Implementation Notes for the User of the kd-tree Viability Framework

Isabelle Alvarez[a b] Romain Reuillon[c] Ricardo De Aldama[c]

May 20, 2016

Abstract

In order to popularize the use of viability analysis we propose a framework in which the viability sets are represented and approximated with particular kd-trees. The computation of the viability kernel is seen a special active learning problem. This framework aims at simplifying the declaration of the viability problem and provides useful methods to assist further use of viability sets produced by the computation. See [1] for details. We give here some indication for the users of the framework and describe some inside algorithms.

Viability theory, kd-tree, decision support

1 Implementation and user indication

1.1 Quick start notes and options

The algorithms are implemented in Scala and are available in a free and open-source implementation [1]. In this repository 3 main folders are exposed:

- the "kd-tree" folder contains the algorithm kd-tree active learning algorithm,
- the "viability" folder contains the viability kernel and the capture basin computation algorithm,
- the "example" folder contains a set of examples.

Using this library, the user can assemble several blocks to define a viability problem. In the first place, the user should program a dynamic. The listing 1 exposes the code of the Consumption model (this model is described in section 1.2). To achieve that, the user extends the "Model" trait and implements the "dynamic" method. This method takes a state (vector) and a control (vector) and computes the resulting state when the dynamic is applied for these given state and control. Controls can be state-dependent.

[a]Irstea, UR LISC, France, isabelle.alvarez[at]irstea.fr
[b]UPMC Univ Paris 06, LIP6, Paris, France
[c]ISC-PIF, France, romain.reuillon[at]iscpif.fr
[1]https://github.com/ISCPIF/viability

Listing 1: Define a dynamic

```
trait Consumer <: Model {

  val integrationStep = 0.002
  val timeStep = 0.1

  def dynamic(state: Point, control: Point) = {
    def xDot(state: Array[Double], t: Double) = state(0) - state(1)
    def yDot(state: Array[Double], t: Double) = control(0)
    val dynamic = Dynamic(xDot, yDot)
    dynamic.integrate(state.toArray, integrationStep, timeStep)
  }

}
```

Then the user can instantiate a so called "ViabilityKernel" computation. When instantiating this class, some additional modules should be selected:

- a module to define the input for the algorithm: the user may either choose "ZoneWithPointInput" meaning that the research zone for the algorithm is provided as well as a point for which the label must be true. If the user as no knowledge of such a point it can use the "ZoneInput" component which automatically looks for a point with a true label before starting the viability kernel algorithm.
- a module to define the constraints zone K: the user may opt for the "ZoneK" module for which the constraints zone match the input zone (which is quite common) or use the "LearnK" module for which the user provide K as an oracle function. In this latter case, an algorithm learns K using a kd-tree before the first step of the viability kernel algorithm.
- a module to define a sampling strategy for the test points: the "GridSampler" samples points on a regular grid at the center of the hyper-rectangle, alternatively the user can opt for a "RandomSampler" which sample points at random in the tested hyper-rectangle.

Listing 2: Compute the viability kernel

```
val viability =
  new ViabilityKernel
    with ZoneInput
    with ZoneK
    with GridSampler
    with Consumer {
    def controls = (-0.5 to 0.5 by 0.1).map(Control(_))
    def zone = Seq((0.0, 2.0), (0.0, 3.0))
    def depth = 16
    def dimension = 2
  }

implicit lazy val rng = new Random(42)

val kernel = viability().lastWithTrace{ (tree, step) => println(step)
     }
```

```
println(kernel.volume)
```

The library as been designed to be flexible. For instance the oracle evaluation can be executed in parallel by adding a single line as it is shown in listing 3.

Listing 3: Compute the viability kernel

```
val viability =
  new ViabilityKernel
    with ...
    with ParallelEvaluator {
  ...
}
```

1.2 Consumption Model

This example is taken from [2].

The consumption model is proposed by [2] to describe the consumption of raw material governed by price. The state variable $x(t)$ represents the consumption of the raw material, and the state variable $y(t)$ its price. The rate of change at each time step of the price is controlled and bounded par parameter c with $u(t) \in [-c, c]$. The constraint set is $K = [0, b] \times [0, d]$. The dynamics are described by the following equations:

$$\left\{ \begin{array}{lcl} x(t+dt) & = & x(t) + (x(t) - y(t))dt \\ y(t+dt) & = & y(t) + u(t)dt \; with \; |u(t)| \leq c \end{array} \right. \tag{1}$$

This viability problem can be resolved analytically (see [2] for details). When dt tends toward 0, the theoretical viability kernel is defined by:

$$((x, y) \in [0, b] \times [0, d]) \in Viab(K) \Leftrightarrow \begin{cases} x \geq y - c + c.e^{(-y/c)} \\ \text{and when } y \leq b \text{ then } x \leq y + c - c.e^{\frac{y-b}{c}} \end{cases} \tag{2}$$

The corresponding dynamics in dimension 1 is $x' = x - c$, it is Lipschitz continuous with constant $\mu = 1$.

2 Algorithms

The viability algorithm used in this framework is based on the classification method described in [3]. Sets are represented by kd-tree as in [4]. The framework is described in [1] submitted to KBS, the preprint can be find from the author.

2.1 Learning a Set with kd-trees: The kd-LA algorithm

We consider that a function $f : R \subset \mathbb{R}^p \mapsto \{0, 1\}$ is available, where f is the indicator $\mathbb{1}_S$ of a compact simply connected set S subset of the hyperrectangle R. This function is called the *oracle*. Calls to the *oracle* can be very costly depending on the set S, but they can be easily parallelized.

The main algorithms: `LearnBoundary`, `ComputeVK`,`BuildStepVK`, and operation algorithms (dilation, erosion) can be found in the paper. Here is some additional details.

Algorithm 1. ***leavesToRefine***$(node)$

used in main Algorithm LearnBoundary
each leaf of the tree is labelled
$leaves \leftarrow \{$
`if` *node* ***is a leaf*** `then`
  `if` $f(node) = 1$ AND *node* ***is a border***
    `then return` $\{node\}$
  `else return`$(\emptyset)$
`else` *{* $(node_1, node_2) \leftarrow node.$***children***
  result $\leftarrow \emptyset$
  result $\leftarrow$ ***result*** $\cup$ `leavesToRefine`$(node_1)$
  result $\leftarrow$ ***result*** $\cup$ `leavesToRefine`$(node_2)$
  result $\leftarrow$ ***result*** $\cup$ `pairsBetweenNodes`$(node_1, node_2)$
  `return` ***result*** *}*
$\}$
return distinct non-atomic elements of $leaves$

Algorithm 2. ***pairsBetweenNodes***$(node_1, node_2)$

$node_1$ *and* $node_2$ *are necessarily adjacent*
result $\leftarrow \{$
`if` ***both*** $node_1$ ***and*** $node_2$ ***are leaves*** `then`
  `if` $f(node_1) \neq f(node_2)$ `then`
  result $\leftarrow \{node_1, node_2\}$
`else if` ***neither*** $node_1$ ***nor*** $node_2$ ***are leaves*** `then`
  `foreach` $node_i$ ***child of*** $node_1$ `do` *{*
  `foreach` $node_j$ ***child of*** $node_2$ `do` *{*
    `if` $node_i$ ***and*** $node_j$ ***are adjacent*** `then`
    result $\leftarrow$ ***result*** $\cup$ `pairsBetweenNodes`$(node_i, node_j)$ *}}*
`else do` *{*
  $leaf \leftarrow$ ***the leaf*** *(either* $node_1$ *or* $node_2$*)*
  $node \leftarrow$ ***the other node***
  leaves $\leftarrow$ ***all leaves*** $leaf_i$ ***in*** $node$ ***adjacent to*** $leaf$ ***whith*** $leaf_i.label \neq leaf.label$
  result $\leftarrow$ ***result*** $\cup \{leaf\} \cup\ leaves$
`return` ***result***

Figure 1 shows an example of the result that can be obtained when using the learning algorithm by itself. When the spatial discretization step tends toward 0 (i.e. when the depth of the kd-tree tends towards infinity) then the learned set converges towards the true set (when some regularity and connectedness properties), see [4] for more details.

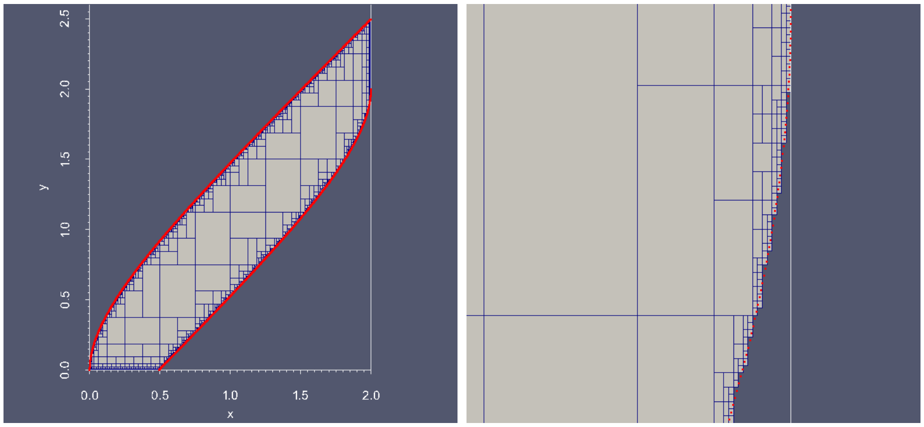

Figure 1: Approximation of the set defined by equations 2 in section 1.2 with an accuracy of 1024 points / axis (depth of 20). The red points show the boundary of the set on the 1024 points regular grid. In grey the learned kd-tree, with leaf boundary in blue. On the right a detail of the boundary of the set.

3 References

References

[1] Isabelle Alvarez, Romain Reuillon, and Ricardo de Aldama. A kd-tree framework for viability-based decision, to be submitted.

[2] J.-P. Aubin. *Viability theory*. Birkhäuser, Basel, 1991.

[3] G. Deffuant, L. Chapel, and S. Martin. Approximating viability kernels with support vector machines. *IEEE T. Automat. Contr.*, 52(5):933–937, 2007.

[4] Jean-Baptiste Rouquier, Isabelle Alvarez, Romain Reuillon, and Pierre-Henri Wuillemin. A kd-tree algorithm to discover the boundary of a black box hypervolume. *Annals of Mathematics and Artificial Intelligence*, pages 1–16, 2015.