Artificial Evolution
Prof. Dario Floreano

AE Laboratory Exercises

Writing a RoboGen Scenario

In order to define a fitness function you will implement a scenario class in ECMAScript (JavaScript). To define a scenario the one thing that you are required to do is to define a function *getFitness*.

For example, this would be a valid scenario file (but it will not be very useful, since every robot would get fitness 0).

```
{
    getFitness: function() {
        return 0;
    },
}
```

Most likely you will want the fitness function to be based on what happens during the simulation. The important thing to understand is that since the *getFitness* function will be called after the simulation(s) are completed you will no longer have access to any information from the simulator at that time. So, in order to actually use information from the simulator you will need to implement additional methods. The methods you can optionally implement are

setupSimulation -- called at the very start of each simulation
afterSimulationStep -- called after every single step of a simulation
endSimulation -- called at the end of each simulation

Each of these functions can do some internal processing, and should return *true* if there are no errors, or *false* if there is a fatal error and the program should exit.

As a simple example, say we want the fitness to be the distance that the robot's core component moved (in two dimensions) during a single simulation. Our script file would then contain the following:

```
{
    setupSimulation: function() {
            // record the starting position
            this.startPos =
this.getRobot().getCoreComponent().getRootPosition();
    return true;
    },

    endSimulation: function() {
        // find the distance between the starting position and ending position
        var currentPos = this.getRobot().getCoreComponent().getRootPosition();
        var xDiff = (currentPos.x - this.startPos.x);
        var yDiff = (currentPos.y - this.startPos.y);
        this.fitness = Math.sqrt(Math.pow(xDiff,2) + Math.pow(yDiff,2));
        return true;
    },

    getFitness: function() {
        return this.fitness;
    },
}
```

More generally, you will want a scenario that can accommodate multiple simulations per fitness evaluation, i.e. to evolve robots that are robust to starting configurations and/or noise. In that case you will need to aggregate information from each simulation and then use this information to arrive at the ultimate fitness value.

The following shows how to implement our example "Racing Scenario" where the fitness in a single evaluation is the distance from the starting position to the closest part of the robot at the end of the simulation (to prevent just falling forward), and the ultimate fitness is the minimum across the evaluations (a robot is only as good as it is in its worst evaluation).

```
{
    // here we define a variable for record keeping
    distances : [],
    // function called at the beginning of each simulation
    setupSimulation: function() {
        this.startPos = this.getRobot().getCoreComponent().getRootPosition();
```

```
        return true;
    },

    // function called at the end of each simulation
    endSimulation: function() {

        // Compute robot ending position from its closest part to the start pos
        var minDistance = Number.MAX_VALUE;

        bodyParts = this.getRobot().getBodyParts();
        console.log(bodyParts.length + " body parts");
        for (var i = 0; i < bodyParts.length; i++) {
            var xDiff = (bodyParts[i].getRootPosition().x - this.startPos.x);
            var yDiff = (bodyParts[i].getRootPosition().y - this.startPos.y);
            var dist = Math.sqrt(Math.pow(xDiff,2) + Math.pow(yDiff,2));
            if (dist < minDistance) {
                minDistance = dist;
            }
        }

        this.distances.push(minDistance);
        return true;
    },

    // here we return minimum distance travelled across evaluations
    getFitness: function() {
        fitness = this.distances[0];
        for (var i=1; i<this.distances.length; i++) {
            if (this.distances[i] < fitness)
            fitness = this.distances[i];
        }
        return fitness;
    },
}
```

IN DEPTH DOCUMENTATION OF SCENARIO API

An ECMAScript Scenario will have access to the following objects and methods

Scenario		
Field/Method	**Type**	**Description**
getRobot()	Robot	Returns the robot
getEnvironment()	Environment	Returns the environment: i.e. all other objects in the Simulation besides the robot(s)

Robot		
Field/Method	**Type**	**Description**
getCoreComponent()	Model	Returns the core component of the robot represented as a Model object.
getBodyParts()	array<Model>	Returns all of the body parts that make up the robot, each is represented as a Model object.
getMotors()	array<Motor>	Returns the motors that the robot has.
getSensors()	array<Sensor>	Returns the sensors that a robot has.
vectorDistance(Vector3 vector1, Vector3 vector2)	float	Helper function to compute the distance between two 3D vectors (will also work for 2D vectors if no z component is given for either vector1 or vector2).

Model		
Field/Method	**Type**	**Description**
getRootPosition()	Vector3	Returns the position of the model's root.
getRootAttitude()	Quaternion	Returns the attitude (orientation) of the model's root.
getType()	string	Returns the type of the part in CamelCase. See http://robogen.org/docs/guidelines-for-writing-a-robot-text-file/

N.B. The component models are built up of several geometric primitives. The position / attitude of the "root" piece are made available.

Sensor		
Field/Method	**Type**	**Description**
getLabel()	string	Returns the label (name) of the sensor.
read()	float	Returns the sensor's current value.

Motor		
Field/Method	**Type**	**Description**
getId()	IoPair	Returns the id of the motor as an IoPair.
getVelocity()	float	Returns the motor's current velocity.
getPosition()	float	Returns the motor's current position.
getTorque()	float	Returns the motor's current torque.

Environment		
Field/Method	**Type**	**Description**
getLightSources()	array<LightSource>	Returns the light sources in the environment.
getAmbientLight()	float	Returns the ambient light present in the environment.
getObstacles()	array<Obstacle>	Returns the obstacles in the environment.

PositionObservable		
Field/Method	**Type**	**Description**
getPosition()	Vector3	Returns the object's position.
getAttitude()	Quaternion	Returns the object's attitude (orientation).

LightSource (extends PositionObservable)		
Field/Method	**Type**	**Description**
getIntensity()	float	Returns the light source's intensity.
setIntensity(float intensity)	void	Set the intensity of the light source.
setPosition(Vector3 position)	void	Set the position of the light source.

Obstacle (extends PositionObservable)

N.B. Currently all obstacles are BoxObstacles, but this may change in the future.

BoxObstacle (extends Obstacle)		
Field/Method	**Type**	**Description**
getSize()	Vector3	Return the box's dimensions.

IoPair		
Field/Method	**Type**	**Description**
partId	string	Id of the part that the sensor/motor is attached to.
ioId	int	ioIndex of the sensor/motor.

Vector3		
Field/Method	**Type**	**Description**
x	float	x component of vector
y	float	y component of vector
z	float	z component of vector

Quaternion		
Field/Method	**Type**	**Description**
x	float	x component of quat
y	float	y component of quat
z	float	z component of quat
w	float	w component of quat

Logging: it is possible to log to the console by using console.log()