Formation IBEX

TP n°2 - **Stratégies et Pavages**

Le but de cet exercice est de programmer SIVIA (*set inversion with interval analysis*), alogrihtme permettant de dessiner un pavage représentant un ensemble $\mathcal{E}$ défini implicitement comme la préimage d'un intervalle par une fonction non-linéaire $f : \mathbb{R}^n \to \mathbb{R}$ quelconque (ici $n = 2$) :

$$\mathcal{E} := \{x \in \mathbb{R}^2,\ \exists y \in [y],\ y = f(x)\}.$$

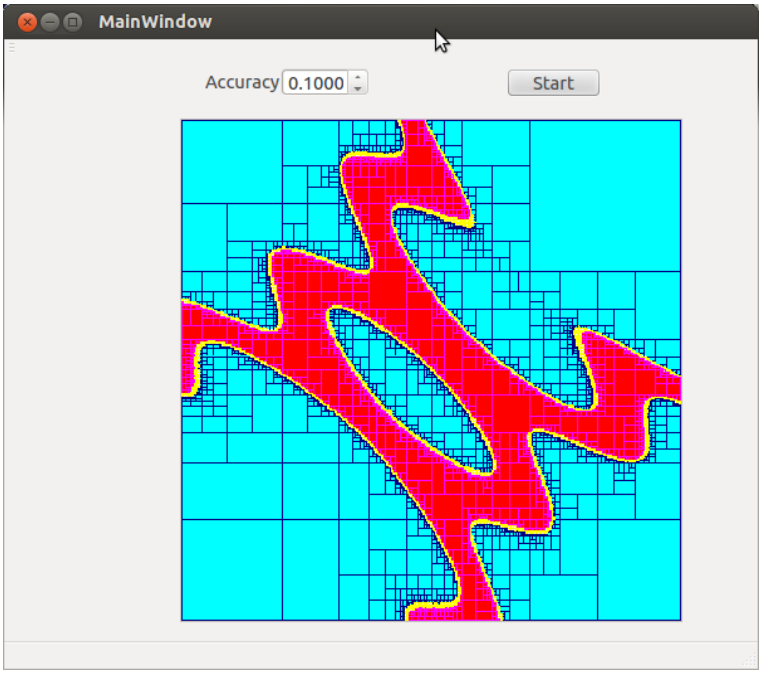


FIGURE 1 – SIVIA (1ère version). Résultat obtenu avec $f(x, y) = sin(x + y) - 0.1 \times x \times y$ et $[y] = [0, 2]$, en alternant simplement une phase d'évaluation et de bissection. Pour une précision $\varepsilon = 0.1$, le nombre de noeuds générés par cet algorithme est **6496**.

L'intérieur de cet ensemble $\mathcal{E}$ apparaît en rouge sur la figure précédente, sa frontière en jaune.

▶ Téléchargez et décompressez l'archive de départ `my_sivia.zip`.

▶ Ouvrez le projet sous Qt et cliquez sur le triangle vert pour tester l'exécution. Rien ne doit s'afficher, ce qui est normal pour le moment.

Votre travail consistera à compléter la class `Sivia`. Vous n'aurez qu'à modifier pour cela le fichier `sivia.cpp`.

1- Evalutation

Dans une première partie, nous allons compléter la fonction auxiliaire suivante de la classe `Sivia` :

```
bool check_and_draw(Function& f, const Interval& y, IntervalVector& box,
                    const QColor& pencolor, const QColor& brushcolor);
```

Cette fonction doit tester si l'inclusion

$$f([x]) \subseteq [y]$$

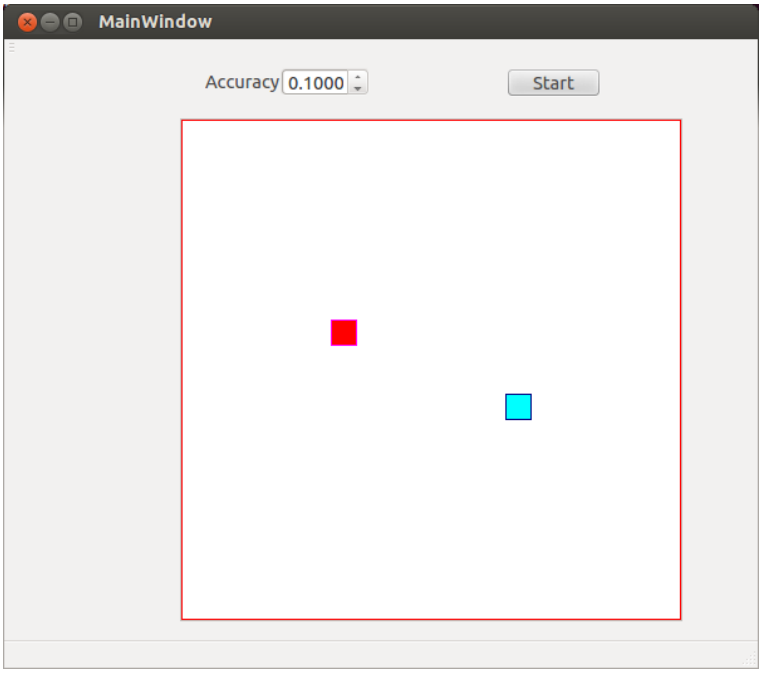
est vérifiée et, le cas échéant :

– afficher la boîte avec les couleurs `pencolor` et `brushcolor` spécifiée
– retourner la valeur `true`
Elle retourne `false` sinon.

Note : étant donné deux intervalles a et b, l'affichage de la boîte (a, b) se fait simplement en écrivant :

`frame.DrawBox(`a, b`,QPen(pencolor),QBrush(brushcolor));`

► Testez votre travail en décommentant le test de question 1 dans le constructeur. Vous devez obtenir l'affichage suivant. Remettez-le en commentaire une fois que ce test est passé.



2- Initialisations

C'est le constructeur de la classe `Sivia` qui contiendra la boucle principale de l'algorithme. Nous travaillons désormais dans ce constructeur. Commençons par déclarer et initialiser les variables qui nous seront nécessaires.

► Créez l'objet `Function` représentant la fonction

$$(x, y) \mapsto sin(x + y) - 0.1 \times x \times y.$$

► Créez la boîte initiale $[-10, 10] \times [-10, 10]$ sur laquelle se fera la recherche

► Créez, pour découper les boîtes, un objet de type `LargestFirst` (utilisez le constructeur par défaut, sans argument).

► Créez un objet de type `stack<IntervalVector>` pour stocker les boîtes de la recherche.

► Créez enfin une variable entière `nb_nodes` qui comptera le nombre de fois où l'on bissectera une boîte.

► Initialisez enfin la pile en y plaçant la boîte initiale.

3- SIVIA (1ère variante)

L'algorithme SIVIA effectue un arbre de recherche en enchaînant les étapes suivantes, tant que la pile n'est pas vide :

1. récupérer la boîte courante $[x]$ en sommet de pile. Utilisez pour cela la fonction `pop` de la class `stack`.
2. **1er test extérieur :** si
$$\forall x \in [x] \quad f(x) \leq 0$$
la boîte est peinte en bleue. Utilisez pour cela la fonction `check_and_draw`. Notez que l'intervalle $]-\infty, 0]$ peut être créé en écrivant `Interval(NEG_INFINITY,0)`. Vous pouvez également récupérer les couleurs données dans le test de la question 1.
3. **2ème test extérieur :** si
$$\forall x \in [x] \quad f(x) \geq 2$$
la boîte est également peinte en bleue
4. **test intérieur :** si
$$\forall x \in [x] \quad 0 \leq f(x) \leq 2$$
la boîte est peinte en rouge.
5. si aucun des tests précédents n'a retourné `true` et si $[x]$ a un diamètre maximal inférieur à ε alors l'afficher en jaune. Utilisez pour cela la fonction `max_diam` de `IntervalVector`.
6. sinon, couper $[x]$ en deux, et stocker les deux sous-boîtes dans la pile. Utilisez pour cela la fonction `bisect` de `LargestFirst` et la fonction `push` de la class `stack` pour l'ajout sur la pile.

▶ Programmez l'algorithme Sivia. Le résultat attendu est celui de la figure 1.

4- Contracteurs

Nous allons améliorer cet algorithme en contractant à chaque étape la boîte courante soit par rapport à la contrainte
$$f(x) \in [y] \tag{1}$$
auquel cas la partie contractée sera *extérieure* à $\mathcal{E}$ et peinte en bleue, soit par rapport à la contrainte
$$f(x) \notin [y] \tag{2}$$
auquel cas la partie contractée sera *intérieure* à $\mathcal{E}$ et peinte en rouge.

Nous commençons par créer les contracteurs en question (toujours dans le constructeur de `Sivia`).

Ibex permet de créer un contracteur par rapport à une égalité ou une inégalité, c.a.d., une *contrainte numérique*. La première étape consiste donc à définir cette contrainte numérique. Pour cela, on crée un objet (intermédiaire) de type `NumConstraint`. Cet objet est ensuite donné au constructeur de la classe `CtcFwdBwd` (classe correspondant aux contracteurs que nous utiliserons).

La ligne suivante par exemple, crée la contrainte `c` : $x+y=z$ et ensuite un contracteur `ctc` pour cette contrainte :

```
Variable x,y,z;
NumConstraint c(x,y,x+y=z);
CtcFwdBwd ctc(c);
```

▶ En vous inspirant de l'exemple ci-dessous, créez des contracteurs pour les 4 contraintes suivantes :
$$f(x) < 0, \quad f(x) \geq 0, \quad f(x) \leq 2 \quad \text{et} \quad f(x) > 2.$$

▶ Grâce à la classe `CtcCompo`, construisez un contracteur pour
$$(1) \iff f(x) \geq 0 \wedge f(x) \leq 2.$$

▶ De même, grâce à la classe `CtcUnion`, construisez à partir des contracteurs précédent un contracteur pour
$$(2) \iff f(x) < 0 \vee f(x) > 2.$$

5- Affichage de la Trace

▶ Décommentez la fonction suivante déclarée dans `sivia.h` :

`contract_and_draw(Ctc& c, IntervalVector& x, const QColor& pencolor, const QColor& brushcolor).`

Cette fonction doit contracter la boîte `x` avec le contracteur c et afficher la *trace* avec les couleurs spécifées, la trace étant définie de la façon suivante :

$$\text{trace}(c, [x]) := [x] \backslash c([x]).$$

▶ implémentez la fonction `contract_and_draw`. Deux remarques importantes :

1. Utilisez la fonction `diff` de la classe `IntervalVector` pour calculer la différence ensembliste entre deux boîtes. Cette fonction stocke le résultat sous forme d'un tableau de boîtes dont l'adresse est passé en argument (consultez l'API). Pensez à bien désallouer ce tableau après usage ; cette fonction ayant vocation à être appelée de façon répétée au cours de la recherche, les fuites mémoire, en s'accumulant, peuvent faire planter le programme.
2. lorsque la boîte est entièrement vidée par le contrateur, une exception de type `EmptyBoxException` est levée ; attrappez cette exception dans un bloc `try {...} catch { }` pour pouvoir afficher la boîte même dans ce cas.

6- SIVIA (2ème variante)

▶ Remplacez dans la boucle principale de votre programme les évaluations par des contractions. N'oubliez pas qu'une boîte peut être vidée par un contracteur et qu'il convient, dans ce cas, de passer directement à la boîte suivante. Le résultat attendu est celui de la figure 2.

FIGURE 2 – SIVIA (2ème version). Le resultat est obtenu en alternant contraction et bissection. Le nombre de noeuds générés par l'algorithme est plus faible, **2623**, et on peut s'aperçevoir que la frontière entre l'intérieur et l'extérieur est plus fine.