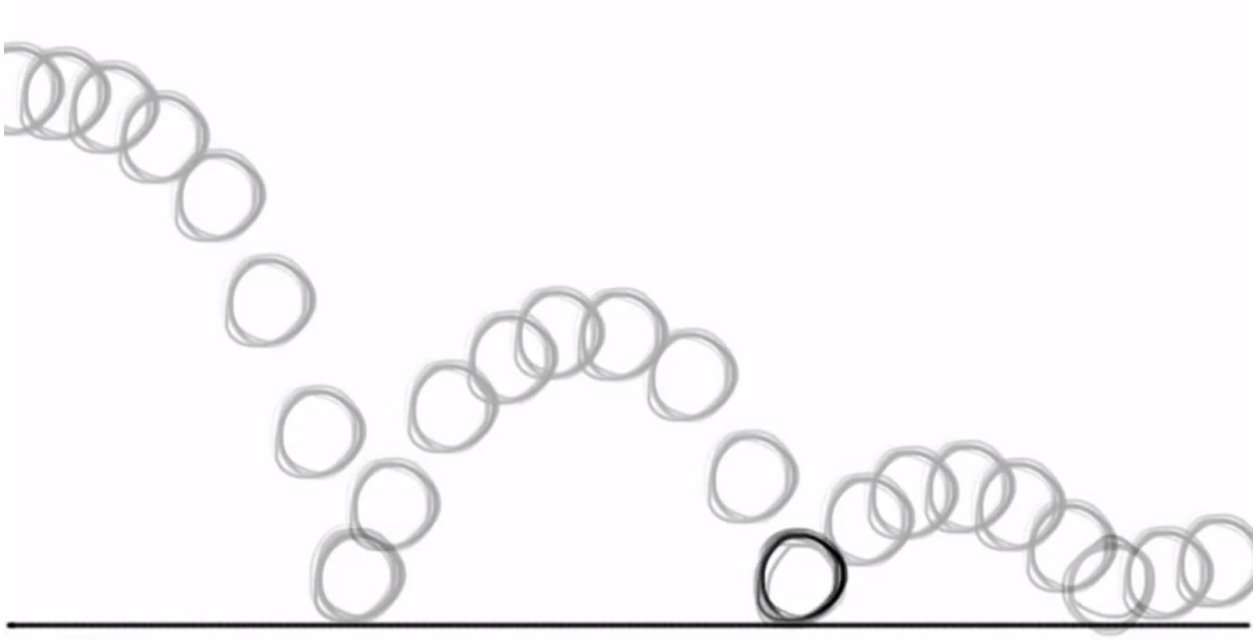


Rapport de TP de COO

Développement d'un simulateur
à événements discret de systèmes hybrides.

Claire Mevolhon
Michael VO
2018/2019

Introduction

Le but principal de ces TPs était d'implémenter un simulateur à événement discret. Pour cela notre travail s'est départagé en plusieurs étapes. La première consistait à implémenter un ordonnanceur de composants par intervalle de temps. Puis nous avons modifié les composants pour prendre en compte des valeurs continues, toujours par intervalle de temps (intégrateur qss). Par la suite nous avons implémenté un intégrateur d'Euler. Enfin nous avons utilisé la base de cette intégrateur pour simuler le tombé d'une balle au sol en fonction du temps.

Nous avons implémenté deux versions de ce problème à la fois en Java et en Python : vous trouverez ci-joint la version Java. Vous pouvez également retrouver cette version sur Github. (Sur cette dernière, la version sera certainement améliorée par la suite, tel qu'avec une interface graphique en Swing.)

Ordonnanceur de composants

Nous avons implémenté ici un ordonnanceur qui ordonnance différents types de composants. Pour cela nous avons créé une classe abstraite ***AtomicComponent***, qui nous servira tout le long de ce TP. Cette classe a ainsi été placé dans un package particulier *Business* avec ses composants liés (ses messages entrée/sortie).

On considère un composant comme une machine à états. Il possède un nom, puis une liste d'états avec un temps d'exécution défini pour chacun d'eux. Ces états sont définis à l'initiation de la classe dans une hashmap **requiredTime** directement lié au temps nécessaire de cet état. Le temps est mis à jour au cours du temps dans la variable **elapsedTime**. L'état courant et l'état suivant sont eux aussi mis à jours au cours du temps dans les attributs **current_state** et **next_state**.

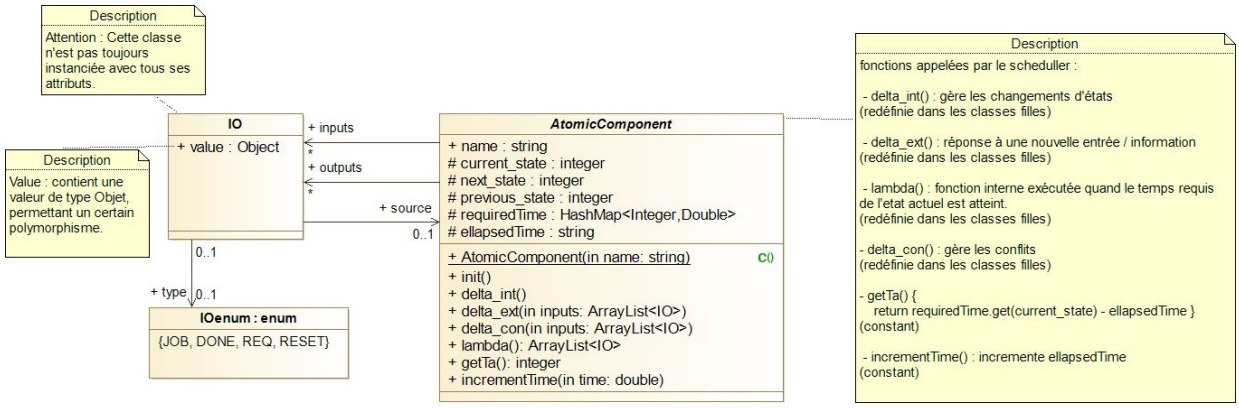
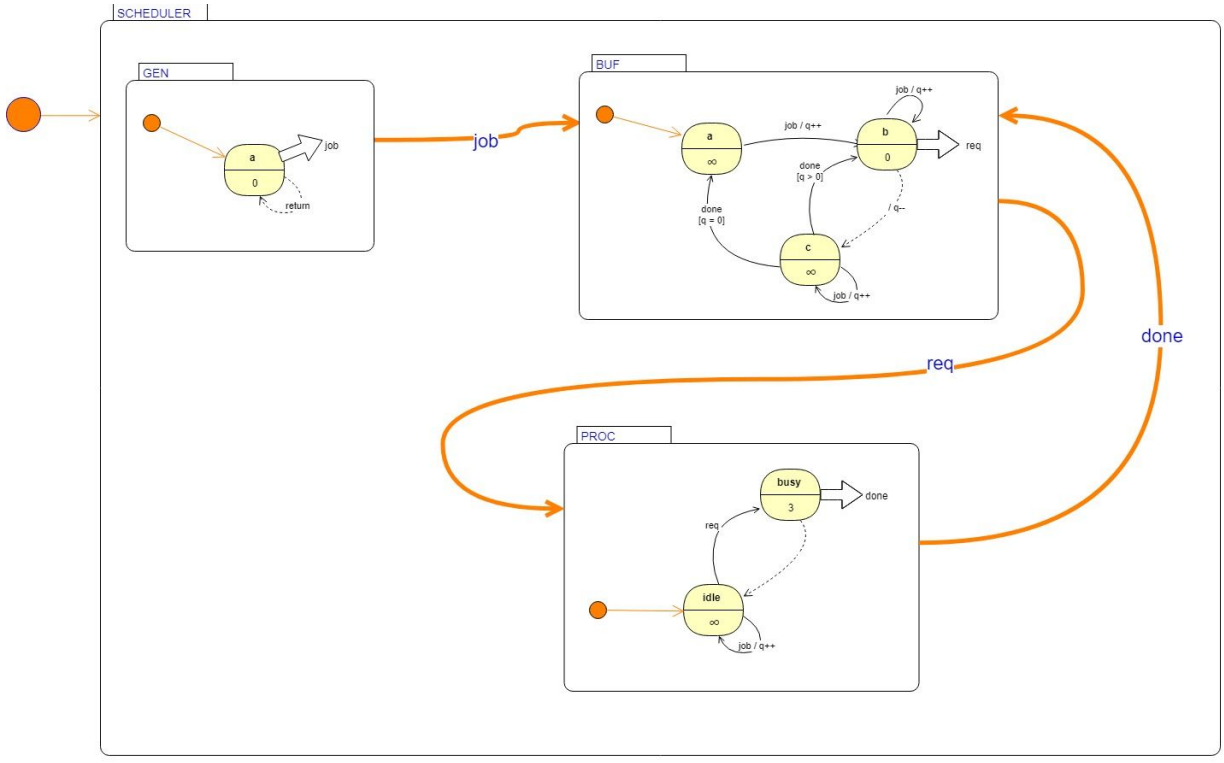


Diagramme de classe de AtomicComponent.

En ce qui concerne uniquement cette ordonnanceur précis, et ses composants, nous avons placé les classes dans un package nommé *discret*.

Les composants

Nous avons implémenté dans cette version, trois composants comme ci-dessous :



L’ordonnanceur

L’ordonnanceur possède une liste de tous ses composants. C’est lui qui gère lequel des composants va s’exécuter avec quel temps et pour quel événement.

Notre ordonnanceur est discret. Il repère quel est le prochain événement qui doit avoir lieu et à quel temps, pour incrémenter le temps directement jusqu’au prochain événement.

Son fonctionnement se fait comme suit : (manque de temps pour faire un diagramme de séquence)

```
imminentComponents.clear();
tempsMin = components.get(0).getTa();
for (AtomicComponent c : components) {
    tmp = c.getTa();
    if (tmp < tempsMin) {
        imminentComponents.clear();
        tempsMin = tmp;
    }
    if (tmp.equals(tempsMin))
        imminentComponents.add(c);
}
System.out.println("Le prochain evenement se produira dans t + " + tempsMin);

// j'execute lambda et recupere les composants impactés
temps += tempsMin;

messageList.clear();
for (AtomicComponent c : imminentComponents) {
    System.out.println(c.toString() + " execute Lambda.");
    messageList.addAll(c.lambda());
}

impactedComponents = impactedComponents(messageList);
System.out.println("Les messages emis sont : " + messageList.toString());
System.out.println("Lesquels impactent : " + impactedComponents.toString());

for (AtomicComponent c : components) {
    if (imminentComponents.contains(c)) {
        if (impactedComponents.contains(c))
            c.delta_con(messageList);
        else
            c.delta_int();
    } else { // non imminent
        if (impactedComponents.contains(c))
            c.delta_ext(messageList);
        else
            c.IncrementTime(tempsMin);
    }
}
```

Résultats

On peut suivre la trace de l'exécution à la console, ou bien on peut afficher le "q" du buffer pour se faire une idée de ce qui se passe. On a pu vérifier et voir que notre implémentation était fonctionnelle. On peut remarquer aussi qu'à la réception de deux messages, le buffer ne peut répondre qu'à un seul des messages : ainsi le déroulement de l'expérience dépend de l'implémentation de delta_ext, en fonction de quel message est pris prioritairement.

Exemple, si on priorise le message JOB, on peut perdre un message DONE et ainsi boucler sur l'état C. (Avec la quantité q qui continue d'augmenter).

Simulation d'une variable continue à temps discret (méthode Euler)

Présentation

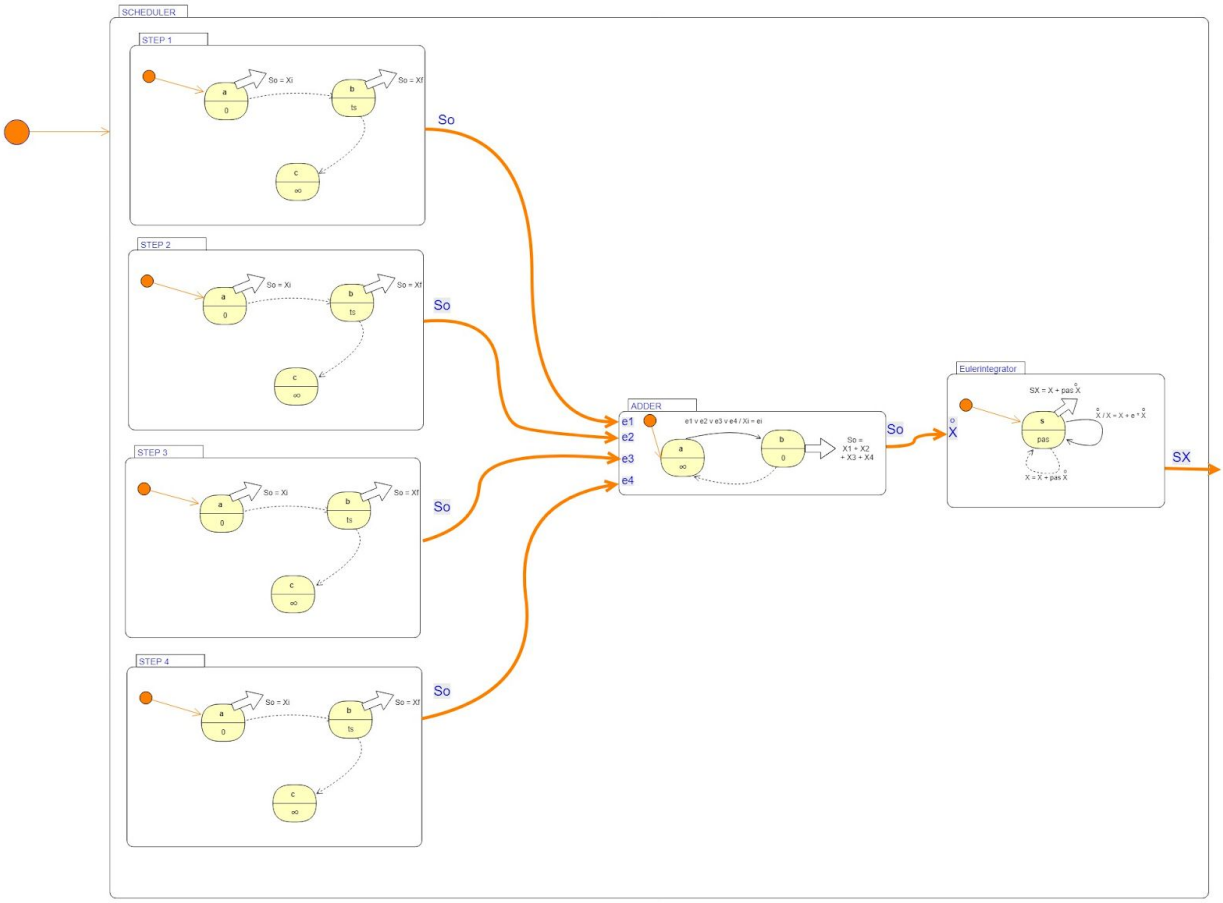
On suppose ici une variable X dont on pourrait connaître (avoir des informations) sur sa dérivée. On cherche à simuler cette variable avec un certain *pas* de temps (donné par le scheduler).

Par exemple, pour une voiture, on peut savoir de combien on accélère en fonction de la pression sur la pédale accélérateur → on obtient alors une information sur la dérivée de la vitesse → ce qui nous permettrait de simuler la vitesse en fonction de l'accélération.

Nous avons ici supposé que nous cherchons à simuler une variable X. Nous avons 4 composantes de la dérivée de cette variable X, que nous appellerons Steps. On somme ces composantes dans un Adder, puis on transmet le résultat à un intégrateur qui en déduira les changements de la variable X.

Le temps d'un état avant de faire la fonction interne delta_int, est défini à l'initialisation avec le pas donné en paramètre du constructeur du composant. (Ce temps est choisi dans le scheduler.)

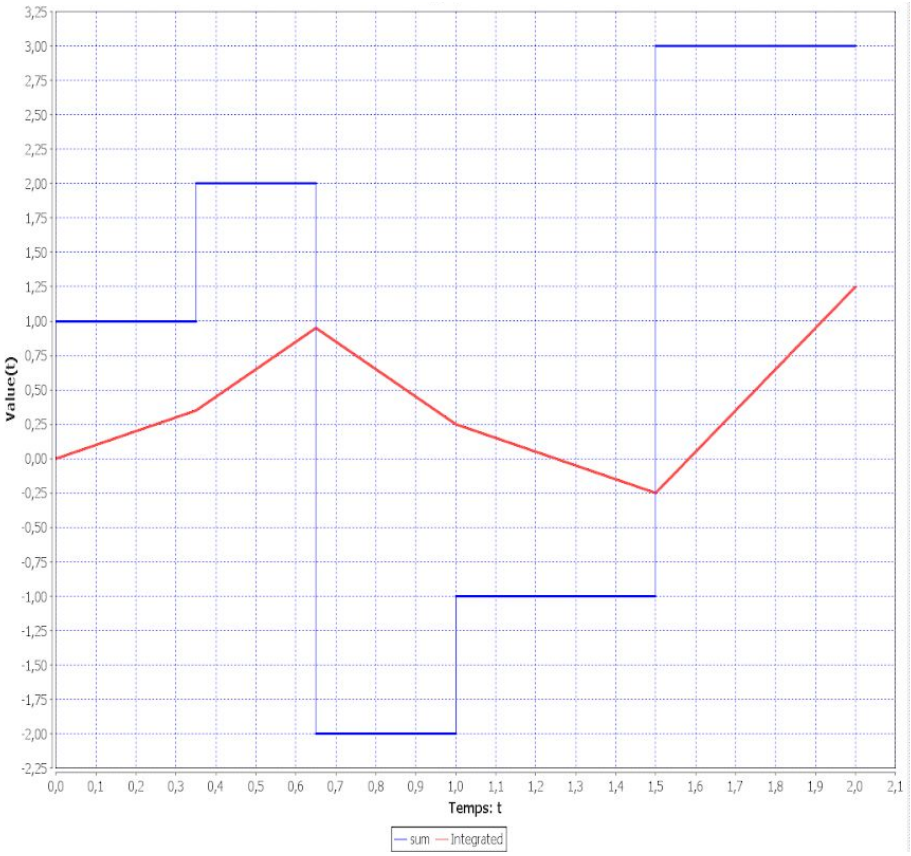
Nous avons ainsi implémenté ces composants comme suit :

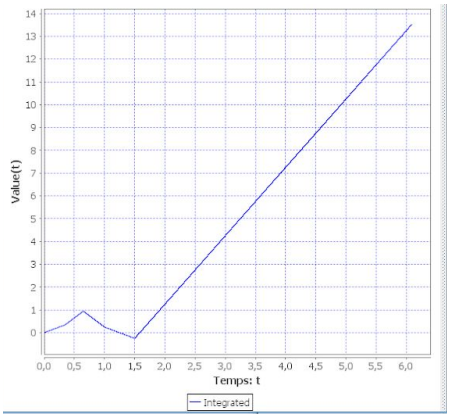


Remarque : ces classes sont placées dans le package *continu*.

Résultats

On peut obtenir ce graphique en fonction du temps, avec en rouge la sortie du Adder, correspondant donc à la dérivée de X, et en bleu la sortie de l'intégrateur, donc le X simulé.





Ci-contre la simulation de X en bleu sur un plus grand temps, avec les paramètres ci-dessous :

```
step1 = new Step("Step1", 1.0, -3.0, 0.65);
components.add(step1);
step2 = new Step("Step2", 0.0, 1.0, 0.35);
components.add(step2);
step3 = new Step("Step3", 0.0, 1.0, 1.0);
components.add(step3);
step4 = new Step("Step4", 0.0, 4.0, 1.5);
components.add(step4);

adder = new Adder("Adder");
components.add(adder);

integrateur = new Integrateur("Integrateur", 0.0001);
components.add(integrateur);
```

(pas de l'intégrateur : 10^{-4})

Simulation d'une variable continue par qss.

Présentation

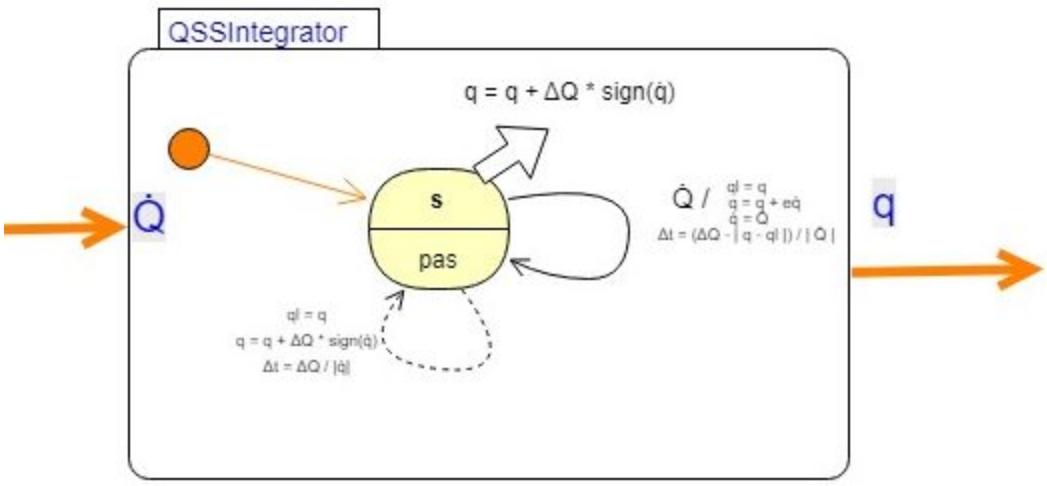
Il n'est pas toujours intéressant de discrétiser en fonction du temps. Il peut être cependant intéressant de discrétiser en fonction de l'**état** d'un composant.

Pour cela, lors d'une nouvelle information en entrée, l'intégrateur met à jour la valeur de X, puis calcule avec sa nouvelle dérivée, dans combien de temps devrait avoir lieu son changement d'état. Il va alors modifier le temps requis dans cet état.

```
@Override
public void delta_int() {
    lastQ = currentQ;
    // I compute the new value of currentQ. the time 'stepTime' has past.
    currentQ += derivativeQ * requiredTime.get(current_state);
    requiredTime.put(current_state, stepQ / Math.abs(derivativeQ));
    ellapsedTime = 0;
}
```

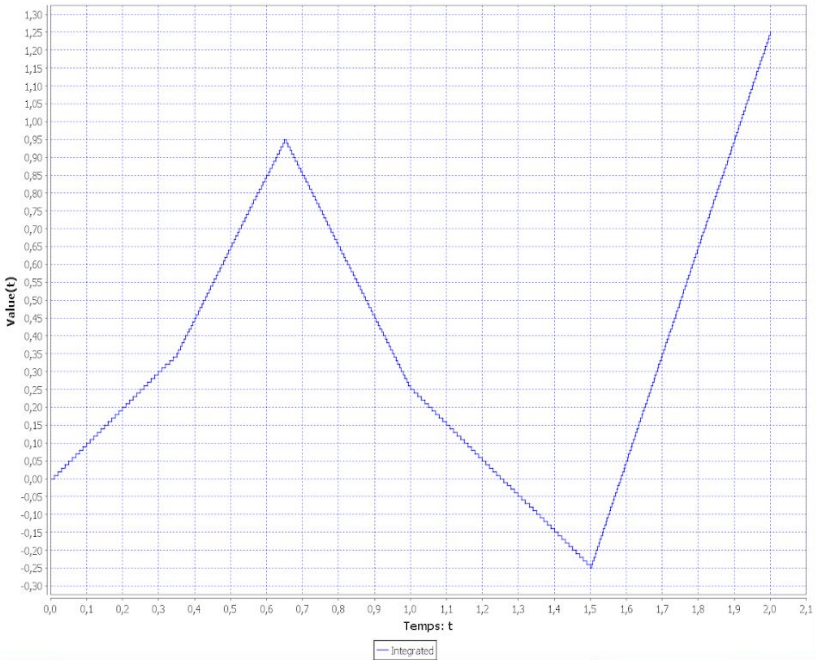
Nous modifions ainsi l'intégrateur comme suit :

DIAGRAMME D'ÉTAT DU QSS_INTEGRATOR



Résultats

En reprenant le schéma précédent : les 4 steps et le Adder, mais en prenant cette intégrateur QSS, nous pouvons obtenir :



avec un pas de 0.01, et les paramètres ci-dessous :

```
step1 = new Step("Step1", 1.0, -3.0, 0.65);
components.add(step1);
step2 = new Step("Step2", 0.0, 1.0, 0.35);
components.add(step2);
step3 = new Step("Step3", 0.0, 1.0, 1.0);
components.add(step3);
step4 = new Step("Step4", 0.0, 4.0, 1.5);
components.add(step4);

adder = new Adder("Adder");
components.add(adder);

integrateur = new QssIntegrateur("Integrateur", 0.01);
components.add(integrateur);
```

Remarque : en faisant bien attention on remarque bien comme prévu que pour une pente plus forte, le pas de temps est alors plus faible.

Simulation d'un tombé de balle.

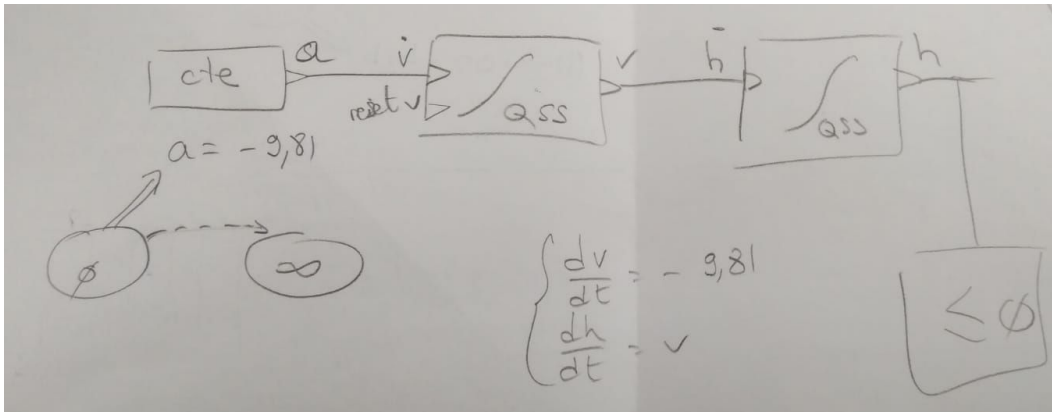
Présentation

Pour obtenir un résultat un peu plus parlant, nous avons simulé en fonction du temps, la position d'une balle qui tombe.

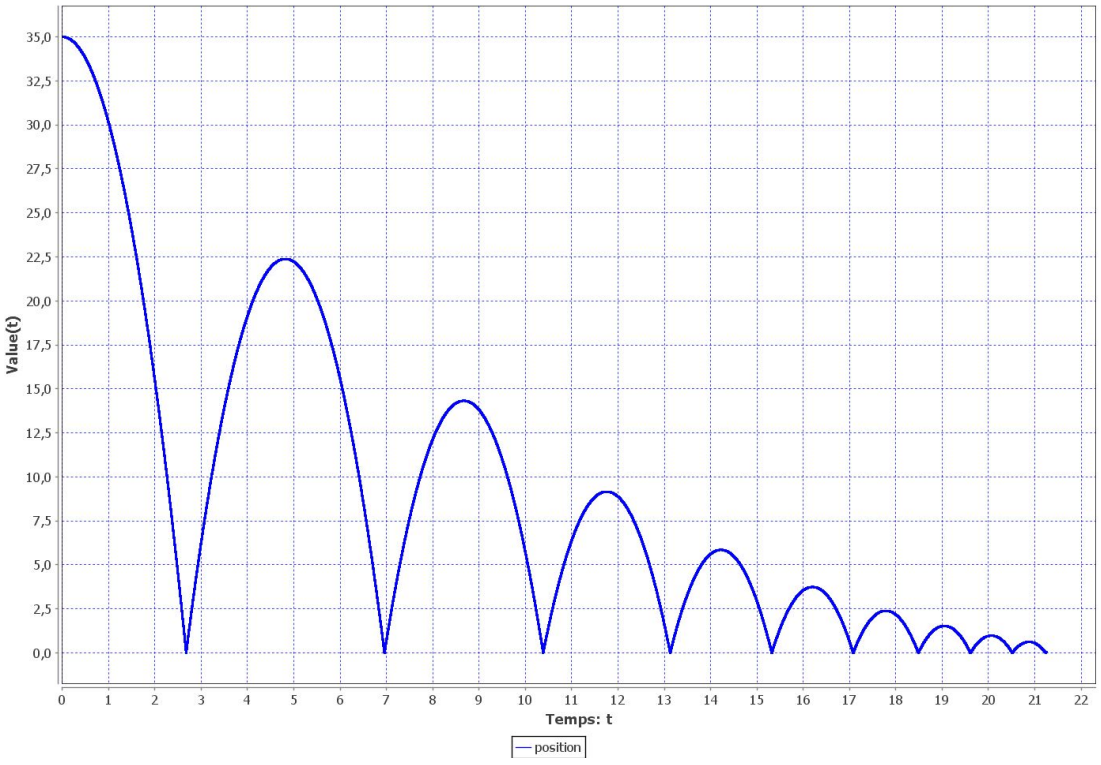
Nous avons défini une hauteur h de la balle à l'initialisation, et nous avons supposé sa vitesse initiale comme nulle. Nous avons pris comme seules forces sur la balle :

- l'accélération terrestre g = 9.8
- la force opposée à celle reçue par le sol, multiplié par un amortissement (<1),

lorsque la balle atteint le sol.



Résultat



ici avec un amortissement de 0.8.

Conclusion.

Nous avons trouvé cela très intéressant de pouvoir implémenter notre propre simulateur et ainsi mieux comprendre comment ces dernier peuvent être implémenter. En revanche il était difficile de modéliser le problème sans avoir pu pratiquer auparavant, nous avons appris beaucoup de nouvelles connaissances durant ce BE. Même si de par les autres UE et partiels, nous avons quelque peu manqué de temps pour finaliser comme nous aurions voulu.
Nous vous remercions en tout cas d'avoir monté ce projet pour nous, étudiants IARF.