# EvoAgents : tutorial

François Suro

LIRMM - Laboratoire d'Informatique, Robotique et Microélectronique de Montpellier
Université de Montpellier - CNRS
161 Rue Ada, 34090 Montpellier, France
suro@lirmm.fr

November 3, 2018

## Contents

## 1 Introduction

EvoAgents is a custom framework written in Java designed to carry on experiments with agents using the MIND hierarchy as their control system. It includes the implementation MIND and its different base modules, a set of learning algorithms for neural networks using the Encog library (https://www.heatonresearch.com/encog/) for the skills internal functions, and a number of interfaces and simulation environments. EvoAgents includes 2d physics environments, both single and multi agent, supported by the JBox2d physics library (http://www.jbox2d.org/) and a viewer using java Swing library. It also includes ad single agent 3d physics environment using the JBullet physics library (http://jbullet.advel.cz/) and a 3d viewer using the JavaFx library. Finally it provides network communication through TCP or UDP to control a remote robot, external simulation environment or other programs such as video games.

EvoAgents is designed to run multi-Threaded learning algorithms in headless configuration and has been used on HPC clusters.

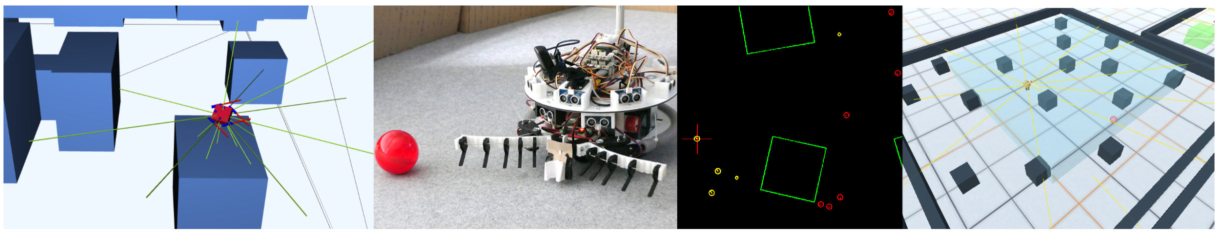

Figure 1: From left to right: 3d physics environment, our remote robot, multi agent environment, a remote environment in unity game engine

## 2 Setup the workspace

The program jar file (*EvoAgentApp_xxxxxx.jar*) must be placed next to a folder named *Minds*. The *Minds* folder will contain the different minds you want to work with, that is a folder containing the MIND hierarchy, the description of the sensor , actuators and variables of your agent and a collection of tasks you want your agent to accomplish.

the program can be run as follows :

```
java -jar EvoAgentApp_xxxxxx.jar ./Minds/MyAgentName/Tasks/MyTaskName.simbatch
```

for the console version (if you run on a cluster):

```
java -jar EvoAgentApp_xxxxxx.jar -nog ./Minds/MyAgentName/Tasks/MyTaskName.simbatch
```

## 3 A mind folder

The mind folder must be named after your agent (here we will name it MyAgentName). it contains two folders named *Skills* and *Tasks* and two files named *MyAgentName.botdesc* and *MyAgentName.vardesc*

### 3.1 MyAgentName.botdesc

Describes the sensors and actuators available to the MIND hierarchy. It must start with the keyword SENSORS, list all the sensors , one sensor per line with the format *sensorName:sensorType(:optionalParameters)*. Then the keyword ACTUATORS, list all the actuators , one actuator per line with the format *actuatorName:actuatorType*. (note: the types are not used in simulation mode, only for remote bot control, it is still a good thing to write what it's supposed to do ...)

The names of the sensors and actuators are the ones you will use when creating the MIND hierarchy. example :

```
SENSORS
S1:proxSensor
S2:proxSensor
TARG_AZIM:TargetSensor:1
TARG_ELEV:TargetSensor:2
ACTUATORS
M1:propeller
M2:propeller
```

### 3.2 MyAgentName.vardesc

Describes the variables available to the MIND hierarchy. list one variable per line with the format *variableName:variableType*.

So far there are 3 variable types :

- VariableModule : used as an input it will provide its current value. used as an output it will store the value outputted (using the MIND principle of influence)
- TickWaveSinVariableModule : used as an input it will provide the value of *sin(t)* normalized between 0 and 1, with t the tick(IE the number of times the MIND hierarchy has been asked to process inputs). used as an output it will vary the frequency of the sin wave (1.0 = t*3 , 0.0 = t/1000)
- CounterVariableModule : used as an input it will provide its current count from 0 to 10 (as 0.0,0.1,0.2,...0.9,1.0). used as an output on an ascending front above 0.8 , it will increment, on a descending front under 0.2 it will reset to 0. overflow resets to 0.

The names of the variables are the ones you will use when creating the MIND hierarchy.
example :

```
VAR_SIN:TickWaveSinVariableModule
VAR_BASE:VariableModule
#testComment
VAR_COUNT:CounterVariableModule
```

## 4 The task folder

The task folder contains the task files describing what the program is supposed to do. Task files use the extension *.simbatch*. Parameters can be listed in any order. superfluous parameters will be ignored (defining which learning method to use in a demo task which does not learn makes no sense, still better too much than not enough) here is an example task file to learn an avoid behavior in a 2d simulator (this tutorial won't touch the more exotic stuff).

```
TYPE:LEARNSIM
BOTMODEL:BotType6
BOTNAME:T6-NEAT
MASTERSKILL:Avoid
LEARNINGSKILL:Avoid
ENV:EXP_Avoid
LEARNINGMETHOD:NEAT
ROOTFOLDER:Minds/T6-NEAT
NUMGEN:50
EVALREP:3
TICKLIM:100000
```

- TYPE : the type of task to run, see the task types chapter

- BOTMODEL : which built-in bot model to use for the simulation, correspond exactly the the name of the java class that describes the agent. (see simulated bot body chapter)
- BOTNAME : the name of the agent, corresponds to the mind folder name (MyAgentName, see A Mind Folder first chapters)
- MASTERSKILL : the name of the skill that will run the MIND hierarchy (corresponds to the skill folder/ skill file name, see skills chapter )
- LEARNINGSKILL : the name of the skill that will be affected by the learning algorithm (corresponds to the skill folder/ skill file name, see skills chapter )
- ENV : name of the build-in environment to use correspond exactly the the name of the java class that describes the environment. (see simulation environment chapter)
- LEARNINGMETHOD : the learning method to use. choices are NEAT (the NEAT algorithm, multi-threaded, support resume training), GENETIC (a basic genetic algorithm, multi-threaded, does not support resume training), ANNEALING (simulated annealing, single threaded, does not support resume training)
- ROOTFOLDER: relative path to the .jar of the mind folder (Minds/MyAgentName) (see a mind folder chapter)
- NUMGEN : the number of generation/iterations to run.
- EVALREP : how many times the same agent is evaluated, final score is the sum of each evaluation. (to minimize the impact of random elements in the environment)
- TICKLIM : the tick limit for an evaluation.

## 4.1 Task types

The EvoAgentApp can perform a number of task and is left open to add more. It is recommended to stick with DEMOSIM and LEARNSIM, using 2d environments.

### 4.1.1 2d environments

The 2d environment uses the JBox2d physics library for collision, actuator forces , sensor raytracing... DEMOSIM will run your agent in real time in the environment specified and provide you with a viewer to observe his behavior. LEARNSIM will run a multi-threaded learning algorithm using the environment and its reward function. here the simulation is ran as fast as the processor is able and no viewer are provided. Progress of the learning algorithm will be displayed. LEARNSIMCONTINUOUS is meant for retraining in final context. all of the skill of the hierarchy will be successively trained in the specified environment, in an infinite loop. (see the MIND article for retraining in a broader context)

### 4.1.2 2d multi agent environments

The 2d multi agent environment uses the JBox2d physics library for collision, actuator forces , sensor raytracing. It also manages multiple teams of multiple agents. DEMO2DSIMMULTI will run multiple teams of agents in real time in the environment specified and provide you with a viewer to observe their behavior. Each team can have a different configuration of body and MIND hierarchy. LEARN2DSIMMULTI will run a multi-threaded learning algorithm using the environment and its reward function. here the simulation is ran as fast as the processor is able and no viewer are provided. Progress of the learning algorithm will be displayed.

### 4.1.3 3d environments

The 3d environment uses the JBullet physics library for collision, actuator forces , sensor raytracing... This is a work in progress. the tasks keywords are DEMO3DSIM and LEARN3DSIM.

#### 4.1.4 remote environments

It is possible to create your own simulation environment in a separate program. EvoAgentApp will use a network socket to contact your program (which should be setup as a server). This is also the technique used to control the robot of the MIND project. the tasks keywords are DEMOREMOTE for the robot, DEMOREMOTESIM for a custom sim and LEARNREMOTESIM (not tested).

#### 4.1.5 open ended development

OPENENDED : the holy grail ... not quite there yet

## 5 The Skills folder and the MIND hierarchy

This section describes how to setup the skills. You should be familiar with the general concept of the MIND hierarchy described in the article.

Each skill has a folder named after him (IE : MySkillName), this folder contains a skill description file using the *.ades* extension (MySkillName.ades) the structure is as follows :

```
*MySkillName*
input:*inputcount*
*inputType*:*inputName*
*inputType*:*inputName*
...
output:*outputcount*
*outputType*:*outputName*
*outputType*:*outputName*
*outputType*:*outputName*
...
*internalFunctionType*
*internalFunctionParameter*:*internalFunctionParameterValue*
```

### 5.1 Input types

- SE : a sensor's value
- SD : a sensor's derivative
- VA : a variable's value
- VD : a variable's derivative

### 5.2 Output types

- MO : transmit a command to an actuator
- SK : transmit influence to a subskill
- VA : transmit a value to a variable

### 5.3 Internal function types

- neural : a neural network. it's only parameter is *layers* which defines the number of hidden layers to use (note: the layer parameter is not used by the NEAT algorithm which defines the network topology on his own).
- hardCoded : a java procedure. it's only parameter is *class* which defines which java class to use.

### 5.4 examples

```
Collect
input:2
SE:SENSOBJ
SE:SENSDZ
output:3
SK:GTO+Avoid
SK:GTDZ+Avoid
SK:ClawControl
neural
layers:2
```

```
moveForward
input:0
output:2
MO:MotL
MO:MotR
hardCoded
class:MoveForward
```

```
Avoid
input:15
SE:US_1
SE:US_2
SE:US_3
SE:US_4
SE:US_5
SE:US_6
SE:US_7
SE:US_8
SE:US_9
SE:US_10
SD:US_10
SD:US_1
SD:US_2
VA:VAR_SIN
SE:DOF
output:4
SK:moveForward
SK:turnLeft
SK:turnRight
SK:moveBackwards
neural
layers:2
```

# 6 Create your own simulation elements for the 2d environment (Java programming)

This section will cover the creation of custom agents, environments and reward functions for the 2d physics environment. the *Simulated bot body* section will explain the creation of an agent using existing sensors and actuators. the *Simulation environment* section will explain the creation of the environment with pre existing world elements. *Reward functions* will explain the design of reward function used by learning algorithms. *Control functions* will explain the creation of functions used to interrupt or reset the simulation which plays an indirect role in the learning process. *Agent sensors* and *Agent actuators* will explain the creation of custom sensors and actuators *World elements* will explain the creation of static and mobile obstacles, target objects, trigger zones ...

## 6.1 Simulated bot body

Create a new class in the *evoagent2dsimulator.bot* package. The name of the class will be the name of type of the agent (the BOTMODEL parameter of a task). Your agent class must extend the *BotBody* class. You're free to override any function you wish (at your own perils). The description of the agent is done in the class constructor.

First, define the physical form of the agent, using the JBox2d library. create a new *FixtureDef* in the *sd* field (sd : shape definition). The *sd* variable will define the shape and collision layers (see JBox2d documentation for more information) create a new *BodyDef* in the *bd* field (bd : body definition). The *bd* variable will define the dynamic properties of the agent (see JBox2d documentation for more information)

example :

```
sd = new FixtureDef();
sd.shape = new CircleShape();
sd.shape.m_radius = (float)size;
//sd.friction = 1.0f;
sd.density = 2.0f;
sd.filter.categoryBits = CollisionDefines.CDBot;
sd.filter.maskBits = CollisionDefines.CDAllMask;
bd = new BodyDef();
bd.type = BodyType.DYNAMIC;
bd.angularDamping = 20.0f;
bd.linearDamping = 5.0f;
bd.allowSleep = false;
```

Then, add sensors and actuators to the agent in the *sensors* and *actuators* hashmaps. the key for the hashmap must correspond to the elements described in the agent description file (see MyAgentName.botdesc chapter) Each sensors and actuators has specific parameters, but usually start by a Vec2 representing the x,y coordinates relative to the agent body and a double representing the angle relative to the agent front orientation.

example :

```
sensors.put("US_1",new S_ProximityArcSensor(new Vec2((float)size,0.0f),0, this,
    12.0,0.8));
sensors.put("SENSOBJB",new S_ObjectDetector(new Vec2((float)size,0),0, this,null,2.0))
    ;
sensors.put("SENSVZ",new S_ZonePresence(new Vec2(0,0),0, this,null));

actuators.put("MotL",new A_Wheel(new Vec2(0.0f,-(float)size),0, this,80.0f));
actuators.put("MotR",new A_Wheel(new Vec2(0.0f,(float)size),0, this,80.0f));
actuators.put("EMAG",new A_AutoClaw(new Vec2((float)size,0f),0, this,1.5f));
```

## 6.2 Simulation environment

Create a new class in the *evoagent2dsimulator.experiments* package. The name of the class will be the name of environment (the ENV parameter of a task). Your environment class must extend the *SimulationEnvironment* class. You're free to override any function you wish (at your own perils).

In the class constructor you set the name and enable the obstacle generation by setting the *hasObstacles* field to true.

```
   public EXP_GTDZA(String botMod)
   {
      super(botMod);
      this.name = "GTDZ+Avoid";
      hasObstacles = true;
   }
```

The obstacle generation method generates a grid of static obstacle with some degree of randomness. In the case of learning methods, it will generate a set of obstacle grids, different for each repetition (successive evaluation of the same agent). All the agents will be evaluated on the same set of obstacle grids (minimizing the bias of random generation).

the parameters of this grid generation can be tweaked by setting the following fields :

```
    protected double minObstacleSize = 1.0;
    protected double maxObstacleVariability = 2.0;
    protected double obstacleSpacing = 18.0;
```

or the method generating an obstacle grid can be overridden :

```
public ArrayList<ObstaclePos> generateObstaclesParameters();

public class ObstaclePos
   public ObstaclePos(Vec2 position,float orientation, float size)
```

The definition of your environment is done by overloading the *init* method. Here follows a commented example that covers most of what you'll have to do.

```
@Override
public void init()
{
   super.init();
   // set the corrdinates of the starting position of the bot, then run the creation
       method.
   botStartPos = new Vec2(-00.5f,-0.0f);
   makeBot();

   // create wold elements and add them to the worldElements list.
   targetZone = new TriggerZone(new Vec2(-20,-20), (float)(Math.PI/4), 5);
   getWorldElements().add(targetZone);

   // create control function and reward functions and add them to their respective
       lists.
   controlFunctions.add(new CF_NextOnCollisionAndTimeout(bot,this, 20000));
   rewardFunctions.add(new RW_SensorOverThreshold(bot, 100, bot.sensors.get("SENSDZ"),
       0.5));
   rewardFunctions.add(new RW_ClosingOnTarget(bot, 0.001, targetZone));

   // some sensors need to be linked to a world element
   ((S_ZonePresence)bot.sensors.get("SENSDZ")).setTarget(targetZone);
   ((S_Radar)bot.sensors.get("RADDZ")).setTarget(targetZone);

   // 2d physics world initialisation
   makeWorld();

   // 2d physics bot initialisation
   bot.registerBotToWorld(getWorld());

   // post initialisation custom functions (here randomly placing a world element)
   randomlyPlaceTargetZone();
}
```

If you have custom world elements that needs operations when the simulation resets , overload the *reset* method

```
@Override
public void reset()
{
   // this resets the reward and control functions, also place the bot at the starting
       position
   super.reset();
   // when the simulation resets, generate a new random position for the target
   randomlyPlaceTargetZone();
}
```

Finally, you can overload *postStepOps* which is a method that is called after each simulation tick.

```
@Override
protected void postStepOps() {
   super.postStepOps();
   // when the bot has reached the target
   if(((S_ZonePresence)bot.sensors.get("SENSDZ")).getNormalizedValue() > 0.5)
   {
      // generate a new random position for the target
      randomlyPlaceTargetZone();
      // and reset the reward functions (since this is called after the simulation tick
          , the bot was already rewarded for reaching the target)
      for(RewardFunction r: rewardFunctions)
         r.reset();
   }
}
```

## 6.3 Reward functions

Reward function must extend the *RewardFunction* class and must define their *computeRewardValue* and *reset* behavior. The class name should start with *RW_*.

```
//reward the bot for getting closer to the target, punish him for getting away.

public class RW_ClosingOnTarget extends RewardFunction{
   public VirtualWorldElement target =null;
   double dist = -1;

   public RW_ClosingOnTarget(BotBody b, double rewardSt, VirtualWorldElement targetin)
       {
      super(b, rewardSt);
      target = targetin;
   }

   @Override
   public double computeRewardValue() {
      double ret = 0.0;
      double curDist = MathUtils.distance(bot.body.getPosition(),target.
          getWorldPosition());
      if(target != null && dist != -1) //only reward if there is a previous distance to
           compare
         ret = rewardStep*(dist-curDist);
      dist = curDist;
      return ret;
   }

   @Override
   public void reset()
   {
      dist = -1; //comparing with the state of a previous simulation wouldn't make
          sense
   }
}
```

## 6.4 Control functions

Control function must extend the *RewardFunction* class and must define their *performCheck* and *reset* behavior. The class name should start with *CF_*

```
//stop the simulation after a set number of ticks
public class CF_NextOnTimeout extends ControlFunction {
   int tickLimit;
   int tickCounter = 0;

   public CF_NextOnTimeout(BotBody b,SimulationEnvironment2DSingleBot w, int ticklim){
      super(b,w);
      tickLimit = ticklim;
   }

   @Override
   public boolean performCheck(){
      tickCounter++;
      if(tickCounter > tickLimit)
         return true;
      return false;
   }

   public void reset(){
      super.reset();
      tickCounter = 0;
   }
}
```

## 6.5 Agent sensors

Sensors must extend the *Sensor* class and must define their *getNormalizedValue* and *reset* behavior. The class name should start with *S_*.

*getNormalizedValue* must return a real value int the [0.0 , 1.0] range

*computeWorldPosAndAngle()* must be called before using the sensor's position in calculations (this will update the position with the translations and rotations of the bot's body).

```
//get the distance to a target, normalised bewteen [0,MaxDistance], 1.0 if over
    MaxDistance.
public class S_Distance extends Sensor{
   public VirtualWorldElement target = null;
   private double maxDist;

   public S_Distance(Vec2 lp, float la, BotBody b, VirtualWorldElement targetin, double
       maxD) {
      super(lp, la, b);
      target = targetin;
      maxDist = maxD;
   }
   public double getValue() {
      if(target != null){
         computeWorldPosAndAngle();
         Vec2 vec = new Vec2(target.getWorldPosition().x-worldPosition.x,target.
             getWorldPosition().y-worldPosition.y);
         return vec.length();
      }
      else
         return maxDist;
   }

   @Override
   public double getNormalizedValue() {
      normalizedValue = Math.min(1.0, getValue() / maxDist);
      return normalizedValue;
   }

   //no reset operation needed, the default one will be used (do nothing)
}
```

## 6.6 Agent actuators

Actuators must extend the *Sensor* Actuator and must define their *step* and *reset* behavior. The class name should start with *A_*.

Before the *step* method is called, the actuator will receive its command and store it in the *normalizedValue* member. The *normalizedValue* will always be a real number in the [0.0 , 1.0] range. The *step* method will convert the *normalizedValue* command into a concrete action of the actuator.

*computeWorldPosAndAngle()* must be called before using the actuator's position in calculations (this will update the position with the translations and rotations of the bot's body).

```
\\ converts a [0 , 1] commands into a [-maxForce , +maxForce] impulse.
\\ a 0.5 command results in a null force.
public class A_Wheel extends Actuator {
   public float maxForce = 30.0f;
   public float actuatorValue = 0.0f;

   public A_Wheel(Vec2 lp , float la, BotBody b,float mf){
      super(lp,la,b);
      maxForce = mf;
   }

   @Override
   public void step() {
      computeWorldPosAndAngle();
      // conversion from command to force value.
      actuatorValue = (((float)normalizedValue*2.0f)-1.0f)*maxForce;
      Vec2 f = new Vec2((float)Math.cos(worldAngle)*actuatorValue,(float)Math.sin(
          worldAngle)*actuatorValue);
      Vec2 p = new Vec2(worldPosition);
      bot.body.applyForce(f, p);
   }
}
```

## 6.7 World elements

At this point you must know the JBox2d library. Defining shapes, dynamic properties, and the collision mask system.

### 6.7.1 VirtualWorldElement

World elements that does not have physical presence should extend the *VirtualWorldElement* class. (ex: waypoints, markers, trigger zones...)

```
public class TriggerZone extends VirtualWorldElement {
   public String name = "dropZone";

   public TriggerZone(Vec2 worldPos, float worldAng,float s, String Label) {
      super(worldPos, worldAng,s);
      name = Label;
   }

   public boolean isPointInDZ(Vec2 point) {
      //find the world coordinates of the zone
      Vec2 points[] = new Vec2[4];
      points[0]= new Vec2(-size,size);
      points[1]= new Vec2(size,size);
      points[2]= new Vec2(size,-size);
      points[3]= new Vec2(-size,-size);
      for(int i = 0 ; i < 4 ; i++)
         points[i].set(getWorldPoint(points[i]));
      //clockwise check of all edges of a convex polygon.
      //if the point is on the left of any edge, it is outside of the polygon
      for(int i = 0 ; i < 4 ;i++)
         if(!isVectorRight(new Vec2(points[(i+1)%4].x-points[i].x,points[(i+1)%4].y-
             points[i].y),new Vec2(point.x-points[i].x,point.y-points[i].y)))
            return false;
      return true;
   }

   private boolean isVectorRight(Vec2 v1, Vec2 v2) {
      if(((v1.x) * (v2.y)) - ((v2.x) * (v1.y))>0.0)
         return false;
      else
         return true;
   }
}
```

### 6.7.2 StaticWorldElement

World elements that have physical presence but will NEVER move of be affected by forces should extend the *StaticWorldElement* class. (ex: static obstacles of any kind)

```
public class ObstacleStaticBox extends StaticWorldElement {

   public ObstacleStaticBox(Vec2 worldPos, float worldAng,float size, World w){
      super(worldPos, worldAng,size);
      sd.shape = new PolygonShape();
      ((PolygonShape)sd.shape).setAsBox(this.size, this.size);
      sd.friction = 0.0f;
      sd.density = 2.0f;
      registerToWorld(w);
   }
}
```

### 6.7.3 DynamicWorldElement

World elements that have physical presence and can be subjected to forces should extend the *Dynamic-WorldElement* class. (ex: projectiles, a rolling ball, boxes to stack...)

```
// the target object is a ball to catch
public class TargetObject extends DynamicWorldElement {
   public TargetObject(Vec2 worldPos, float worldAng, float s) {
      super(worldPos, worldAng, s);
      sd.shape = new CircleShape();
      sd.friction = 0.0f;
      sd.restitution = 1.8f;
      sd.density = 2.0f;
      sd.filter.categoryBits = CollisionDefines.CDTargetObj;
      sd.shape.m_radius = size;
      bd.angularDamping = 1.5f;
      bd.linearDamping = 0.15f;
   }
}
```