`CallForOptim` : simulation optimization by mean of function optimization and reinforcement learning.

September 19, 2013

Contents

1 Introduction

The package `CallForOptim` provides optimization methods and generic atomic models for calling optimization processes. Two types of optimizations are available in this package : function optimization and reinforcement learning agents.

- Given a function $f : I_d \rightarrow R$, where I_d is the domain of input of f, the function optimization problem consists in finding $x_{min} \in I_d$ such that: $x_{min} = argmin_{x \in I_d}(f(x))$. In this type of optimization, the simulation model is actually seen as a black-box function (eg. model ExBohachevsky.vpz, section 5.1).

- Reinforcement learning [4] consists in learning a policy from observations and rewards provided by the model (the model is also called environment). A policy is a function that gives the next action to perform, during simulation, in order to optimize a function that aggregates the observed rewards over the simulation. This type of optimization requires intimate relations between the agent that learns the policy and the model. Thus, the architecture is little different from the architecture for function optimization (see e.g. model ExMinesEnvironment.vpz, section 5.2).

The use-cases rely on atomic models and interfaces that are described in the next section. The available function optimization methods in this package are listed in the section 3 and RL methods are listed in section 4.

2 Global architecture

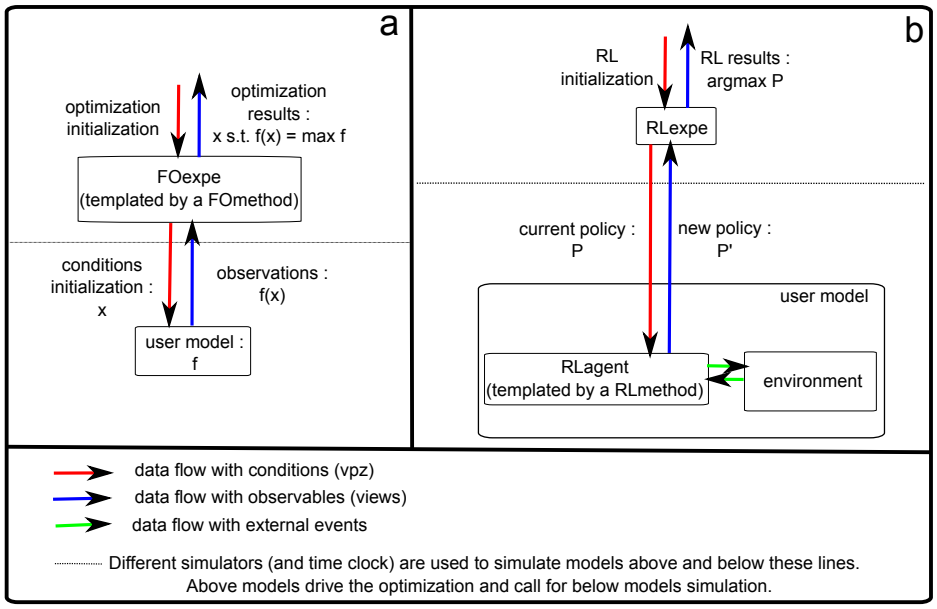

Figure 1: In use-case *a*, the user performs a simulation optimization by mean of function optimization. In use-case *b*, the user performs a simulation optimization by mean of reinforcement learning.

The first objective of this package is to provide methods for optimization. For this, atomic models (`FOexpe` and `RLexpe`) are provided that model the behaviour of optimization processes, mainly the succession of calls of the model to optimize. Configuration of these processes is given directly in the conditions

of these models, eg. the user model which is used for the optimization, the parameters for optimization, etc..

The second objective of this package is to help in developping optimization methods. Thus, the role of the developper is to implement one of the two interfaces : `FOmethod` for a function optimization method or `RLmethod` for a reinforcement learning agent.

The `FOmethod` interface contains, among others, the function :

```
virtual bool optimizeStep(unsigned int current_step);
```

The `RLmethod` interface, which is the one proposed by RL-glue [5] contains, among others, the functions :

```
virtual const vv::Value& agent_start(
                    const vv::Value& observation);
virtual const vv::Value& agent_step(double reward,
                    const vv::Value& observation);
virtual void agent_episode_ends(double reward);
virtual bool is_last_episode() const;
```

Note that one simulation of the user model represents one episode. Note also that RL methods require to be embedded in the user model, this is done thanks to the templace atomic model `RLagent`.

3 Function optimization methods

3.1 Random optimizer

The random optimizer strategy consists in generating n random vectors into the input space and to take the best input vector according to the output.

3.2 P2 algorithm

P2 [1] is a family of methods for simulation based continuous optimization for stochastic problems. They allow to optimise expected value or a quantile value of simulation[1]. These algorithms are based on hierarchical decomposition procedure. They aim at partitionning the decision space into smaller ones, and continuing the research into the potential optimal ones. The main issues are to evaluate regions of continuous decision variables, divide one region into smaller ones, and select one region among all as the one that the algorithm will investigate further.

3.3 Covariance Matrix Adaptation Evolution Srategy

CMA-ES algorithms [3] are methods for continuous optimization problems. They rely on a variance-covariance matrix over the real inputs which is used

[1] Multi objective optimization is not available yet.

to mutate candidates. This matrix globally determines the best "direction" for candidate generation into the search domain. (1+1) CMA-ES is an hill climbing method since only one new candidate is generated from the best promising candidate generated so far.

Note that this optimizer requires the use of the Shark library.

4 Reinforcement learning agents

4.1 SARSA (State-Action-Reward-State-Action)

SARSA maintains a q-matrix, ie a structure that gives the expected reward value by performing an action a in a state s. The q-matrix is updated with the following rule : $Q(s_1, a_1) \leftarrow Q(s_1, a_1) + \alpha[r + \gamma Q(s_2, a_2) - Q(s_1, a_1)]$. This update is done each time the environement is in state s_1, when the agent has performed action a_1 and that this action has lead the environment to be in state s_2 . Thus, the expected reward for pair (s_1, a_1) depends on the immediate reward r, the current expected reward $Q(s_1, a_1)$ and the expected reward of performing action a_2 from state s_2. In order to explore the state-action space, an ϵ-greedy policy is implemented. This exploring policy is used to select the next action to perform a_2. α and γ are paremeters of the method SARSA.

This reinforcement learning agent implementation is the one proposed in the book of Sutton and Barto [4].

4.2 Xitek

Xitek [2] is an implementation of the reinforcement learning method R-learning [4] adapted to simulation optimization. With Xitek, the problem of policy learning can moreover be broken down. Indeed, the policy can be split into multiple state-action spaces, each one is called "stage" and represent a particular problem.

5 Examples

Examples can be run by launching the simulation of the model `Experiments.vpz` provided into the package. For each example, appropriate conditions and dynamic have to be attached to the atomic model `Experiments`.

5.1 Using P2 for optimizing the Bohachevsky function

The bohachevsky function is defined by $[-100; 100]^2 \rightarrow R^+$, the minimal value of this function is 0 and it is taken for $x_{min} = (0, 0)$.

The model `ExBohachevsky.vpz` is a simulation atomic model, with no dynamic behaviour, that computes the value of the function for a given $x = (x_1, x_2)$. x values are given in the condition list of the model, and the function value is observed on port y.

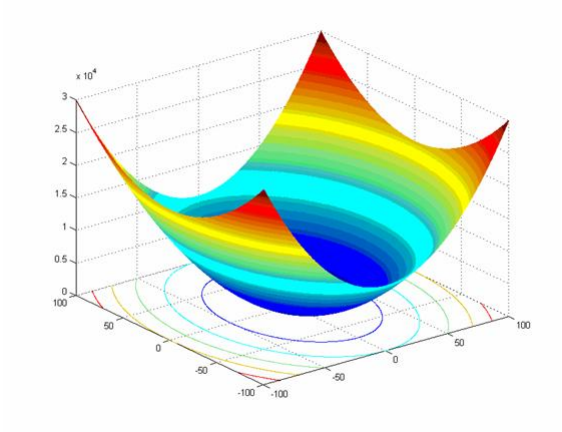

Figure 2: Bohachevsky function value in R^2.

5.2 Using SARSA for learning a policy on mines environment

This example is proposed by the developpers of the library RL-Glue [5]. The goal is to learn a policy for moving an agent into a 2D grid where cell can contain mines.

The simulation of the model `ExMinesEnvironment.vpz` stops if either the agent arrives into a goal state (the agent obtains a reward of +1000) or if the agent arrives into a state whith a mine (the agent obtains a reward of -100), see figure 3.

A policy defines the direction to take (East, West, North or South) for each cell in order to maximise the reward.

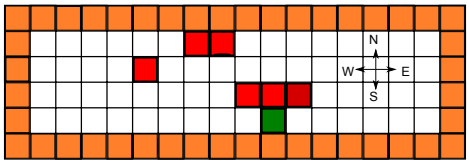

Figure 3: Mines Environment 2D grid. Red cells represent mines and the green cell represent the goal to reach.

6 Implementation notes

When implementing an optimization method (see section 2), sharing common tools is an efficient practice. This section is intended to describe tools that are

already available.

- VleSimulator is a wrapper to a simulator of vle models (available for optimization methods). Simulation of one point and simulation of a set of points can be performed. Input names of the model have to be ports of the condition "CallForOptim". Output names have to be port of the view "CallForOptim".

```
using namespace vle::value;
Map init;
Map& mod = init.addMap("model");
mod.addString("package","CallForOptim");
mod.addString("vpz","ExBohachevsky.vpz");
Set& inputs = mod.addSet("inputs");
inputs.addString("x1");
inputs.addString("x2");
Set& outputs = mod.addSet("outputs");
outputs.addString("y");
cfo::tools::VleSimulator sim(init);
Set point;
point.addDouble(3);
point.addDouble(-10);
double out = sim.simulate(point).toDouble().value();
cout << out << `` equals `` << 209.6 << endl;
```

- Accumulators are tools for incremental computation of statistics (mean, standard deviation, quantiles, etc..). See e.g. Boost accumulators library. For example :

```
using namespace cfo::tools;
Accu<MonoDim, MEAN_VAR> acc;
acc.insert(1); acc.insert(5);
acc.insert(4); acc.insert(3.6);
cout << acc.mean() << `` equals `` << 3.4 << endl;
cout << acc.moment2() << `` equals `` <<  13.74 << endl;
```

References

[1] O. Crespo. *Conception par simulation pour la conduite de culture.* PhD thesis, Université de Toulouse III, 2008.

[2] F. Garcia. Use of reinforcement learning and simulation to optimize wheat crop technical management. In *Proceedings of International Congress on Modelling and Simulation (MODSIM'99)*, pages 801–806, 1999.

[3] Christian Igel, Tobias Glasmachers, and Verena Heidrich-Meisner. Shark. *Journal of Machine Learning Research*, 9:993–996, 2008.

[4] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1998.

[5] Brian Tanner and Adam White. RL-Glue : Language-independent software for reinforcement-learning experiments. *Journal of Machine Learning Research*, 10:2133–2136, September 2009.