

Programmation par Contracteurs sous Quimper

Travaux Pratiques

Ecole des JDMACS 2009

Gilles Chabert et Luc Jaulin

20 Mars 2009

Prérequis : connaître les principes de base de la programmation par contracteurs et avoir installé Ibex/Quimper.

1 Introduction au calcul ensembliste

1.1 Variables, contraintes, contracteurs

Un programme **Quimper** comporte quatre parties :

- déclaration de constantes (mot-clé **constants**)
- déclaration de variables (mot-clé **variables**)
- déclaration de fonctions
- déclaration de contraintes (mot-clé **constraint**) ou de liste de contraintes (mot-clé **constraint-list**)
- déclaration de contracteurs (mot-clé **contractor**) ou de liste de contracteurs (mot-clé **contractor-list**)

Quimper peut associer un contracteur par défaut à une contrainte. Il suffit pour cela d'écrire directement une contrainte dans la définition d'un contracteur pour qu'elle soit « transformée » en contracteur. Exemple (cf. *circle1.qpr*) :

```
variables
  x in [-7,7];
  y in [-7,7];

contractor circle // on donne simplement un nom (circle) au contracteur suivant:
  sqrt(x^2+y^2)=5; // ici, l'expression (x^2+y^2)=5 est un contracteur.
end
```

Enfin, **Quimper** possède un contracteur paramétré **maxDiamGT**(ε) (ε étant un nombre réel), qui retire (contracte à l'ensemble vide) toutes les boîtes dont le rayon sur chaque dimension est plus petit que ε (on dira par la suite « de taille ε »).

1.2 L'outil qPave

qPave est un programme dédié aux problèmes à deux dimensions, c.a.d., comportant deux variables¹. Lorsqu'un programme **Quimper** est chargé sous **qPave**, ce dernier permet de lancer un algorithme de pavage avec les contracteurs *top-level* définis par le mot-clé **contractor** dans le programme (tel que **circle** ci-dessus).

Pour représenter un cercle de rayon 5 avec des boîtes de taille plus petite que 0.1, il faut donc ajouter dans le programme précédent un autre contracteur *top-level* :

```
contractor isthick
  maxdiamGT(0.1)
end
```

Sous **qPave**, il faut utiliser l'option **open** du menu **file** pour charger un programme **Quimper**. S'il n'y a pas d'erreur de syntaxe, les contracteurs *top-level* apparaissent alors sur la gauche. Le fait de cliquer sur **[add all]** les transmet au paveur. Il suffit ensuite de cliquer sur **[go]** pour générer le pavage.

¹L'outil **qSolve** (non graphique) permet de traiter des systèmes avec un nombre arbitraire de variables.

Exercice 1 - Tracé de courbe (*solution : sin.qpr*)

1. Ecrivez le programme **Quimper** dessinant la courbe $y = \sin(x)$ pour x variant entre -2 et 2 avec des boîtes de taille inférieure à 10^{-1} .
2. Exécutez-le sous **qPave** (observez les couleurs).
3. Modifiez votre programme pour qu'il dessine $y = \sin(1/x)$.

1.3 Constantes, vecteurs, matrices, fonctions

Il est possible de créer des variables vectorielles ou matricielles et d'introduire des fonctions pour éviter de dupliquer des expressions mathématiques dans le programme. Il est possible également de définir des constantes (scalaires, vectorielles ou matricielles).

Ainsi, on peut réécrire le programme précédent (cf. *circle2.qpr*) en considérant que le cercle est défini par la contrainte qu'un vecteur x à deux dimensions est à distance 5 du point $(0,0)$:

```
constants
  zero in [0 ; 0]; // ! ne pas confondre avec [0,0] (l'intervalle nul)

variables
  x[2] in [[-7,7] ; [-7,7]];

function d=dist(x[2],y[2])
  d = sqrt((x[1]-y[1])^2 + (x[2]-y[2])^2);
end

contractor circle
  dist(x,zero)=5;
end
```

1.4 Opérations de base sur les contracteurs

L'opération la plus simple entre deux contracteurs C_1 et C_2 est l'intersection. Elle se fait par le mot-clé **inter** : il suffit d'écrire C_1 **inter** C_2 .

Exercice 2 - Système d'équation (*solution : equ.qpr*)

1. Ecrivez le programme **Quimper** représentant les vecteurs x vérifiant à la fois $x_2 = f_1(x_1)$ et $x_2 = f_2(x_1)$ avec $f_1(x) = \sin(x)$ et $f_2(x) = 0.05x$. Taille des boîtes : 0.1.
2. Ajouter une incertitude sur les équations : modifiez votre programme pour obtenir l'ensemble des x vérifiant $x_2 = f_1(x_1) \pm \varepsilon$ et $x_2 = f_2(x_1) \pm \varepsilon$ où ε pourra être modifié facilement. Fixez $\varepsilon = 0.1$.

Il est possible également de faire une union entre deux contracteurs, via le mot-clé **union**.

Par exemple, le contracteur suivant :

```
contractor mystere
  sqrt(x^2+y^2) > 6 union sqrt(x^2+y^2) < 4;
end
```

enlèvera d'une boîte uniquement des points (x,y) ne vérifiant pas ($\sqrt{x^2+y^2} > 6 \vee \sqrt{x^2+y^2} < 4$). Il enlève donc uniquement des points (x,y) qui vérifient $\sqrt{x^2+y^2} \in [4,6]$.

Exercice 3 - Inversion ensembliste (*solution : equ2.qpr*)

1. Fixez $\varepsilon = 0.5$ dans le programme précédent et observez.
2. Ajoutez un contracteur permettant également de décrire efficacement l'intérieur de l'ensemble solution. (*Remarque : vous pouvez utiliser dans une expression les opérations `sup(x)` et `inf(x)` qui retournent les bornes supérieures et inférieures du domaine « courant » de x .*)

Les contracteurs permettent également de représenter plusieurs ensembles. Etant donné une fonction f , supposons que l'on souhaite faire un pavage du plan permettant de déterminer si un point x vérifie $f(x) > 0.4$, ou, dans le cas contraire, s'il vérifie au moins $f(x) > 0.2$.

Difficulté : le fait de créer un contracteur qui décrit l'extérieur de la région $\{f > 0.4\}$ enlèvera indifféremment des points x qui vérifient $f(x) > 0.2$ et d'autres qui ne le vérifient pas. De même, décrire l'intérieur de la région $\{f > 0.2\}$ ôtera des points x qui vérifient $f(x) > 0.4$. Il faut donc introduire des intersections/unions pour éviter cela. Ainsi, le contracteur extérieur devra enlever les points qui n'appartiennent à aucune des deux régions, c.a.d., ne vérifiant pas $f(x) > 0.2$. Le contracteur intérieur de la région $\{f > 0.2\}$ ne devra retirer que les points intérieurs à cette région et extérieurs à la région $\{f > 0.4\}$. Cela peut s'écrire (*cf. level1.qpr*)

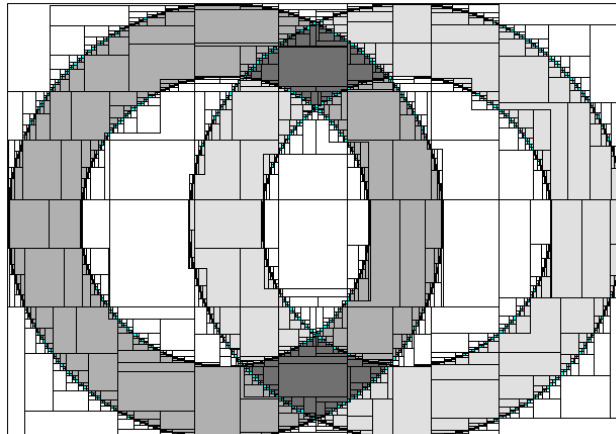
```
contractor level1
  f(x)<0.4;
end

contractor level2
  f(x) not-in [0.2,0.4];
end

contractor toolow
  f(x)>=0.2;
end
```

Exercice 4 - Disjonction ensembliste (*solution : rings1.qpr et rings2.qpr*) Le but de cet exercice est d'obtenir la figure ci-dessous, représentant deux anneaux d'équation respective $dist(x, (0,0)) \in [4, 6]$ et $dist(x, (5,0)) \in [4, 6]$.

1. Faites le programme **Quimper** calculant un pavage des deux anneaux (un pour chaque) et où l'intersection commune est représentée d'une couleur différente. Taille des boîtes à la frontière : 0.1.
2. Epaississez les frontières en remplaçant le seuil 0.1 par la valeur 0.5.
3. Modifiez le programme pour « enlever » les frontières à l'intérieur des anneaux c.a.d. : si on peut prouver qu'un point appartient à un anneau mais pas à l'intersection des deux ce point apparaît dans le sous-pavage intérieur de l'anneau. (*Indication : introduisez le contracteur `maxDiamGT` dans les unions/intersections*)



1.5 Boucles

Il est possible d'effectuer des « boucles de construction » avec les contracteurs dans une syntaxe «à la matlab». Par exemple, pour construire l'intersection des contracteurs associés à 10 contraintes, il est souvent préférable de créer d'abord une liste de contraintes (mot-clé `constraint-list`) :

```
constraint-list my_constraints // liste de contraintes (séparées par des points-virgules)
...; // 1ère contrainte
...;
...; // 10ème contrainte
end
```

puis de construire le contracteur au moyen d'une boucle de construction :

```
contractor intersect_all
inter i=1:10;
    my_constraints(i) // contracteur associé à la ième contrainte de la liste
end;
end
```

Bien entendu, la boucle `inter` constitue un « bloc » de type *contracteur* et peut donc être combiné avec d'autres opérations d'union et d'intersection. De même, il existe la boucle `union`.

Enfin, il existe la boucle `for` permettant de construire une liste de **contraintes**, exemple :

```
Variables
x[10];

constraint-list my_constraints
for i=1:10;
    x[i] <= i;
end;
```

De nouveau, la boucle `for` constitue un « bloc » de type *liste de contraintes* et peut être suivi ou précédé par d'autres contraintes.

2 Exemple de problème : la commande robuste

Supposons que l'on souhaite réguler le roulis ϕ d'une moto à très faible vitesse avec l'angle du guidon.

L'objectif est de prouver que le système est robuste, c'est à dire qu'il est stable dans tous les cas de figure. En effet, la stabilité du système prend en compte les valeurs possibles de ϕ , mais pour une valeur fixée des paramètres physiques de la moto, du régulateur et du capteur utilisé pour mesurer le roulis. La stabilité est *robuste* lorsque le système est stable pour toutes les valeurs possibles de ces paramètres.

La relation entrée-sortie du système régulé est :

$$\phi(s) = \frac{\alpha_2 + \alpha_3 s}{(s^2 - \alpha_1)(\tau s + 1) + (\alpha_2 + \alpha_3 s)(1 + 2s + ks^2)} \phi_d(s)$$

où :

- $\alpha_1, \alpha_2, \alpha_3, \tau$ et k sont les différents paramètres. Leurs domaines possibles sont les suivants : $\alpha_1 \in [8.8, 9.2]$, $\alpha_2 \in [2.8, 3.2]$, $\alpha_3 \in [0.8, 1.2]$, $\tau \in [1.8, 2.2]$, $k \in [-3.2, -2.8]$.
- ϕ_d est l'angle de roulis désiré

Le polynôme caractéristique de la relation est :

$$(s^2 - \alpha_1)(\tau s + 1) + (\alpha_2 + \alpha_3 s)(1 + 2s + ks^2) = a_3 s^3 + a_2 s^2 + a_1 s + a_0,$$

avec $a_3 = \tau + \alpha_3 k$, $a_2 = \alpha_2 k + 2\alpha_3 + 1$, $a_1 = \alpha_3 - \alpha_1 \tau + 2\alpha_2$ et $a_0 = -\alpha_1 + \alpha_2$. En utilisant la table de Routh, on conclut que le système est stable si $a_3, a_2, \frac{a_2 a_1 - a_3 a_0}{a_2}$ et a_0 sont de même signe.

Exercice 5 - Commande robuste (*solution : control.qpr*)

Ecrivez le programme **Quimper** déterminant la stabilité du système en séparant la déclaration des contraintes et des contracteurs (utilisation d’une boucle de construction).

Indications : Vous pouvez utiliser les opérateurs min et max dans des expressions.

3 Exemple de problème : le SLAM

Un robot autonome se déplace dans le plan suivant une équation différentielle qui une fois discrétisée s’écrit très simplement :

$$p_{i+1} = p_i + \delta_t v_i,$$

où $\delta_t = 0.01$ est le pas de temps et v_i est un vecteur vitesse dont on suppose que le domaine est suffisamment grand pour à la fois prendre en compte les erreurs de mesure et rendre la discrétisation “rigoureuse”.

Le robot détecte également des objets appelés *balises* à certains instants. A chaque détection, il n’est capable de déterminer que la **distance** qui le sépare de l’objet détecté. L’intérêt des détections est de permettre au robot de se recalculer : si un objet a été localisé de façon assez précise à un moment donné, le robot peut se servir de cette estimation lorsqu’il le redétecte pour affiner sa propre position (grâce à la connaissance de la distance qui le sépare de l’objet).

Le but est de calculer le plus précisément possible une estimation rigoureuse de la trajectoire du robot.

3.1 L’outil qTraj

Pour cela, nous utiliserons l’outil **qTraj** dont les différences avec **qPave** sont les suivantes :

1. **qTraj** ne permet que d’appliquer des contracteurs (pas de découpage des domaines).
2. **qTraj** permet d’afficher sur le même plan des boîtes supposées correspondre à la position d’un même objet à des instants différents. Ces boîtes sont les domaines d’une variable à deux dimensions : la première indiquant le nombre d’instants, la deuxième étant fixée à 2 (coordonnées dans le plan).

L’usage est très simple : une fois le programme **Quimper** ouvert, il faut commencer par saisir la variable « position ». Il ne reste alors plus qu’à répéter les actions suivantes : sélectionner un contracteur, l’appliquer en cliquant sur **[contract]** puis observer le résultat produit en cliquant sur **[plot]**.

3.2 Importation de données

Le but ici est simplement d’estimer la trajectoire en intégrant les incertitudes des vecteurs vitesse et d’observer que le robot « se perd » petit à petit.

Les vecteurs vitesse mesurés à chaque pas de temps sont enregistrés dans le fichier de données **speed.dat**.

Initialiser chaque vecteur vitesse avec son domaine dans le programme **Quimper** serait très fastidieux. Les outils **Quimper** permettent d’éviter cela grâce à l’importation de données. Il est possible en effet de laisser les domaines à leur valeur par défaut $([-\infty, +\infty])$ puis de les charger *on-line* en cliquant sur le bouton **[import]**. Il suffit alors de choisir le fichier de données puis de saisir le vecteur de variables concerné.

Exercice 6 - Estimation sans balise (*solution : slam1.qpr*)

1. Ecrivez le programme **Quimper** permettant d’estimer la trajectoire du robot, sachant que :
 - la position initiale du robot est $(0, 0)$
 - les vecteurs vitesses seront chargés à l’exécution

Indication : utilisez les boucles for et inter.

2. Lancez **qTraj**, ouvrez votre programme et indiquez la variable de position. (*Remarque : votre variable de position doit être définie comme un vecteur $p[x][2]$, où x est la taille de la discrétisation.*)
3. Importez les données, le domaine du vecteur vitesse à l’instant i étant la i ème ligne du fichier **speed.dat**. (*Remarque : pour que l’importation des données fonctionne, la variable de vitesse doit être définie comme un vecteur $v[x][2]$, où x est le nombre de lignes du fichier.*)
4. Appliquez le contracteur estimant la trajectoire et affichez le résultat.

Nous allons maintenant introduire les détections. Le fichier `pings.txt` contient les informations relatives aux différentes détections (le format est décrit dans le fichier lui-même). Dans le cadre de ce TP, nous allons nous servir uniquement du 4ème objet (l’usage des autres balises étant laissé à titre d’exercice libre).

Dans un premier temps, nous supposons que la position de cet objet est connue : il s’agit du point $(0.1, -0.4)$.

Exercice 7 - Estimation avec balise localisée (solution : `slam2.qpr`)

1. Ajoutez dans votre programme **Quimper** un contracteur prenant en compte l’ensemble des détections du quatrième objet.
2. Observez les recalages visuellement.
3. Une fois les recalages effectués, appliquez plusieurs fois le contracteur de trajectoire. On observe que la contraction ne se propage que dans le sens du temps.
4. Ajoutez un contracteur permettant d’affiner la trajectoire dans le sens inverse.

3.3 Opérations complexes sur les contracteurs

On suppose désormais que la position de la 4ème balise n’est plus connue.

Les contraintes de détection peuvent suffir à estimer la position de la balise, mais il faut pour cela découper les domaines correspondant à sa position. C’est exactement ce qui est illustré à la figure de l’exercice 4 : l’intersection des deux anneaux (les équations de distance), c.a.d., là où peut se trouver la balise, ne peut être obtenue qu’après un découpage des domaines (la première étape de contraction réduit en effet à peine la boîte initiale).

Il est possible d’utiliser un découpage à l’intérieur d’un contracteur, notamment grâce à l’opérateur de *rognage* qu’est **shave**. Etant donné un contracteur C , **shave**(C) est un contracteur plus efficace (mais plus lent) qui pour chaque variable du problème applique C en découpant les domaines en « tranches » suivant cette variable, et retourne la plus petite boîte englobant le résultat.

Le problème ici est qu’il faut éviter que le rognage se fasse suivant les milliers de variables que représentent les tableaux de position et de vitesse, mais uniquement suivant les deux variables correspondant aux coordonnées de la balise.

Or, il est possible sous **Quimper** d’empêcher qu’une variable **x** soit coupée (par le paveur lui-même ou par les opérateurs tels que le rognage). Il faut pour cela non plus la déclarer avec le mot-clé **variables** mais avec le mot-clé **parameters**, de la façon suivante :

Parameters

```
! x; // attention, ne pas oublier le ‘!’
```

Exercice 8 - Estimation avec balise non localisée (solution : `slam3.qpr`)

1. En utilisant l’opérateur **shave** et le mot-clé **parameters**, faites en sorte que les recalages aient lieu avec pour domaine initial de l’objet $[-\infty, +\infty] \times [-\infty, +\infty]$.
2. Retirez le shaving et comparez.