When applying an RL method to a robot system, there are several kinds of cycles; at least, episode, action, low-level control time-step. The script of SkyAI can flexibly adjust to specific systems as shown in Fig. 4. In this task, the degree-of-freedom (DoF) of the robot is constrained to five. The module `action_converter` defines this constraint by a matrix. Thus, we can change it easily.

Fig. 6 shows a resulting learning curve (return per episode), which is logged by the `logger_eps_return` module. Fig. 7 shows an acquired crawling motion.

C. Demonstration of Crawling Task of Actual Spider Robot

Next, we demonstrate an application to a crawling task of a spider robot (ROBOTIS Bioloid) shown in Fig. 8. The Bioloid can be assembled into many shapes, such as a spider, a humanoid, and a puppy. The Bioloid controller in SkyAI can handle any number of actuators as follows:

```
environment.config ={
    ActuatorIndexes = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
                       11, 12, 13, 14, 15, 16, 17, 18)
    SensingAngleIndexes = (1, 3, 7, 9, 15)
    DistanceSensorIndex = 100
  }
```

In the crawling task, the DoF of the robot is also constrained to five. Thus, only five angles are observed. We also define the `action_converter` module like Fig. 4 according to the constraint.

The experimental environment is setup as shown in Fig. 9. The robot is put in front of a wall. The robot has an infrared distance sensor to observe the distance to the wall. The horizontal velocity is calculated by differentiating the distance value, which is given to the robot as reward. The robot communicates using a serial protocol with a PC [6] into which the program with SkyAI is installed. Target joint angles are sent to the robot in every $0.1[\mathrm{s}]$.

We apply DCOB and Q(λ)-learning from scratch. Fig. 10 shows resulting learning curves (return per episode) in 4 runs. Though the robot learns from scratch, it successfully acquires a crawling motion in all runs. Fig. 11 shows snapshots of an acquired motion.

[6]Pentium M, 2[GHz], 512[MB] RAM, Debian Linux.

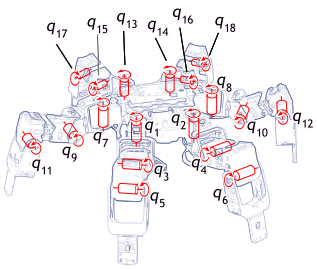

Fig. 8. King Spider (ROBOTIS Bioloid) which has 18 DoF. Its DoF is constrained to five.

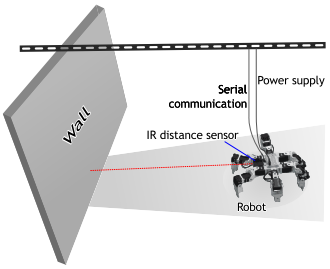

Fig. 9. Experimental environment of the crawling task of the spider robot.

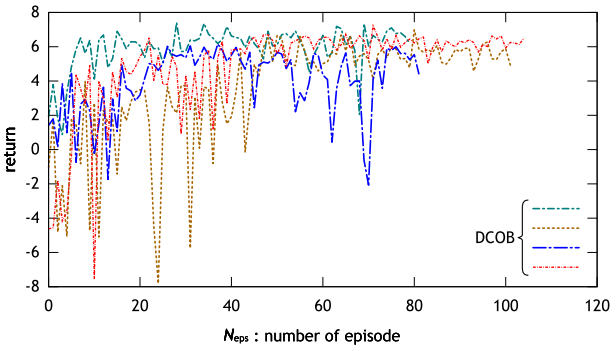

Fig. 10. Resulting learning curves of the crawling task of the spider robot. Each curve shows the return per episode in a run.

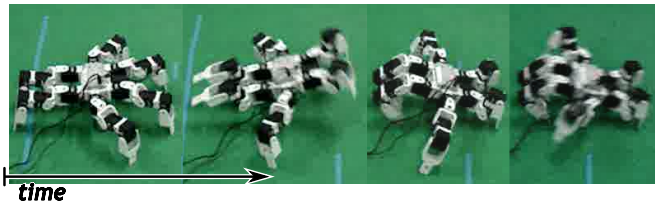

Fig. 11. Snapshots of an acquired crawling motion (4.8[s], 5.4[s], 6.0[s], 7.2[s]).

VII. Conclusion and Future Work

In this paper, we described the principal concepts, the requirements, and the solutions of SkyAI. We also demonstrated how easily a new module is created, and some applications that are also provided as benchmarks with SkyAI.

Thus, we consider that SkyAI already has an ability to handle real world tasks. However, a lot of improvements are possible. The critical ones are:

Improving the script language and GUI: A script seems complicated due to its flexibility. We need to simplify it. A graphical user interface is also useful.

Multi-threading: The current program works on only a single thread. Multi-threading is desirable to speed-up more.

Transporting to the other platforms: Current implementation may be available only on Linux. But, we are going to transport to the other platforms, such as Microsoft Windows and Mac OS.

Of course, implementing new modules of state-of-the-art RL methods and more benchmark modules is an important future work. We also have a plan to make an interface to the RL-Glue [17] which is a language-independent software package for RL experiments. We encourage researchers to joint the SkyAI project. Please visit **`skyai.sourceforge.net`**.

References

[1] A. Yamaguchi, J. Takamatsu, and T. Ogasawara, "Constructing action set from basis functions for reinforcement learning of robot control," in *the IEEE Internactional Conference in Robotics and Automation (ICRA'09)*, Kobe, Japan, 2009, pp. 2525–2532.

[2] J. Morimoto, S. Hyon, C. Atkeson, and G. Cheng, "Low-dimensional feature extraction for humanoid locomotion using kernel dimension reduction," in *the IEEE Internactional Conference in Robotics and Automation (ICRA'08)*, 2008, pp. 2711–2716.

[3] A. M. Farahmand, A. Shademan, M. Jägersand, and C. Szepesvári, "Model-based and model-free reinforcement learning for visual servoing," in *the IEEE Internactional Conference in Robotics and Automation (ICRA'09)*, Kobe, Japan, May 2009, pp. 2917–2924.

[4] J. Morimoto and K. Doya, "Acquisition of stand-up behavior by a real robot using hierarchical reinforcement learning," *Robotics and Autonomous Systems*, vol. 36, no. 1, pp. 37–51, 31 July 2001.

[5] Y. Takahashi and M. Asada, "Multi-layered learning systems for vision-based behavior acquisition of a real mobile robot," in *Proceedings of SICE Annual Conference 2003*, 2003, pp. 2937–2942.

[6] J. Kober and J. Peters, "Learning motor primitives for robotics," in *the IEEE Internactional Conference in Robotics and Automation (ICRA'09)*, 2009, pp. 2509–2515.

[7] J. Zhang and B. Rössler, "Self-valuing learning and generalization with application in visually guided grasping of complex objects," *Robotics and Autonomous Systems*, vol. 47, no. 2-3, pp. 117–127, 2004.

[8] J. Peters, S. Vijayakumar, and S. Schaal, "Reinforcement learning for humanoid robotics," in *Humanoids2003, IEEE-RAS International Conference on Humanoid Robots*, 2003.

[9] N. Ando, T. Suehiro, and T. Kotoku, "A software platform for component based rt-system development: OpenRTM-Aist," *Simulation, Modeling, and Programming for Autonomous Robots*, vol. 5325, pp. 87–98, 2008.

[10] T. Schaul, J. Bayer, D. Wierstra, Y. Sun, M. Felder, F. Sehnke, T. Rückstieß, and J. Schmidhuber, "Pybrain," *Journal of Machine Learning Research*, vol. 11, pp. 743–746, 2010.

[11] J. Peng and R. J. Williams, "Incremental multi-step Q-learning," in *International Conference on Machine Learning*, 1994, pp. 226–232.

[12] G. A. Rummery and M. Niranjan, "On-line Q-learning using connectionist systems," Cambridge University Engineering Department, Tech. Rep. CUED/F-INFENG/TR 166, 1994.

[13] D. Ernst, P. Geurts, and L. Wehenkel, "Tree-based batch mode reinforcement learning," *Journal of Machine Learning Research*, vol. 6, pp. 503–556, 2005.

[14] S. Cohen, O. Maimon, and E. Khmlenitsky, "Reinforcement learning with hierarchical decision-making," in *ISDA '06: Proceedings of the Sixth International Conference on Intelligent Systems Design and Applications*. USA: IEEE Computer Society, 2006, pp. 177–182.

[15] L. C. Baird and A. H. Klopf, "Reinforcement learning with high-dimensional, continuous actions," Wright Laboratory, Wright-Patterson Air Force Base, Tech. Rep. WL-TR-93-1147, 1993.

[16] M. Sato and S. Ishii, "On-line EM algorithm for the normalized Gaussian network," *Neural Computation*, vol. 12, no. 2, pp. 407–432, 2000.

[17] B. Tanner and A. White, "Rl-glue: Language-independent software for reinforcement-learning experiments," *Journal of Machine Learning Research*, vol. 10, pp. 2133–2136, 2009.