

IBEX
(An **I**nterval-**B**ased **E**Xplorer)

Guide du programmeur

français

Table des matières

1 Premiers pas	7
1.1 Opérations de base	7
1.1.1 Charger un système d'équations	7
1.1.2 Compiler un programme avec IBEX	8
1.1.3 Accès aux domaines	8
1.1.4 Afficher les contraintes et les variables	9
1.1.5 Images intervalles des fonctions	10
1.1.6 Dérivées intervalles	11
1.2 Utilisation de contracteurs standards	11
1.2.1 Projection (HC4Revise)	11
1.2.2 Propagation (HC4)	12
1.2.3 Box Cohérence (Box & BoxNarrow)	13
1.2.4 Newton intervalle (Newton)	13
1.3 Résolution	14
2 Concepts	15
2.1 Espace	15
2.1.1 Notions de domaine en PPC vs analyse numérique	16
2.1.2 Les différents statuts d'une « variable »	17
2.2 Environnement	18

2.3	Entité	19
2.4	Contracteur	20
2.4.1	Un objet « stratégie »	21
2.4.2	Un objet « poids-mouche »	21
2.4.3	Premier exemple	22
2.5	Bissecteur	24
2.5.1	Premier exemple	25
2.6	Paveur	25
2.6.1	Utilisation simplifiée	26
2.6.2	Utilisation générale	26
3	Symboles & Contraintes	29
3.1	Créer un nouvel environnement	29
3.2	Symboles	30
3.2.1	Dimensions d'un symbole	30
3.2.2	Déclaration d'un Symbole	31
3.2.3	Clés des symboles	32
3.2.4	La structure Dim	33
3.3	Contraintes	33
3.3.1	Via la surcharge d'opérateurs	33
3.3.2	Via un fichier <code>Quimper</code>	34
3.4	CSP & Systèmes	35
3.4.1	Création	35
3.4.2	Espace et environnement d'un système	37
4	Contracteurs	39
4.1	Introduction	39
4.2	Règles de construction	40

<i>TABLE DES MATIÈRES</i>	5
4.3 Seuil de contraction	41
4.3.1 La valeur spéciale 0	43
4.4 Indicateurs	44
4.4.1 Exemple illustratif	45
4.4.2 Accès aux indicateurs	47
4.5 Adjacence	47
4.5.1 Généralisation	48
4.5.2 La classe <code>Adjacency</code>	48
5 Bissecteurs	51
6 Pavés	53

Chapitre 1

Premiers pas

1.1 Opérations de base

1.1.1 Charger un système d'équations

Tout d'abord, écrivez le petit système d'équations suivant dans un fichier texte (la syntaxe utilisée est celle du langage Quimper).

Variables

```
x in [0,8];  
y in [2,10];
```

Constraints

```
(x-4)^2+y^2=9;  
x^2+y^2=16;  
end
```

Sauvegardez ce fichier sous le nom `my_system.txt`, puis créez ensuite un nouveau programme C++.

La première chose à faire est d'inclure les fichiers en-tête d'IBEX qui seront utilisés par votre programme. Tous les fichiers en-tête d'IBEX commencent par `Ibex...`. En revanche, les classes correspondantes n'ont pas de préfixes : elles sont définies dans l'espace de nommage `ibex`. Pour pouvoir manipuler directement un système d'équations, il faut donc inclure `IbexSystem.h` et ouvrir l'espace de nommage. Notre programme peut donc commencer ainsi :

```
#include "IbexSystem.h"

using namespace ibex;

int main() {
```

Nous pouvons maintenant charger le fichier `my_system.txt` dans un objet de type `ibex::System`. On utilise pour cela le constructeur de cette classe qui prend comme argument un nom de fichier :

```
System sys("my_system.txt");
```

1.1.2 Compiler un programme avec IBEX

Créez un fichier `makefile` et copiez-collez les lignes suivantes. Les variables servent uniquement à lier votre programme aux bibliothèques IBEX et Profil/Bias.

Remplacez `<mytarget>` et `<mysource>` par les noms appropriés et `<ibexdir>` par le répertoire où se trouve installé IBEX.

1.1.3 Accès aux domaines

L'accès aux domaines se fait à travers un « espace », ici, un champ de la classe `System` nommé `space`. A partir de cet objet, plusieurs fonctions peuvent être invoquées pour inspecter les domaines. La plus conviviale est `domain`, qui prend en argument directement le symbole de la variable :

```
// will display [x]=[0,8]
cout << "[x]=" << sys.space.domain("x") << endl;
```

Mais il existe bien entendu un moyen d'accéder au domaine directement par les indices des variables.

En réalité, tous les domaines sont stockés directement dans un tableau, pour des raisons d'efficacité. Ce tableau est un objet `sys.space.box` de type `INTERVAL_VECTOR`, type qui n'est pas défini dans IBEX mais dans la bibliothèque sous-jacente Profil/Bias. Il existe même une référence directe vers ce tableau dans la classe `System`, il s'agit de `sys.box` (ce n'est qu'un alias). Naturellement, la taille de ce tableau coïncide avec la dimension du problème (ici, 2).

Pour un système, les indices des variables suivent leur ordre de déclaration dans le fichier `Quimper`. Ainsi, dans notre exemple, l'indice de `x` est 0 et celui de `y`, 1.

Les contraintes suivent d'ailleurs la même règle, si bien que chaque fois qu'un numéro de contrainte est attendu, 0 représentera la contrainte $(x-4)^2 + y^2 = 9$ et 1, $x^2 + y^2 = 16$.

Par exemple, un appel à `sys.ctr(0)` retournera un objet représentant la première contrainte. De même, un appel à `sys.space.domain(IBEX_VAR, 0)` retournera de nouveau le domaine de `x`, l'intervalle $[0, 8]$.

Cependant, il y a un « piège » dans ce système d'indices : les indices dans `Profil/Bias` commencent, eux, à 1. Pour tout appel à une fonction de `Profil/Bias`, il faut donc décaler les indice de 1.

Il faut retenir la règle suivante, qui s'applique systématiquement :

Ibex Function	→	indexes start from 0
Bias/Profil Function	→	indexes start from 1

Ainsi, la façon la plus directe d'accéder à un domaine est d'utiliser le vecteur `sys.box`. Mais comme il s'agit d'un objet `Profil/Bias`, les indices doivent être incrémentés de 1 : `sys.box(1)` est le domaine de `x`, la première variable, alors que `sys.box(0)` retourne un résultat indéfini.

```
INTERVAL x=sys.box(1); // sets domain of x to [0,8]
sys.box(2) = 0; // sets domain of y to [0,0]
sys.box(2) = INTERVAL(2,10); // sets domain of y back to [2,10]
```

1.1.4 Afficher les contraintes et les variables

Pour afficher les contraintes, il suffit d'utiliser directement l'opérateur de redirection avec le système :

```
cout << sys << endl;
```

Cette ligne produira l'affichage escompté :

```
((x-4)^2+y^2)-9)=0
(x^2+y^2)-16)=0
```

Pour afficher l'ensemble des domaines, il existe deux façons de procéder. Vous pouvez utiliser le format d'IBEX, qui consiste à aligner les boîtes par colonne afin de faciliter la lecture de boîtes de taille importante. On écrit tout simplement :

```
cout << sys.space << endl;
```

L'affichage produit ici est :

```
x [      0,      8]   y [      2,      10]
```

Si vous n'aimez pas ce format, vous pouvez également utiliser directement l'affichage produit par Profil/Bias sur l'objet INTERVAL_VECTOR :

```
cout << sys.box << endl; // sys.box is an INTERVAL_VECTOR
```

ce qui donnera :

```
([0,8] ; [2,10])
```

1.1.5 Images intervalles des fonctions

Considérons la fonction $f : \mathbb{R}^2 \rightarrow \mathbb{R}^2$ représentée par la première contrainte, c'est à dire

$$f(x, y) := \begin{pmatrix} (x-4)^2 + y^2 - 9 \\ x^2 + y^2 - 16 \end{pmatrix}$$

Calculons l'image par f de la boîte initiale $[0, 8] \times [2, 10]$:

```
// build a vector of size 2
INTERVAL_VECTOR img(2);
// calculate the image of the initial box by f and store the
// result into img
// img(1) is set to ([0,8]-[4,4])^2+[2,10]^2 -[9,9]
// img(2) is set to [0,8]^2+[2,10]^2-16
sys.eval(img);
// display the result
cout << "Evaluation: " << img << endl;
```

L'affichage produit est :

```
Evaluation: ([-5,107] ; [-12,148])
```

1.1.6 Dérivées intervalles

Nous pouvons calculer une enveloppe de la matrice jacobienne ou de Hansen (en utilisant la différentiation automatique), ainsi :

```
// declare a matrix with appropriate dimensions
INTERVAL_MATRIX J(2,2);

// compute jacobian and store the result in J
sys.jacobian(J);
cout << "Jacobian:" << endl << J << endl;

// compute Hansen's matrix and store the result in J
sys.hansen_matrix(J);
cout << "Hansen matrix:" << endl << J << endl;
```

L'affichage produit est :

```
Jacobian:
((-8,8] ; [4,20])
 ([0,16] ; [4,20]))
Hansen matrix:
((-8,8] ; [4,20])
 ([0,16] ; [4,20]))
```

Notez que dans ce cas les deux matrices coïncident.

1.2 Utilisation de contracteurs standards

Les contracteurs standards sont représentés dans `IBEX` par des classes qui dérivent d'une classe abstraite `Contractor`. La principale fonction (virtuelle pure) de cette classe étant

```
void contract() { ... }
```

qui a pour rôle de contracter les domaines dans l'espace du contracteur (c.a.d., les domaines « attachés » au contracteur. Voir §2.1 pour plus de détails sur cette notion).

1.2.1 Projection (HC4Revise)

La projection d'une contrainte (algorithme `hc4revise`) s'obtient via un objet de type `HC4Revise`. Il faut ajouter l'en-tête `IbexHC4Revise.h` pour pouvoir l'utiliser. Il se construit à partir d'une contrainte (ici, `sys.ctr(0)`, c.a.d., la première) et

un espace sur lequel il travaille. L'espace transmis ici est celui du système : le contracteur et le système partageant donc les mêmes domaines.

```
// build the contractor associated to the 1st constraint
HC4Revise hc4revis0(sys.ctr(0), sys.space);
// contract domains inside sys.space
hc4revis0.contract();
cout << sys.box << endl;

// build the contractor associated to the 2nd constraint
HC4Revise hc4revis1(sys.ctr(1), sys.space);
// contract domains inside sys.space
hc4revis1.contract();
cout << sys.box << endl;
```

Affichage produit :

```
([1.7639320225002093,6.2360679774997907] ; [1.9999999999999999,3.0000000000000018])
([2.6457513110645876,3.4641016151377562] ; [1.9999999999999999,3.0000000000000018])
```

1.2.2 Propagation (HC4)

Le mécanisme de propagation est générique dans IBEX dans le sens où il peut fonctionner avec n'importe quel type de contracteurs. Son principe consiste à enchaîner les contracteurs jusqu'à l'obtention d'un point fixe sur les domaines, en minimisant le nombre d'appels inutiles. Il gère pour cela un *agenda* dans lequel seuls les contracteurs potentiellement impactés par le travail effectué sont présents.

Pour l'utiliser, il faut d'abord inclure l'en-tête `IbexPropagation.h`. Pour obtenir l'algorithme `hc4`, c'est à dire, une propagation où chaque contrainte donne lieu au contracteur `hc4revise`, on peut construire l'algorithme explicitement :

```
// we put all the contractors in a vector
vector<const Contractor*> vec;
vec.push_back(&hc4revis0);
vec.push_back(&hc4revis1);

// the propagation will be performed with these contractors
Propagation propag(vec, sys.space);
cout << sys.box << endl;
```

Le résultat affiché est une boîte très précise, qui montre qu'un point fixe a bien eu lieu :

```
(2.8749999999999999[55,145] ; 2.78107443266087[09,72])
```

Une alternative consiste à utiliser l'objet HC4 directement. Ce n'est qu'une sorte de macro qui reprend tout ce qu'on a fait à la main au-dessus.

```
HC4 hc4(sys);

hc4.contract();      // enforce propagation
cout << sys.box << endl;
```

L'affichage produit est le même.

1.2.3 Box Cohérence (Box & BoxNarrow)

L'appel à la box-cohérence ou à l'opérateur local de « box narrowing », s'obtient de façon tout à fait similaire au paragraphe précédent. L'en-tête `IbexBox.h` doit être incluse.

La Box-cohérence peut être appelée directement, via un objet similaire à HC4. Dans l'exemple suivant, nous comparons les résultats de ces deux contracteurs standards.

```
HC4 hc4(sys); // build HC4
Box box(sys); // build Box consistency

// save the initial domains
INTERVAL_VECTOR init=sys.box;

hc4.contract();      // call HC4
cout << sys.box << endl; // print the result

sys.box = init;      // restore the initial domains
box.contract();      // call Box
cout << sys.box << endl; // print the result
```

Sur cet exemple, les deux contracteurs donnent des résultats très proches, à quelques flottants près.

1.2.4 Newton intervalle (Newton)

Terminons ce tour d'horizon des contracteurs standards par le Newton intervalle. Contrairement à la propagation, il ne se construit pas à partir d'une liste de contracteurs. Il ne peut être construit qu'à partir d'un système d'équations directement.

Il nécessite l'en-tête `IbexNewton.h`. Son usage est ensuite élémentaire :

```
sys.box = init;      // restore the initial domains
Newton newton(sys);
```

```
newton.contract();      // perform interval Newton
cout << sys.box << endl;
```

1.3 Résolution

Pour résoudre un système d'équations, on utilise un objet de type `Paver`. Cet objet a pour principe d'appliquer alternativement une liste de contracteurs (dans l'exemple suivant, il n'y en aura qu'un) et un *bissecteur*, objet chargé de découper un intervalle en deux suivant une heuristique qui lui est propre.

L'heuristique choisie dans l'exemple est la plus simple : celle de *round-robin*, qui consiste à balayer systématiquement les variables suivant leur ordre de définition. Les variables trop "petites" (ici, dont le diamètre du domaine est en deça de 1^{-08}) sont retirées de ce processus de balayage afin d'éviter des bissections (points de choix) inutiles. Pour mieux comprendre l'intérêt de ce paramètre "plancher", il suffit d'imaginer qu'un domaine soit réduit à un seul point : dans ce cas, la bissection ne fait que dupliquer l'intervalle, ce qui conduit à une récursion infinie.

L'objet `Paver` peut être construit avec trois arguments : un contracteur, un bissecteur et une valeur qui correspond à la précision souhaitée pour une boîte. Ici, on considère qu'une boîte dont le diamètre sur chaque dimension est au plus 1^{-07} est suffisamment petite pour être considérée comme "solution". Remarquez que ce paramètre a une sémantique proche de celui du round robin : il ne faut notamment pas que le round-robin ait une précision plus grossière que celle du paveur. En effet, il pourrait se produire qu'une boîte ne puisse plus être découpée sans pour autant être assez précise pour que le paveur s'arrête. Il s'ensuivrait de nouveau une boucle infinie.

Les en-tête suivant doivent être ajoutés : `IbexPaver.h` et `IbexBisector.h`.

```
// create a round-robin bisector
RoundRobin rr(sys.space, 1e-08);

// create a paver with a single contractor and the bisector.
Paver paver(hc4,rr, 1e-07);

// (will optimize branch & bound)
paver.solver_mode=true;

// perform branch & bound
paver.explore();

// display some information (computation time, etc.)
paver.report();
```

Chapitre 2

Concepts

Les principaux concepts dans IBEX sont

- l'**environnement** (**Env**), qui centralise tous les symboles et expressions mathématiques. Il joue essentiellement le rôle de « table des symboles »;
- l'**espace** (**Space**), qui contient les domaines associés aux symboles, pour chaque opérateur (contracteur ou bissecteur);
- l'**entité** (**Entity**), qui généralise la notion de variable : suivant le “statut” d’une entité, celle-ci peut être filtrée ou non, bissectée ou non (par exemple, une *variable* est une entité “filtrable” et “bissectable”);
- le **contracteur** (**Contractor**), opérateur réduisant les domaines, équivalent au concept de *propagateur* en domaines discrets;
- le **bissecteur** (**Bissector**), opérateur chargé de découper une boîte en deux sous-boîtes;
- le **paveur** (**Paver**), algorithme généralisant celui de *solveur* pour permettre principalement la description d’une disjonction de variétés de mêmes dimensions (un ensemble de solutions ponctuelles étant un cas particulier).

2.1 Espace

La notion d’espace s’explique par le fait qu’il ne peut pas y avoir une seule structure centrale représentant les domaines. Il y a en fait une *vue* des domaines propre à chaque opérateur (contracteur, bissecteur). Cette notion de vue peut se justifier par la nécessité de traiter des vecteurs de variables “en bloc” dans le cas d’opérateurs issus de l’analyse numérique, et par le fait que des variables doivent être considérées comme de simples paramètres ou constantes dans certaines situations.

2.1.1 Notions de domaine en PPC vs analyse numérique

En programmation par contraintes, on a coutume de considérer un ensemble de contraintes $\{c_1, \dots, c_m\}$ et un ensemble de variables $\{x_1, \dots, x_n\}$, chaque contrainte impliquant un sous-ensemble de variables.

Cette définition convient bien à la PPC qui applique des filtrages "locaux" dont elle propage les résultats à travers le graphe de contraintes. Ainsi, dans la vision PPC, on dit qu'une variable x_i et une contrainte c_j sont reliées si une arête existe entre la variable x_i et la contrainte c_j dans ce graphe.

On peut donc, dans ce contexte, considérer une unique structure contenant les domaines des variables, chaque contrainte se chargeant d'aller lire ou écrire le domaine d'une variable impliquée dans son expression.

En revanche, cette représentation des choses n'est pas adaptée pour l'analyse numérique, notamment dans le cas de systèmes d'équations. En effet, les algorithmes tels que l'itération de Newton considèrent un unique vecteur de variables

$$\mathbf{x} := (x_1, \dots, x_n),$$

et une unique équation vectorielle :

$$f(\mathbf{x}) = 0_{\mathbb{R}^m},$$

avec $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ telle que la j^{eme} composante de f corresponde à l'expression de la j^{eme} contrainte. Exemple :

Vue "PPC"	Vue "numérique"
$vars = \{x_1, x_2, x_3\}$ $ctr s = \{c_1, c_2, c_3\}$ $c_1 : x_1 + x_2 = x_3;$ $c_2 : x_1 = 10 \times x_2;$ $c_3 : x_3 \times x_1 = 1;$	$\mathbf{x} \in \mathbb{R}^3$ $f(\mathbf{x}) = \begin{pmatrix} x_1 + x_2 - x_3 \\ x_1 - 10 \times x_2 \\ x_3 * x_1 - 1 \end{pmatrix} = 0$

Dans cette vision, toutes les contraintes partagent toutes les variables¹.

L'algorithme de Newton (intervalle) applique des opérations directement sur la boîte $[\mathbf{x}]$ représentant le domaine du vecteur \mathbf{x} . Par exemple, il sera amené à multiplier certaines matrices par ce vecteur $[\mathbf{x}]$. Ces produits matriciels sont implémentés directement au niveau de la librairie arithmétique. Il serait très inefficace de redéfinir cette opération de produit matriciel pour que l'accès à la i^{eme} composante de $[\mathbf{x}]$ soit *mappé* sur une structure représentant le domaine de la variable x_i . Autrement dit, il ne peut y avoir une unique structure de données centrale représentant les domaines, sur laquelle travaillerait l'ensemble des opérateurs.

¹la notion de dépendance existe aussi en analyse numérique, mais elle n'est pas symbolique. Elle apparaît numériquement après une phase dite de *préconditionnement*

On pourrait imaginer que cette structure *soit* la boîte $[x]$, c'est à dire que les domaines des variables soit stocké une bonne fois pour toute comme un unique vecteur d'intervalles. Malheureusement, cette solution serait beaucoup trop contraignante : il suffit d'imaginer que l'opérateur de Newton soit appliqué sur un sous-ensemble de variables (comme c'est souvent le cas).

IBEX propose à travers la notion d'*espace* un moyen d'associer à un opérateur (comme celui de Newton) une "vue" particulière des domaines de telle sorte que si cet opérateur traite 3 variables, il ne voie que celles-ci ; c'est à dire que pour cet opérateur, il n'y a bien en tout et pour tout que 3 variables. Un espace `space` contient principalement une boîte `space.box` contenant une copie *locale* des domaines.

Pour fixer les idées, supposons que l'on ait n variables dans un système de contraintes et que l'on souhaite appliquer l'opérateur de Newton avec 2 contraintes sur les variables x_4 et x_7 . On met en place un espace global `global_space` qui regroupe tous les domaines $[x_1], \dots, [x_n]$ puis un espace propre à l'opérateur de Newton, notons-le `newton_space`, réduit à 2 intervalles. Lorsque cet opérateur est invoqué, les domaines sont importés depuis l'espace global. Ils sont exportés une fois le travail terminé :

```
// importation des domaines
newton_space.box(1)=global_space.box(4);
newton_space.box(2)=global_space.box(7);

// application de l'opérateur de Newton
...

// exportation
global_space.box(4)=newton_space.box(1);
global_space.box(7)=newton_space.box(2);
```

Evidemment, ces mécanismes de synchronisation (ici réalisés à la main) peuvent être fait automatiquement.

On peut déjà observer qu'il faut distinguer les indices des variables de leur noms. Ainsi, si la variable x_4 a bien son domaine à l'indice 4 dans `global_space`, son indice dans `newton_space` est 1.

2.1.2 Les différents statuts d'une « variable »

Une autre raison motive la notion d'espace. Un grand nombre d'opérateurs manipule un ensemble de contraintes. Or, parmi l'ensemble des variables impliquées par ces contraintes, toutes n'ont pas le même statut. Prenons de nouveau l'exemple de l'opérateur de Newton. Supposons que parmi les 100 variables du problème initial, 90 soient quasiment fixées (leur domaine est un intervalle très fin). On cherche à

réduire le domaine des 10 variables restantes. Supposons qu'elles apparaissent dans 10 contraintes mais que ces contraintes impliquent également en tout 30 variables « fixées ».

On peut souhaiter dès lors appliquer un Newton avec ces 10 contraintes, mais clairement, il y aura 10 « vraies » variables et 30 « paramètres ».

La notion d'espace permet, d'un opérateur à l'autre, de spécifier quel est le statut d'une variable.

Afin de conserver le mot « variable » aux « vraies » variables, nous utiliserons le terme plus général d'*entité*.

Une entité est donc un symbole apparaissant dans des contraintes, et auquel est associé un domaine. Pour certains opérateurs, cette entité pourra devenir une variable ou un paramètre. Nous verrons qu'il existe en fait 4 statuts différents.

Lorsqu'un espace est créé, le statut est spécifié par des constantes prédéfinies telles que `IBEX_VAR` ou `IBEX_SYB`. Voici un exemple de code illustrant la façon dont est spécifié le statut d'une entité :

```
// Création d'une fabrique d'espace
SpaceFactory fac(...);

// le symbole x est une variable dans space
fac.set_entity("x", IBEX_VAR);

// le symbole y est un paramètre dans space
fac.set_entity("y", IBEX_SYB);

// Création de l'espace
Space space(fac);
```

2.2 Environnement

Nous avons introduit à la section précédente la notion d'espace. Chaque opérateur travaille sur un espace particulier, mais comment savoir à quelle entité se réfère un symbole "x" dans une contrainte? comment synchroniser automatiquement deux espaces différents (importer/exporter les domaines) ?

La notion d'environnement est la structure centrale qui permet de faire le liens entre les espaces.

Elle n'est en fait qu'une table des symboles améliorée, chaque symbole ayant une clé unique qui sert de référence. Chaque espace possède, en interne, la table des clés correspondant à ses entités. Lorsque un espace `space1` doit importer les domaines d'une espace `space2`, le domaine de chaque symbole "x" partagé par les

deux espaces va pouvoir être copié au bon endroit grâce à ces tables.

Pour résumer :

- un **symbole** est défini dans l'**environnement** et possède une **clé**.
- une **entité** est définie dans un **espace** et possède un **indice** et un **domaine**.
- l'espace associe à chaque **entité** la **clé** du symbole correspondant.

Cela est illustré sur la figure 2.1.

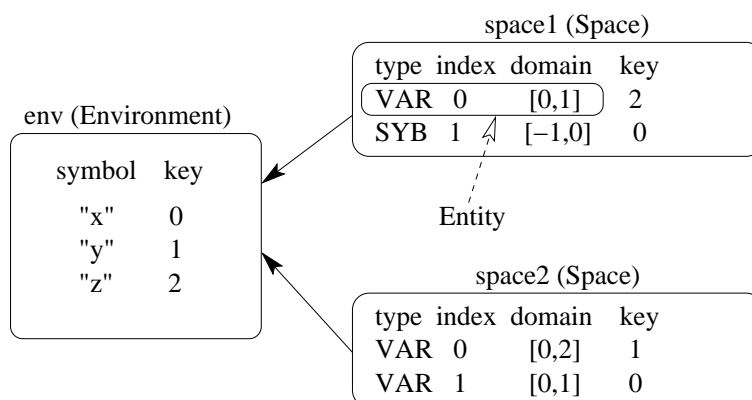


FIG. 2.1 – Exemple de deux espaces liés au même environnement.

L'espace **space1** contient une variable (z) et un paramètre (x) tandis que l'espace **space2** contient deux variables (y et x). Les entités x des deux espaces correspondent au même symbole (clé n°0).

L'environnement ne contient pas uniquement les symboles. Il contient également les expressions mathématiques des contraintes. La raison à cela est que tout est fait en sorte que les concepts d'IBEX (autre que l'environnement) puissent « ignorer » les symboles des entités qu'ils manipulent. En effet, le fonctionnement d'un algorithme de branch & bound doit être indépendant du fait que le symbole d'une variable soit "x" ou "y". Or, les contraintes sont des expressions mathématiques construites à partir des symboles. Elles sont donc, elles, liées au symboles et trouvent leur place dans l'environnement.

2.3 Entité

Une des spécificités des problèmes numériques est que les modèles sont incertains. En effet, on a rarement en pratique un système d'équation $f(x) = 0$ mais plutôt $f(x,p) = 0$ où p désigne un ensemble de paramètres plus ou moins connus. Une solution pourrait être de considérer p comme des variables supplémentaires, mais c'est en général un très mauvais choix du point de vue des performances.

On peut imaginer, par exemple, un point (x, y) que l'on cherche à déterminer dans le plan et se trouvant à l'intersection de deux cercles dont les rayons sont les portées maximales d'antennes, résultant d'un calcul faisant intervenir 10 paramètres physiques. Chacun de ces paramètres peut être incertain, et donc avoir un petit domaine de variations.

Ces domaines ne sont pas du même ordre que ceux des variables x et y , au départ $(-\infty, +\infty)$. La résolution devra s'effectuer en "travaillant" sur x et y et en considérant le reste comme des constantes floutées. Autrement dit, on reste dans un problème à 2 dimensions (et non 12).

Il est clair qu'en pratique, on ne cherche pas à réduire l'incertitude sur les paramètres. Les paramètres sont les « entrées » du problème, les coordonnées x et y la « sortie ». Ce sont les paramètres qui contraignent les coordonnées du point et non l'inverse.

Malgré tout, notons qu'il pourrait être possible de réduire les paramètres. Dans notre exemple, l'existence d'une intersection commune aux deux cercles pourrait en effet amener à éliminer certaines combinaisons de paramètres conduisant à une intersection vide. Mais ce n'est pas, de toute façon, une information qui nous intéresse a priori.

Rappelons que ce qui prend le nom de *paramètre* et de *variable* du point de vue numérique ne coïncide pas nécessairement avec le sens physique. Ainsi, prenons par exemple un problème d'identification où l'on cherche à déterminer les paramètres physiques d'un système, disons des grandeurs géométriques, à partir de mesures de son état. L'état devient numériquement un paramètre, et la géométrie la variable.

Les deux principales entités dans IBEX sont **var** (« *variable* ») et **syb** (« *symbolic constant* »). D'autres types d'entités ont été prévues, mais ils n'ont pas trouvé de grande utilité jusqu'ici. Nous pouvons les ignorer. La signification des acronymes donnée entre parenthèse importe peu. La seule différence entre les entités se fait d'un point de vue opérationnel. Lorsqu'un contracteur est amené à réduire un domaine, il n'est autorisé à le faire que s'il s'agit d'une variable. De même, lorsqu'un bissecteur doit choisir un domaine à couper, il ne fait son choix que parmi les domaines des variables (**var**).

En réalité, dans un arbre de recherche, les domaines de variables sont copiés alors que ceux des **syb** sont partagés. Il est donc essentiel qu'aucun opérateur (contracteur/bissecteur) ne modifie le domaine d'un **syb**.

2.4 Contracteur

Il est possible d'imaginer un grand nombre d'opérations sur des boîtes. Des exemples simples seraient une fonction $vol : \mathbb{I}\mathbb{R}^n \rightarrow \mathbb{R}$ qui retourne le volume ou une fonction $rot : \mathbb{I}\mathbb{R}^n \rightarrow \mathbb{I}\mathbb{R}^n$ qui applique une rotation.

Les seules opérations qui vont nous intéresser sont :

- les *bissecteurs*, fonctions de $\mathbb{R}^n \rightarrow \mathbb{R}^{2n}$ qui coupent une boîte en deux sous-boîtes.
- les *contracteurs*, fonctions de $\mathbb{R}^n \rightarrow \mathbb{R}^n$ qui transforment une boîte en une boîte plus petite

Un contracteur C doit donc vérifier la propriété suivante dite de *contraction* :

$$\forall [x] \in \mathbb{R}^n, \mathcal{C}([x]) \subseteq [x]$$

Un contracteur correspond très simplement à l'idée intuitive d'un « filtre » : c'est un opérateur qui, appliqué à une boîte, retourne une boîte plus petite.

2.4.1 Un objet « stratégie »

Dans IBEX, les contracteurs sont implémentés suivant le patron de conception « Stratégie » (cf. http://en.wikipedia.org/wiki/Strategy_pattern). Cela signifie que chaque algorithme est une classe particulière qui étend une classe générique `Contractor`, de telle sorte que les algorithmes puissent être combinés dynamiquement par simple manipulation d'objets.

Par exemple, le contracteur HC4 dont nous avons parlé plus haut est une classe `HC4` qui étend `Contractor`.

L'action associée à chaque contracteur se trouve dans la méthode `contract()`. Des exemples d'appels à cette méthode depuis différents objets ont été donnés au chapitre 1. L'intérêt d'IBEX est qu'il permet au programmeur de définir son propre contracteur et de l'utiliser ensuite directement avec le reste de l'API. Nous allons donner un premier exemple. Mentionnons toutefois auparavant un point important sur les choix d'architecture d'IBEX.

2.4.2 Un objet « poids-mouche »

Les contracteurs sont considérés dans IBEX comme des objets de type « poids-mouche » (cf. http://en.wikipedia.org/wiki/Flyweight_pattern), que l'on peut dupliquer en temps constant (indépendamment du nombre de variables, etc.). La raison principale à cela est qu'IBEX est basé sur le principe de programmation *par contracteurs* : les contracteurs sont en quelque sorte les mots d'un langage, ils doivent donc pouvoir être combinés rapidement.

L'algorithme propre à chaque contracteur ne prend aucune place, ce n'est que du code ou presque. En revanche, les domaines (c'est à dire l'espace sur lequel il travaille) est une partie qui peut être très lourde. Heureusement, un même espace est souvent partagé par plusieurs contracteurs. Par rapport à l'exemple donné sur le lien wikipedia plus haut, l'espace joue le rôle du *glyph*, le contracteur celui de caractère.

L'espace n'est donc qu'une référence vers un objet extérieur à la classe du contracteur. Il n'est pas créé par le contracteur, et il n'en fait pas la copie dans son constructeur de copie. Il se contente de transmettre la référence.

Autrement dit, le programmeur doit prendre soin d'un côté de gérer les espaces qu'ils crée et, d'un autre côté, de décider sur quel espace va travailler tel ou tel contracteur. La création/destruction de contracteurs n'impacte pas les espaces qui existent de façon indépendante.

Il y a quelques exceptions (un contracteur tel que `HC4Revise` peut construire son propre espace si on le souhaite. En revanche, il partagera bien cet espace avec tous ses clones).

2.4.3 Premier exemple

Supposons que l'on cherche les solutions dans $[0, \pi] \times [0, \pi]$ du système d'équations suivant :

$$\begin{cases} y = x \\ y = \cos(x) \end{cases}$$

mais que l'on souhaite définir nous-même, et de façon indépendante, la façon de contracter pour chaque contrainte.

Téléchargez le programme `exemple2.cpp` pour obtenir le code complet de cet exemple

Voici le code du premier contracteur :

```
class MyCtc1 : public Contractor {
public:
    // constructor. Our contractor is an operator
    // that will work on the space given in argument.
    // Only a reference to this space will be kept.
    MyCtc1(Space& space) : Operator(space) { }

    // copy constructor.
    // the space is passed by reference to the new object,
    // as it is strongly recommended in IBEX
    MyCtc1(const MyCtc1& c) : Operator(c.space), Contractor(c) { }

    // copy function (mandatory to be implemented).
    MyCtc1* copy() const { return new MyCtc1(*this); }

    void contract() {
        // [x]:= [x] inter [y]
```

```

    space.box(1) &= space.box(2);
    // [y]:=[y] inter [x]
    space.box(2) &= space.box(1);
  }
};

```

La fonction `copy()` est une conséquence du fait que cet objet est à la fois une stratégie et un poids-mouche, il doit pouvoir être copié mais sans que l'on connaisse exactement son type. Il y a donc une fonction `copy()` définie dans la classe mère `Contractor` qu'il doit implémenter. L'implémentation se résume systématiquement à appeler le constructeur par copie (ici par défaut). Elle ne présente donc pas de difficulté, ce n'est que de la syntaxe.

La fonction `contract()` applique l'algorithme suivant pour la contrainte $x = y$:

$$[x] \leftarrow [x] \cap [y]$$

$$[y] \leftarrow [y] \cap [x]$$

Le code du second contracteur est totalement similaire. Seule la méthode `contract` est légèrement différente, elle applique l'algorithme suivant :

$$[y] \leftarrow [y] \cap \sin([x])$$

$$[x] \leftarrow [x] \cap \arcsin([y])$$

```

void MyCtc2::contract() {
  // [y]:=[y] inter cos([x])
  space.box(2) &= Cos(space.box(1));
  // [x]:=[x] inter arccos([y])
  space.box(1) &= ArcCos(space.box(2));
}

```

Il suffit ensuite de définir un environnement et un espace pour déclarer x et y comme deux variables avec les domaines souhaités :

```

int main() {
  Env env;
  env.add_symbol("x");
  env.add_symbol("y");

  SpaceFactory spacef(env);
  // the domain of x will be in space.box(1)
  // the initial bounds are [0,3.14159]
  spacef.set_entity("x",IBEX_VAR, INTERVAL(0,3.14159));
  // the domain of y will be in space.box(2)

```

```

// the initial bounds are [0,3.14159]
spacef.set_entity("y",IBEX_VAR, INTERVAL(0,3.14159));

Space space(spacef);

```

On construit les deux contracteurs sur le même espace, puis on les met en séquence. L'objet `Sequence` est lui-même un contracteur.

```

// create the contractors
MyCtc1 ctc1(space);
MyCtc2 ctc2(space);

// put them in sequence
// space will also be shared by "seq"
Sequence seq=ctc1 & ctc2;

```

Enfin, on crée un paveur pour effectuer la recherche.

```

// create a round-robin heuristic
// still on the same space
RoundRobin rr(space,1e-07);

// create a paver
Paver paver(seq, rr, 1e-07);

// run the paver
paver.explore();

// plot the solution found
cout << paver.box(1,0) << endl;

```

2.5 Bissecteur

La classe `Contractor` étend elle-même une classe encore plus générique, celle d'opérateur (classe `Operator`). S'il n'est pas un contracteur, un opérateur est pour le moment un bissecteur. Mais cela n'est pas figé. Il est question, à l'avenir, de faire par exemple entrer les « chercheurs locaux » comme un troisième type d'opérateur.

Il existe donc une classe générique pour les bissecteurs, `Bissector`. Les bissecteurs possèdent exactement les mêmes concepts que les contracteurs : ce sont des objets de type *stratégie* et, structurellement, *poids-plume*.

La fonction implémentant l'action du bissecteur est tout simplement `bisect()`.

Dans le cas du contracteur, `contract()` a pour type de retour `void` car la boîte contractée est celle de l'espace du contracteur. Dans le cas du bissecteur, deux boîtes doivent être retournées. On ne peut donc plus fonctionner de la même manière. La fonction `bissect()` retourne un objet de type `Bisection`. La classe `Bisection` n'est rien d'autre qu'un couple de boîtes, `box1` et `box2`. Elle contient également la variable qui a été coupée (ce qui permet d'éviter à la retrouver en inspectant les boîtes). Notons qu'un bissecteur ne modifie jamais directement l'espace auquel il est associé. Il s'en sert que pour construire l'objet `Bisection`, il accède à l'espace en lecture seule.

2.5.1 Premier exemple

Donnons maintenant un exemple de bissecteur, en reprenant l'exemple précédent. Plutôt que de découper les domaines par une stratégie *round-robin*, supposons que l'on souhaite désormais découper systématiquement la variable x et au tiers de l'intervalle (c.a.d. que la partie gauche a un rayon deux fois plus petit que la partie droite).

```
class MyBis : public Bisector {
public:
    MyBis(Space& space) : Bisector(space) { }

    // generic copy
    MyBis* copy() const { return new MyBis(space); }

    Bisection bisect() {
        // l'indice de la variable x est 0
        return Bisection(space.box, 0, 0.33);
    }
};
```

2.6 Paveur

Le paveur (classe `Paver`) est une classe qui permet d'implémenter rapidement un algorithme de type branch & bound.

Il y a deux constructeurs pour `Paver`, l'un général et l'autre simplifié. L'interface simplifiée est celle dont on a le plus souvent besoin. Elle permet typiquement de construire un "solveur", algorithme de branch & bound dont le rôle est d'isoler des vecteurs (ou tuples), en éliminant par un contracteur les zones qui ne satisfont pas des contraintes.

2.6.1 Utilisation simplifiée

Le constructeur simplifié prend en entrée :

- un contracteur `ctc`
- un bissecteur `bsc`
- une précision ϵ (un double, par ex, $1e - 07$).

C'est cette interface-là que nous avons utilisée dans les exemples précédents (voir §2.4 ou le fichier `exemple2.cpp` directement). Le contracteur et le bissecteur peuvent travailler sur des espaces différents. Le paveur prend comme espace de référence celui du contracteur ; appelons-le `space` (si vous souhaitez faire les choses autrement, utilisez l'interface générale).

Lorsqu'on lance le paveur (via la fonction `explore()`), il faut avoir précédemment mis dans `space` les domaines initiaux des variables. La fonction `explore()` applique alors les opérations suivantes :

1. appel à `ctc.contract()`, ce qui a pour effet de contracter la boîte courante contenue dans `space`
2. synchronisation de l'espace du bissecteur avec `space`.
3. appel à `bsc.bisect()` qui produit deux boîtes.
4. Chaque boîte est à tour de rôle placée dans l'espace `space`. A chaque fois, les opérations 1-3 sont répétées récursivement.

Cet algorithme produit un arbre de recherche combinatoire. Lorsqu'une boîte est entièrement vidée par un contracteur, une exception de type `EmptyBoxException` est levée par ce contracteur. La branche courante du processus est coupée.

Lorsque sur toutes ses dimensions, le diamètre d'une boîte est plus petit que la précision ϵ , la branche de recherche est également coupée mais la boîte est conservée. Elle est une boîte d'intérêt, celle qui contient potentiellement une solution.

Pour accéder à ces boîtes une fois la résolution terminée, on peut utiliser les fonctions `nb_boxes()` et `box(i, j)` qui s'utilisent typiquement de la façon suivante :

```
// get the solutions
for (int i=0; i<paver.nb_boxes(1); i++) {
    cout << "solution #" << i << " : " << paver.box(1,i) << endl;
}
```

L'indice i correspond à la i ème boîte solution. L'indice j qui vaut systématiquement 1 est expliqué ci-dessous.

2.6.2 Utilisation générale

(A compléter)

Le déroulement du paveur est le suivant : la boîte initiale est filtrée par une sélection de contracteurs jusqu'à ce que rien ne puisse être retiré (voir la figure 2.2 détaillant

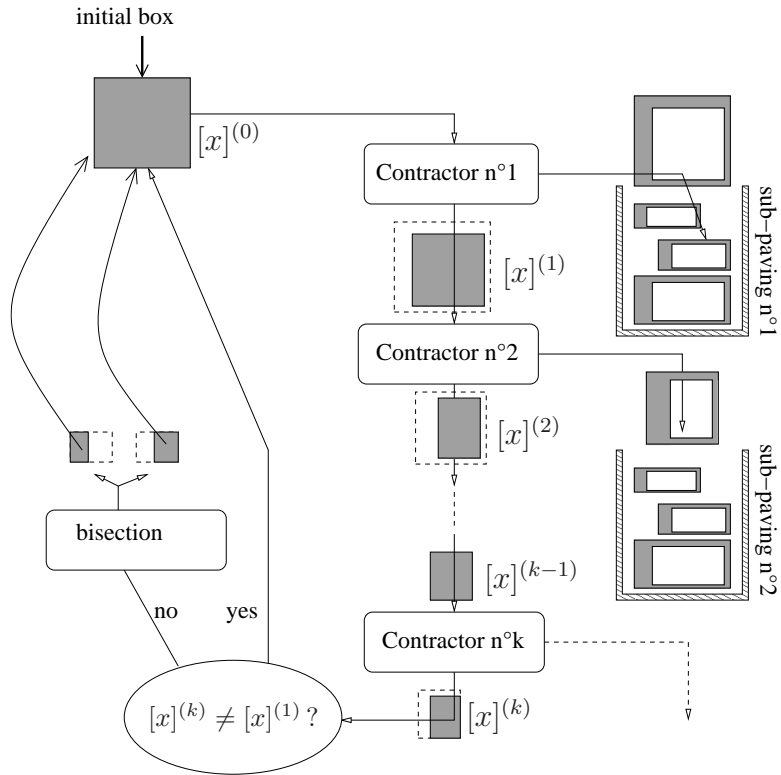


FIG. 2.2 – Algorithme de pavage.

le fonctionnement). La boîte finale est automatiquement coupée en deux puis chaque sous-boîte est de nouveau soumise aux contracteurs, et ainsi de suite. Bien évidemment, une boîte vide cesse d'être traitée.

(A compléter)

Chapitre 3

Symboles & Contraintes

3.1 Créer un nouvel environnement

L'environnement stocke toute la partie symbolique : symboles, et expressions des contraintes.

Rappelons que dans un système (comprenez : un espace), une variable x est la donnée de :

1. un numéro (par exemple, 0 si x est la première variable dans le système)
2. un domaine (un intervalle)
3. un type (IBEX_VAR ou IBEX_SYB si c'est un paramètre)
4. et un symbole associé, en l'occurrence "x" (symbole possédant lui-même une clé globale, indépendante du système)

Les systèmes, contracteurs, bissecteurs, etc. ne manipulent pas directement les symboles, ils ne manipulent que leurs numéros correspondants dans l'espace particulier auquel ses objets sont reliés.

Toute la partie symbolique est regroupée dans une seule structure, l'environnement. Pour illustrer notre propos, si l'on demande à un contracteur le symbole de la i ème variable sur laquelle il travaille, on n'a d'autre choix que d'écrire :

```
/* a function that returns the symbol of the  
 * ith variable of a contractor */  
const char* symbol_name(Contractor& ctc, int i) {  
    // get the key of the ith variable  
    int key = ctc.space.key(IBEX_VAR,i);  
    // get the environment of the contractor  
    Env& env = ctc.space.env;  
    // return the symbol corresponding to the key
```

```
return env.symbol_name(key);
}
```

Vu qu'il s'agit d'un concept "racine" dans Ibex, l'environnement ne dépend de rien d'autre. Son constructeur n'a aucun argument :

```
Env env;
```

3.2 Symboles

3.2.1 Dimensions d'un symbole

Les symboles peuvent être scalaires, vectoriels or matriciels. Ils ont donc des dimensions associées.

Cependant, ces dimensions n'interviennent qu'**au niveau des contraintes**. Pour illustrer notre propos, prenons l'exemple d'un vecteur x de dimension 2 et d'une matrice A de dimension 2×2 . Dire que la notion de vecteur/matrice est limitée aux contraintes, signifie que son rôle se limite aux points suivants :

1. les expressions des contraintes peuvent faire intervenir des opérations entre vecteurs et matrices. Ainsi, il est possible d'entrer la contrainte $A \times x = 0$ (ici le 0 est automatiquement interprété comme le vecteur $(0, 0)^T$).
2. les algorithmes d'évaluation, dérivation, etc. basés sur les expressions des contraintes utilisent le cas échéant les opérations vectorielles/matricielles intervalles. Ainsi, le contracteur `HC4revise` effectuera le calcul de $[A] \times [x]$ en appelant directement les opérations intervalles matricielles.

Mais du point de vue des autres concepts : *espace*, *contracteur*, *bissecteur*, *paveur*, il n'y a pas de notion d'entité multidimensionnelle. Tout est "mis à plat". Par exemple, le contracteur $A \times x = 0$ travaille sur un espace contenant 6 entités : $A[1][1]$, $A[1][2]$, $A[2][1]$, $A[2][2]$, $x[1]$ et $x[2]$.

S'il y a bien un symbole A dans l'environnement, il n'y a pas d'entité "A".

Pour s'en convaincre : vous pouvez écrire le petit programme suivant.

Téléchargez le programme `exemple3.cpp` pour obtenir le code complet de cet exemple

```
int main() {
    Env env;
    env.add_symbol("A", 2, 2);
    env.add_symbol("x", 2);
}
```

```

env.add_symbol("Id",2,2);

SpaceFactory spacef(env);
spacef.set_entity("A",IBEX_VAR);
spacef.set_entity("x",IBEX_VAR);
spacef.set_entity("Id",IBEX_SYB);

Space space(spacef);
cout << space << endl;
}

```

L'affichage produit est le suivant :

```

A[1][1] [          0,          0] A[1][2] [          0,          0]
A[2][1] [          0,          0] A[2][2] [          0,          0]
  x[1] [          0,          0]   x[2] [          0,          0]

```

Les variables `Id[1][1]`, ..., `Id[2][2]` n'apparaissent pas car, par défaut, l'affichage d'un espace cache les sybs. Pour changer le mode d'affichage (faire apparaître les sybs), il faut utiliser la fonction `set_output_flags` de la classe `Space`.

3.2.2 Déclaration d'un Symbole

Si aucune dimension n'est spécifiée pour un symbole, il s'agit par défaut d'un scalaire (ou symbole *zero-dimensionnel*).

Si `x` est un vecteur à 2 composantes, alors trois symboles sont valides :

- le symbole `x`, de dimension 1.
- les symboles `x[1]` et `x[2]`, scalaires.

Notez qu'un vecteur ayant 1 composante est différente d'un scalaire. Cependant, les deux types s'identifient dans la syntaxe des expressions, lorsque cela est commode. Par exemple, si `lambda` est un vecteur à 1 composante et `A` une matrice, il est légal d'écrire :

$$lambda * A$$

sachant que `lambda` est dans ce cas identifié à un scalaire, en l'occurrence `lambda[1]`.

Si `x` est une matrice 2×3 , alors 9 symboles sont valides :

- le symbole `x`, de dimension 2;
- les symboles `x[1]` et `x[2]`, de dimensions 1;
- les symboles `x[1][1]`, ..., `x[2][3]`, scalaires.

De façon similaire, une matrice unicolonne peut être identifiée à un vecteur colonne dans une expression.

Pour créer un nouveau symbole `x`, utilisez la fonction `add_symbol`. Les arguments, optionnels, donne les dimensions du symbole. Exemple :

```
env.add_symbol("x"); // creates le symbol x
env.add_symbol("y",2); // creates a vector symbol y with 2
                        components (le symbols y[1] et y[2])
env.add_symbol("z",1,2); // creates a 1x2 matrix symbol z
env.add_symbol("w",2,3,4); // creates a 3-dimensional symbol w
```

Il n'est pas possible de créer un symbole à plus de 3 dimensions.

3.2.3 Clés des symboles

Comme dit précédemment, chaque symbole possède une clé (*key*), qui l'identifie dans son environnement. Dans l'exemple précédent, si on écrit :

```
cout << env << endl;
```

La table des clés de l'environnement sera affichée :

```
Symboles :
x key=0 (0 x 0 rows x 0 cols)
y key=1 (0 x 0 rows x 2 cols)
z key=3 (0 x 1 rows x 2 cols)
w key=5 (2 x 3 rows x 4 cols)
```

Notez que la clé de *z* est 3 (et non 2) car 2 est en fait la clé de *y[2]* : En effet, puisque nous avons entré *y* comme étant un symbole à 2 composantes, *y[1]* et *y[2]* sont des symboles valides. A ce titre, ils ont eux aussi leur clés.

En réalité, l'environnement ne stocke pas *tous* les symboles possibles. Il ne stocke que les symboles dits *de base*, c'est à dire sans indexation (ce qu'on observe dans l'affichage précédent). Il réserve par contre, des clés pour les symboles indexés.

Le principe est simple : il y a autant de clés réservées par un symbole de base qu'il y a de symboles valides possible obtenus en l'indexant. Ainsi, le tableau *w* de dimension $2 \times 3 \times 4$ a réservé 24 clés. Une conséquence de cette structure est que la clé de *w[1][1][1]* est 5, comme *w[1][1]*, *w[1]* et *w* lui-même, ce qui peut paraître curieux à première vue.

Pour obtenir la clé d'un symbole, utilisez `symbol_key`. Par exemple :

```
env.symbol_key("x")
```

retournera 0. Utilisez `env.nb_keys()` pour obtenir le nombre total de clés de l'environnement.

Retenir :

exemple d'un symbole de base :	x
exemple d'un symbole indexé :	x[1]

⇒ L'environnement ne stocke que les symboles de base.

3.2.4 La structure Dim

Nous allons expliquer ici comment est représentée l'information de dimension associée aux symboles de base. Il s'agit de la structure `Dim` (dans `IbexDomain.h`).

Dans cette structure, tout symbole est considéré comme un vecteur de dimension 3, sauf que la "valeur" d'une dimension peut être 0.

La "valeur" (`value`) de la i ème dimension représente le nombre de symboles valides obtenus en indexant dans cette dimension. Les valeurs des trois dimensions sont représentés par les champs `dim1`, `dim2` et `dim3`. Si x est scalaire, `dim1=0`, `dim2=0`, `dim3=0`. En effet, `x[1]` n'est pas un symbole valide. Si x est un vecteur à deux composantes, `dim1=0`, `dim2=0`, `dim3=2` (car `x[1]` et `x[2]` sont des symboles valides mais pas `x[1][1]`). Si x est une matrice 2×3 , `dim1=0`, `dim2=2`, `dim3=3`, etc.

La "taille" (`size`) d'une dimension représente le nombre d'éléments dans cette dimension (cette fois, vu comme un tableau). La taille est simplement égale à la "valeur" si celle-ci est supérieur à 0, et 1 sinon.

Ainsi, un vecteur à une seule composante est distingué d'un scalaire : le champ `dim3` vaut 1 et 0 respectivement. Mais, dans les deux cases, `size3()` retourne bien 1.

Ces informations peuvent être retrouvées, par exemple, avec la fonction `symbol_dim`. Un exemple d'utilisation (toujours dans l'exemple précédent) est :

```
Dim d=env.symbol_dim("x");
```

3.3 Contraintes

Il existe deux façons de créer des contraintes :

- soit directement dans le code C++ en utilisant la surcharge d'opérateurs. Cette solution peut notamment être très utiles lorsqu'il s'agit de générer des contraintes dynamiquement.
- soit en passant par un fichier externe en langage `Quimper`. Cette solution est utile par exemple lorsque l'on souhaite lire un grand nombre de contraintes, obtenues via un processus externe.

3.3.1 Via la surcharge d'opérateurs

Les opérateurs `+`, `-`, `*`, `/`, ainsi que les fonctions standardes (`cos`, `sin`, etc.) peuvent être appliqués sur des symboles pour pouvoir construire des expressions, et des contraintes.

Tout d'abord, la méthode `add_symbol()` vue plus haut retourne un objet représentant le symbole de base, de la classe `Symbol`. C'est sur cet objet que s'appliqueront les opérateurs.

Plus exactement, l'objet retourné par `add_symbol()` est une référence (constante), c'est à dire un objet de type `const Symbol&`. En effet, le symbole doit posséder une unique représentation dans l'environnement. On ne gère ni sa création, ni sa destruction (qui ne se fait qu'au moment où l'environnement est lui-même détruit).

Il suffit ensuite d'appliquer les opérateurs sur cette référence. Voici un exemple :

```
Symbol& x=env.add_symbol("x");
// build a new constraint 'sin(x)+x=0'
// and add it to the environment
env.add_ctr(sin(x)+x=0);
```

Cette manière d'entrer des contraintes est utile pour générer des contraintes dynamiquement. L'objet créé dans l'environnement est de type `Constraint`. De même que pour un symbole, on ne peut que récupérer une référence constante (de type `const Constraint&`) sur cet objet, et son allocation et destruction est gérée par l'environnement lui-même. Retenir :

Ne jamais créer ou détruire soi-même les symboles, expressions ou contraintes. Utiliser simplement les fonctions `add_symbol`, `add_ctr`.

Pour récupérer l'objet, on utilise la fonction `constraint(int)`. Cette fonction attend en argument un entier qui n'est autre que le numéro de la contrainte, c.a.d., son ordre d'apparition dans l'environnement. Le numéro d'une contrainte est également retourné par la fonction `add_ctr`. Ainsi, on peut écrire par exemple :

```
int num=env.add_ctr(sin(x)+x>=0);

/* get the object that represents sin(x)+x>=0 */
const Constraint& c=env.constraint(num);
```

3.3.2 Via un fichier Quimper

Il est possible d'utiliser un fichier dans la syntaxe du langage `Quimper` pour entrer des contraintes. Un exemple a été donné au début du chapitre 1.

Le mot-clé `constraints` dans le langage `Quimper` permet d'entrer une liste de contraintes sans pour autant lui donner un nom (contrairement au mot-clé `constraint -list`, utilisé dans le langage pour créer des contracteurs).

De tels fichiers `Quimper` peuvent être chargés directement dans un objet `CSP` ou bien `System`, que nous allons détailler dans la section suivante.

Une fois un objet CSP (ou `System`) créé, on peut de nouveau retrouver la référence de type `const Constraint&`, en écrivant par exemple :

```
System sys("my_system.txt");

/* get the first constraint of the system: */
const Constraint& c=sys.ctr(0);
```

La première contrainte est celle qui apparaît en premier dans le fichier. L'environnement auquel appartient cet objet contrainte est celui créé automatiquement par le système, comme nous l'expliquons ci-dessous.

3.4 CSP & Systèmes

Dans IBEX, le CSP et le `System` ne sont que des collections de contraintes, associées à un espace (et donc un environnement). La classe `System` étend `CSP`, un système est donc un CSP particulier. Ces classes n'ont pas vraiment d'importance du point de vue opérationnel. C'est une différence assez fondamentale avec les autres logiciels de programmation par contraintes. On peut les voir comme de simples structures "préparant" la création de contracteurs. En effet, un contracteur est souvent construit à partir d'une liste de contraintes et d'un espace. C'est le cas par exemple de HC4 ou du Newton intervalle. L'idée est donc de regrouper dans un premier temps les contraintes, puis ensuite de fabriquer des contracteurs à partir de cette liste.

Toutefois, la classe `System` n'est pas *strictement* un simple container de contraintes. Il est possible, par exemple, de calculer une matrice jacobienne intervalle ou une matrice "de Hansen" à partir d'un système.

3.4.1 Création

Pour créer un système, nous avons donné comme exemple dans le chapitre introductif :

```
System sys("my_system.txt");
```

Cela suppose que le fichier `my_system.txt` ne contient que des contraintes dites *arithmétiques*. Une contrainte arithmétique est une équation ou une inéquation, c'est à dire une contrainte que l'on peut mettre sous la forme $f(x) \text{ op } 0$, où f est une fonction mathématique identifiée et op un opérateur de comparaison parmi $\{<, \leq, =, \geq, >\}$.

Une contrainte telle que `n_vector` n'est pas une contrainte arithmétique. C'est une contrainte *globale* qui n'a pas d'équivalent (simple) algébrique. Une contrainte telle que $f(x) \notin [0, 1]$, valide dans le langage `Quimper` (elle s'écrit `f(x) not-in [0,1]`),

n'est pas considérée non plus comme une contrainte arithmétique car elle contient une disjonction.

Lorsque le fichier en entrée possède des contraintes non-arithmétiques, on ne peut pas le charger dans un objet `System`, mais dans un objet `CSP`, qui est plus général. La classe `CSP` possède donc une interface moins riche : une différence importante, notamment, est qu'il n'est pas possible de calculer la jacobienne d'un `CSP` (en effet, cela n'a pas de sens pour une contrainte globale de parler de dérivée). La syntaxe de création est exactement la même :

```
CSP csp("my_system.txt");
```

Remarque 3.1 *Si le fichier `Quimper` possède des contrateurs, des listes de contracteurs ou d'autres listes de contraintes, ils seront simplement ignorés lors de la création du `CSP` ou du système. Seule la liste de contraintes suivant le mot-clé `Constraints` sera prise en compte.*

Il est possible, bien entendu, de créer un système à partir de contraintes entrées directement en C++. Il faut pour cela placer les contraintes dans un vecteur, puis de le passer en paramètre au constructeur de `System`. La seule subtilité technique est qu'il est impossible en C++ de placer des références dans un vecteur. Nous devons donc utiliser un vecteur d'adresses (de type `const Constraint*`), mais cela ne présente aucune difficulté (on récupère simplement l'adresse de l'objet "référence" via `&`). Nous poursuivons ici l'exemple de la section 3.2.1.

Téléchargez le programme `exemple3.cpp` pour obtenir le code complet de cet exemple

```
const Symbol& A=env.symbol("A");
const Symbol& x=env.symbol("x");
const Symbol& Id=env.symbol("Id");

int num;
num=env.add_ctr(A*x=0);
const Constraint& c1=env.constraint(num);

num=env.add_ctr(A*transpose(A)=Id);
const Constraint& c2=env.constraint(num);

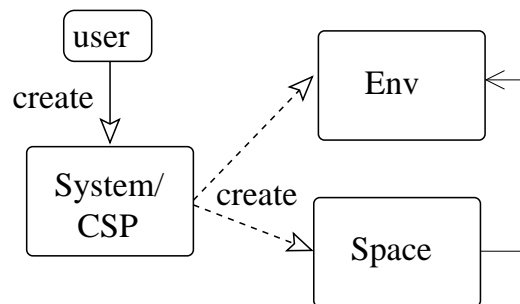
vector<const Constraint*> vec;
vec.push_back(&c1);
vec.push_back(&c2);
```

```
CSP csp(vec,space);
```

Remarque 3.2 La contrainte $A \times A^T = Id$ entrée ici permet de contraindre la matrice A (inconnue) à être une matrice orthogonale.

3.4.2 Espace et environnement d'un système

Lorsqu'un objet System ou CSP est construit, un environnement est créé et toutes les contraintes sont ajoutées dans cet environnement. Un espace est également créé, et chaque symbole devient une entité dans cet espace. L'entité peut être soit une variable (si le symbole était déclaré dans la partie `variables` du fichier `Quimper`), soit un `syb` (si le symbole était déclaré dans la partie `parameters` du fichier).



Pour accéder à l'espace : utiliser `csp.space`. Pour accéder à l'environnement : utiliser `csp.space.env`.

Ces objets peuvent évidemment être utilisés pour construire d'autres opérateurs. Voici un exemple type :

```

/* return a HC4 contractor built from
 * a system in a Quimper file */
Contractor* get_HC4(char* filename) {
    System sys(filename);
    return new HC4(sys);
}

```

Dans cet exemple, l'objet `HC4` retourné par la fonction `get_HC4` a pour espace celui du système `sys` créé dans la fonction. Mais ce système étant une variable locale, il est détruit lorsque la fonction retourne. S'il en est de même pour l'espace, le contracteur retourné pointerait vers une structure inexistante.

Il est clair que l'espace et l'environnement doivent être "persistents", c'est à dire, continuer à exister indépendamment du système qui les a créés. C'est d'ailleurs cohérent avec le fait qu'on construit normalement en premier lieu un environnement, puis un espace, et enfin un système. Dans le cas d'un fichier, tout se fait en même temps par commodité, mais c'est bien le système qui dépend d'un environnement et non le contraire. Le fait que l'on récupère, du point de vue du programmeur, un objet `System` peut induire en erreur, mais on garde bien la même hiérarchie entre ces objets.

L'espace et l'environnement se détruisent donc eux-même lorsque plus aucun opérateur (contracteur ou bissecteur) ne les référencent (à l'instar d'un ramasse-miette). L'exemple de code précédent ne pose donc pas de problème.

Lorsqu'il s'agit de créer un système (et non CSP), il faut indiquer à C++ que les contraintes manipulées sont en réalité des contraintes arithmétiques. On reprend donc le même code, à quelques "cast" près :

```
vector<const ArithConstraint*> vec;  
vec.push_back((const ArithConstraint*) &c1);  
vec.push_back((const ArithConstraint*) &c2);  
  
System system(vec,space);
```

Chapitre 4

Contracteurs

4.1 Introduction

Nous avons déjà introduit le concept de contracteur au chapitre 2. Nous reprenons ici les points principaux.

Tout contracteur dans **IBEX** est une classe qui doit étendre la classe abstraite **Contractor**. Deux fonctions doivent être implémentées :

- la fonction `contract()`, qui contient le code spécifique au contracteur pour transformer une boîte en une boîte plus petite (chaque classe est une *stratégie* particulière).
- la fonction `copy()`, qui retourne une copie de l'instance courante du contracteur (suivant le patron *poids-mouche*).

Enfin, les constructeurs doivent respecter certaines règles. L'implémentation de la fonction `copy()` et les règles que nous allons décrire pour la construction peuvent être appliquées par de simples copier-coller. Il est tout à fait possible d'ignorer les subtilités de C++ qui en sont à l'origine.

Le chapitre décrit ensuite des mécanismes qui permettent de contrôler plus finement les appels aux contracteurs. Ces contrôles seront rendus possibles grâce à la structure d'*indicateurs* (**Indicators**) et au champ appelé *seuil de contraction* (`contract_floor`). Ces moyens de contrôle ne sont a priori important que dans le cas où le contracteur est appelé depuis un paveur (c'est à dire, de façon répétitive). Il semble en effet peu utile, a priori, d'optimiser le traitement fait par un contracteur s'il n'est appelé que de façon isolé (sur une seule boîte).

4.2 Règles de construction

Tout contracteur travaille sur un espace. Cet espace peut lui être fourni directement. Si votre application concerne un ensemble fixe bien déterminé de variables, vous pouvez déclarer une variable globale de type `Space`, que vous construisez à l'initialisation de votre application, puis que vous utiliserez systématiquement pour fabriquer vos contracteurs.

Tout constructeur doit alors en premier lieu transmettre cet espace au constructeur de la classe-mère `Operator`. Voici un exemple.

```
// my unique (global) space
// that will be built in the main function
Space* space;

/* My contractor */
class MyCtc : public Contractor {
public:
    MyCtc() : Operator(*space) { }
    ...
}
```

Notez que cela impose qu'il y ait au moins un constructeur dans votre classe. Si votre application comporte plusieurs espaces, vous pouvez transmettre l'espace qui vous intéresse en argument du constructeur.

```
class MyCtc : public Contractor {
public:
    MyCtc(Space& space) : Operator(space) { }
    ...
}
```

La seconde règle concerne le constructeur de copie. Il doit également transmettre l'espace à la classe-mère `Operator`. Mais il doit aussi, cette fois, appeler le constructeur de copie de `Contractor`. La raison est que la superclasse `Contractor` possède ses propres données (en l'occurrence, un seul champ, `contract_floor`, mais il pourrait y en avoir d'autres). Ces données sont les données génériques, c'est à dire, communes à tout contracteur (c'est pour cette raison qu'elles sont "remontées" au niveau de la classe `Contractor`). Ne pas appeler le constructeur de copie de `Contractor` aurait pour effet d'appeler le constructeur par défaut. Les données en question ne seraient pas copiées, mais réinitialisées.

```
// copy constructor.
MyCtc(const MyCtc1& c) : Operator(c.space), Contractor(c) { }
```

Remarque 4.1 (Sur l'héritage d'`Operator`) *La raison pour laquelle le constructeur d'`Operator` doit être invoqué bien que `Contractor` en hérite est lié à l'héritage*

virtuel. Puisque *Contractor* hérite virtuellement d'*Operator*, chaque classe dérivée de *Contractor* possède sa propre instance *Operator* dont il a la charge de la construction (au lieu de partager l'instance de *Contractor*, comme dans le cas de l'héritage classique). Enfin, la raison pour laquelle cet héritage virtuel a été mis en place est lié à l'héritage multiple. Un contracteur peut être amené à hériter de plusieurs contracteurs, de part la nature "compositionnelle" de notre langage. Or, l'héritage virtuel est une façon propre de gérer l'héritage multiple. L'autre choix aurait été de créer uniquement une interface *Contractor*, mais les données génériques n'auraient plus trouvé leur place.

Encore une fois, ces détails techniques purement spécifiques à C++ peuvent être complètement ignorés!

La fonction `copy` ne fait alors qu'invoquer le constructeur de copie.

```
// copy function (mandatory to be implemented).
MyCtc* copy() const { return new MyCtc1(*this); }
```

La fonction `copy()` est indispensable simplement parce que pour appeler le constructeur d'une classe, il faut en connaître statiquement le type. Puisque l'on souhaite pouvoir dupliquer un contracteur sans en connaître le type, on est bien obligé d'introduire une telle fonction intermédiaire.

4.3 Seuil de contraction

Un premier paramètre de contrôle souvent important s'appelle le *seuil de contraction*. Dans ce qui suit, nous allons par simplicité illustrer l'intérêt de ce seuil en se basant sur les appels à la fonction `contract()` que fait un paveur. Cependant, ce seuil impacte directement le résultat d'un contracteur (*oui* : il y a eu contraction, *non* : la boîte n'a pas changée). Or, le besoin de contrôler (ou même simplement connaître) ce résultat n'est pas limité au paveur. Il peut intervenir notamment dans la programmation d'un contracteur basé sur un ensemble de sous-contracteurs, comme c'est le cas pour la propagation (**Propagation**) ou le shaving (**Shaving**). Le programmeur peut donc être amené lui-même à devoir "jouer" avec ces seuils pour développer un algorithme correct à partir de sous-contracteurs.

Le paveur appelle en boucle tous ses contracteurs sur la boîte courante jusqu'à ce qu'il y ait un point fixe (c'est à dire, jusqu'à ce que plus aucun contracteur ne puisse réduire les domaines des variables). Voir à ce propos §2.6.

Il est clair qu'en domaines continus, le point fixe peut être long à atteindre : imaginez simplement que le cas extrême où un contracteur n'ôte qu'un seul flottant du domaine d'une variable, il y a bien eu contraction. Cela signifie que l'on peut perdre beaucoup de temps à raffiner les domaines avant d'entrer dans la phase de bisection (point de choix). L'efficacité du paveur repose sur un équilibre entre

contraction et bisection. La contraction permet de réduire les domaines en évitant de faire des choix mais elle consomme aussi du temps. Globalement, il peut être plus avantageux de limiter la contraction est de séparer davantage le problème en sous-problèmes.

Il n'y a pas de règle pour obtenir le bon compromis, qui dépend évidemment du problème, des contracteurs utilisés, de la stratégie de bisection, etc. C'est même, on peut dire, un des grand dilemmes de la recherche combinatoire en général. Ce qui est clair en revanche, c'est que laisser les contracteurs travailler jusqu'aux flottants est un très mauvais choix en pratique. Il faut donc pouvoir dire à un contracteur de ne *pas* faire de contraction si celle-ci n'est pas suffisamment conséquente, c'est à dire si la taille de l'intervalle retiré est en dessous d'un certain seuil.

C'est ce seuil que représente le champ `contract_floor`. C'est un seuil relatif, il est fixé par défaut à 2% de la taille de l'intervalle réduit.

Ainsi, lorsqu'un contracteur C contracte une boîte $[x]$ en $[y] = C([x])$, si pour toute variable i on a

$$\frac{\text{rad}([x]_i \setminus [y]_i)}{\text{rad}[x]_i} < \text{contract_floor} \quad (4.1)$$

alors, la contraction est annulée (sorte de *rollback*) et la boîte $[x]$ laissée dans son état initial.

Voici un exemple qui illustre l'effet de ce seuil.

Téléchargez le programme `exemple4.cpp` pour obtenir le code complet de cet exemple

Dans cet exemple, deux contraintes (`c1` et `c2`) ont été créées. On crée à partir de ces contraintes deux contracteurs (`ctc1` et `ctc2`) que l'on donne (avec le contracteur de précision) à un paveur. On fixe également leur seuil de contraction à 2% (la valeur par défaut)

```
// create a contractor for constraint y=x
HC4Revise ctc1(c1,space);
// create a contractor for constraint y=sin(x)
HC4Revise ctc2(c2,space);
// create the precision contractor
Precision ctc3(space,1e-05);

ctc1.contract_floor=0.02;
ctc2.contract_floor=0.02;
```

Le paveur donne le résultat suivant :

```
[...] 8813 boxes created.
total time   : 0.264016s
```

En mettant 0.01 au lieu de 0.02, on obtient :

```
[...] 7725 boxes created.  
total time   : 0.232014s
```

On peut voir que le nombre de points de choix (représentés par le nombre de boîtes totales) est passé de 8813 à 7725. On a donc bien filtré davantage. Il y a eu également un gain de temps.

En revanche, si l'on fait passer le seuil à 0.002, on obtient :

```
[...] 4521 boxes created.  
total time   : 0.336021s
```

Le filtrage a permis cette fois de diviser quasiment par deux le nombre de boîtes à traiter mais le temps total est passé de 0.26 à 0.33...

Pour ce problème particulier, le bon compromis semble se situer autour de 0.1.

4.3.1 La valeur spéciale 0

Il faut avoir conscience que pour pouvoir faire le calcul (4.1), cela nécessite de stocker une copie du domaine initial, c'est à dire la boîte courante de l'espace, objet *lourd*.

En présence d'un grand nombre de variables, ces copies peuvent être désastreuses.

Un moyen de désactiver ce calcul consiste simplement à fixer le seuil à 0 (ou une valeur négative). C'est ce qui est fait par exemple dans l'algorithme de **Propagation**. Tous les sous-contracteurs ont leur seuil fixé à 0 pour éviter de copier l'ensemble des domaines à chaque étape intermédiaire.

Il ne faut évidemment pas donner au *paveur* un contracteur avec le seuil à 0. Le paveur se chargeant simplement d'appeler en boucle ses contracteurs tant qu'il y a eu contraction, cela entraînerait automatiquement¹ une boucle infinie.

Lorsque ce seuil est fixé à 0, il faut donc se méfier des risques de boucler infiniment et gérer soi-même l'arrêt du point fixe. Cette gestion peut d'ailleurs se résumer à une stratégie très simple. C'est le cas par exemple du contracteur **Sequence**, qui se limite à appeler une fois et une seule chacun de ses sous-contracteurs. Le seuil est bien mis à 0 pour éviter des copies inutiles, et ce, sans aucun risque.

¹A l'exception d'un seul cas : celui où l'un des contracteurs provoquerait une **EmptyBoxException** avec la toute première boîte.

4.4 Indicateurs

Il existe un autre type de paramètres pour permettre aux algorithmes (surtout de propagation de contraintes) d'être efficaces. Ces paramètres rejoignent la notion d'*événement* que l'on retrouve dans les solveurs de variables discrètes, mais sous une forme pour le moment beaucoup plus rudimentaire.

Le but de ces paramètres est de donner un minimum d'information à un contracteur pendant la recherche sur le "contexte" dans lequel il est appelé. Par exemple, on peut informer un contracteur que seule la contraction sur une variable particulière est vraiment attendue (les contractions sur les autres variables étant moins intéressantes voire superflues).

Vu que la complexité en temps d'un contracteur dépend très souvent du nombre de variables qu'il a à traiter, un gain de performance substantiel peut être obtenu.

Nous pouvons également informer un contracteur que, depuis la dernière fois où il a été invoqué, seule une variable a eu son domaine modifié (comprenez, entre-temps, par un autre contracteur). De nouveau, si le contracteur est capable de fonctionner de façon incrémentale, cela entraînera un gain de performance.

Deux indicateurs ont été intégrés dans IBEX jusqu'ici ; ils correspondent justement aux exemples donnés ci-dessus. Le premier s'appelle `scope` et indique sur quelles variables doit porter la contraction. Il s'agit en fait d'un simple entier (integer) dont la valeur indique soit un numéro de variable, soit *toutes les variables*, soit *aucune variable* (ces deux dernières étant des valeurs constantes particulières nommées respectivement `ALL_VARS` et `NO_VAR`).

L'autre s'appelle `impact` et indique de la même façon les variables impactées depuis le dernier appel. De même, il n'est possible actuellement de spécifier qu'une seule variable ; dans les autres cas, les valeurs `ALL_VARS` ou `NO_VAR` peuvent être utilisées.

Remarque 4.2 (Pourquoi une seule variable ?) *Il peut paraître restrictif de ne pouvoir spécifier plus d'une variable. Il est possible que cela évolue à l'avenir, mais voilà ce qui justifie au moins pour le `scope` cette restriction : S'il fallait pouvoir donner à un contracteur un ensemble de variables, d'une part cela compliquerait considérablement le code des contracteurs (et donc, probablement, ralentirait leur exécution) ; d'autre part la structure d'ensemble de variables deviendrait très proche de la notion d'espace qui, rappelons-le, indique précisément à un contracteur sur quelles variables il doit travailler ! Autrement dit, le paramétrage dynamique que représentent les indicateurs tendrait à remplacer le paramétrage statique que sont les espaces, en perdant leur avantage essentiel : celui de pouvoir programmer un contracteur sans se préoccuper justement des variables sur lesquelles il sera instancié.*

Il est toujours possible, dynamiquement, de reconstruire un contracteur en lui donnant un espace différent à chaque fois.

L'usage des indicateurs n'est soumise qu'à une seule règle fondamentale : ils peuvent être totalement ignorés par le contracteur sans que cela compromette la sémantique de celui qu'il l'appelle (par exemple, le fait qu'aucune solutions ne soit perdue!). D'où le nom d'*indicateur*, puisqu'ils ne jouent qu'un rôle *indicatif*.

Ainsi, lorsqu'une personne développe un contracteur, elle n'est pas tenue de se soucier des indications que l'on peut lui transmettre ; et ce contracteur doit pouvoir être utilisé n'importe où, y compris à l'intérieur d'un autre contracteur basé sur ces indicateurs (typiquement, la propagation). La conséquence ne peut être qu'une perte d'efficacité.

On n'est pas non plus obligé de transmettre des indicateurs à un contracteur.

Les indicateurs sont tous regroupés dans une unique classe `Indicators` qui est une classe imbriquée dans `Contractor`. Elle possède deux champs qui sont donc `scope` et `impact`.

Pour transmettre des indicateurs à un contracteur, il suffit d'utiliser une variante de la fonction `contract`, qui prend en argument une référence de type `Indicators`. Cette fonction est implémentée au niveau de `Contractor`, elle est donc systématiquement accessible (modulo un *cast*, cf. l'exemple ci-dessous).

4.4.1 Exemple illustratif

Téléchargez le programme `exemple5.cpp` pour obtenir le code complet de cet exemple

Quatre variables sont créées, x_1, x_2, y_1 , et y_2 ainsi que deux contraintes indépendantes : $x_1 = x_2$ et $y_1 = y_2$. Les domaines de x_1 et y_1 sont $[-1, 0]$, ceux de x_2 et y_2 sont $[0, 1]$. L'idée est de pouvoir observer qu'une contraction a eu lieu pour la 1ère contrainte lorsque $[x_1] = [x_2] = [0, 0]$, et de même pour la 2nde contrainte lorsque $[y_1] = [y_2] = [0, 0]$.

On construit donc deux contracteurs de base à partir de ces contraintes que l'on place dans un vecteur `vec`. L'exemple ne possède qu'un unique espace `space` partagé par tous les contracteurs.

A partir de là, on construit un objet de type `Propagation`. Pour donner une idée, le contracteur `Propagation` a pour but de lancer une liste de sous-contracteurs jusqu'à l'obtention d'un point fixe, et ce, de façon plus efficace qu'en procédant par une simple boucle. Ce contracteur est capable de prendre en compte l'indicateur *impact*, dans son mode incrémental.

L'activation du mode incrémental ce fait via le 4ème argument (optionnel) du constructeur. Cela oblige à spécifier le 3ème argument, en l'occurrence le seuil de contraction qui permet, en interne à la propagation, de gérer l'arrêt du point fixe (un chapitre futur sur les contracteurs existants présentera plus en détail ce contrac-

teur).

Ici, cet argument ne nous intéresse pas, on utilise donc la valeur par défaut fournie dans la classe (la constante `Propagation::default_ratio`). On construit donc cet objet `propagation` en lui fournissant le vecteur `vec`, le ratio par défaut et la valeur `true` pour activer le mode incrémental.

```
Propagation propag(ctc, space, Propagation::default_ratio, true);
```

L'idée ensuite est d'observer le résultat de `propag`, en lui annonçant que seule la variable x_1 a été impactée depuis la dernière fois où il a été exécuté. Dès lors, y_1 et y_2 , indépendants de x_1 , n'ont pas besoin d'être traités à nouveau.

```
// print the initial box
cout << "before: " << space << endl;

// crete a new structure of indicators
// by default every field is "ALL_VARS"
Contractor::Indicators indic;

// set the impact to the variable x1
indic.impact = 0; // 0 is the index of x1 in space
```

Pour une raison qui m'est encore un peu obscure, C++ interdit d'invoquer directement la fonction `contract` en écrivant :

```
propag.contract(indic);
```

bien que celle-ci soit héritée de la classe mère `Contractor`. Il faut donc passer par un `cast` (qui me semble inutile) :

```
// call the contractor with the indicators
((Contractor&) propag).contract(indic);

// print the result
cout << "after: " << space << endl;
```

On observe alors le résultat suivant qui montre bien que les variables y_1 et y_2 n'ont pas été touchées.

```
before:  x1 [  -1,    0]    x2 [   0,    1]
         y1 [  -1,    0]    y2 [   0,    1]

after:   x1 [   0,    0]    x2 [  -0,    0]
         y1 [  -1,    0]    y2 [   0,    1]
```

4.4.2 Accès aux indicateurs

Dans le code de votre contracteur, vous pouvez accéder aux indicateurs via la fonction `current_indic()`, définie dans `Contractor`.

Cette fonction tout d'abord retournera `null` si aucun indicateur n'a été transmis.

Si le résultat n'est pas nul, il s'agit simplement d'un pointeur vers la structure `Indicators` transmise.

4.5 Adjacence

En programmation par contraintes, on considère très souvent dans les algorithmes de propagation la notion d'*adjacence* : une variable x et une contrainte c sont adjacentes si la contrainte c implique x dans son expression. C'est vérifié en particulier du point de vue numérique lorsque la dérivée de c par rapport à x n'est pas partout nulle.

La réciproque en revanche n'est pas toujours vraie, ainsi la contrainte $x - x = 0$ bien que de dérivée partout nulle implique bien la variable x du point de vue "contraintes", car son expression symbolique fait effectivement apparaître le symbole x .

Pour savoir si une contrainte c implique x , on doit utiliser le champ `adj` de la classe `Contrainte`. Ce champ est une simple table de hachage (`hash_map`) qui associe à la clé de chaque symbole le nombre d'occurrence de ce symbole dans la contrainte (se référer à la documentation de la STL pour l'utilisation du type `hash_map`, par exemple sur www.sgi.com/tech/stl/). Voici un exemple :

Téléchargez le programme `exemple6.cpp` pour obtenir le code complet de cet exemple

```
void func(const Constraint& c, const Symbol& s) {
    if (c.adj.find(s.key)==c.adj.end())
        cout << "The constraint does not involve " << s.name << endl;
    else
        cout << "The constraint has " << (c.adj.find(s.key))->second <<
            " occurrences of " << s.name << endl;
}
```

Cette fonction affiche un message indiquant si une contrainte c implique un symbole s , et le cas échéant le nombre de fois.

Lorsqu'on appelle cette fonction avec comme premier argument la contrainte $x * x = y$, et comme second argument successivement les symboles x et z , l'affichage produit est bien :

```
The constraint has 2 occurrences of x
The constraint does not involve z
```

4.5.1 Généralisation

En programmation par contracteurs, la notion est légèrement généralisée. On dit qu'un contracteur implique une entité si son action (sa façon de réduire les domaines) peut potentiellement dépendre du domaine de cette entité².

Il y a une première différence importante, c'est le *potentiellement*. Autrement dit, si une contrainte implique x , on a un résultat sûr. Si un contracteur implique x , cela reste indicatif. En revanche, si un contracteur n'implique pas x , il est effectivement garanti que deux appels à ce contracteur donneront un résultat identique, si seul le domaine de x change.

On retrouve bien entendu ici une notion proche des indicateurs, sauf qu'ici, c'est le contracteur qui nous donne une information (et non le contraire). Tout contracteur, par défaut, implique toutes les entités.

Il est défini dans la classe `Contractor` une fonction virtuelle `involves` qui prend en argument la clé du symbole d'une entité x et qui retourne `false` seulement si le contracteur n'implique pas x .

Dire que par défaut un contracteur implique toutes les entités signifie que cette fonction est déjà implémentée dans la classe `Contractor`, et qu'elle ne fait simplement que retourner `true`. Libre au programmeur ensuite de réimplémenter cette méthode s'il le souhaite.

Il faut savoir qu'un contracteur tel que la `Propagation` n'est efficace que si les sous-contracteurs qu'on lui passe en argument sont capables de "dire" quelles sont les variables qu'ils n'impliquent pas (c'est à dire, auxquelles ils ne sont pas "sensibles"). Ainsi, les appels aux contracteurs pourront se faire de façon fine, en évitant de réveiller un contracteur pour une variable qui ne provoquera aucun effet.

4.5.2 La classe `Adjacency`

Il est clair que les accès à une table de hachage (cf. par exemple le code de `example6.cpp`) ne sont pas rapides. En pratique, un algorithme de propagation cherche en permanence à savoir quel contracteur implique quelle variable. Il a été introduit dans IBEX une classe appelée `Adjacency` dont le rôle est de construire une matrice d'adjacence entre une liste de contracteurs et un espace "commun", où chaque accès est en $O(1)$.

²Attention, l'action n'est pas forcément une réduction du domaine de l'entité elle-même. Ainsi, un contracteur peut impliquer un syb...

Son inconvénient, bien entendu, est son coût de création qui est en $O(n \times m)$ (en temps et place), où n est le nombre d'entités dans l'espace et m le nombre de contracteurs dans la liste.

Elle s'utilise ensuite de façon extrêmement simple, voici un exemple.

Téléchargez le programme `exemple7.cpp` pour obtenir le code complet de cet exemple

Dans cet exemple, nous avons deux contracteurs placés dans un vecteur `vec` : le premier pour la contrainte $x * x = y$, le second pour $y - z = 0$. Dans l'espace `space`, x et y sont des variables, z un syb.

```
ContractorList cl(vec);
Adjacency adj(cl, space);

cout << "the first contractor involves " << adj.ctr_nb_vars(0) <<
      " variables, which are:\n";
for (int i=0; i<adj.ctr_nb_vars(0); i++)
    cout << " variable #" << adj.ctr_ith_var(0,i) << endl;

cout << "the first syb (z) is involved in " << adj.syb_nb_ctrs(0)
      << " contractors, which are:\n";
for (int i=0; i<adj.syb_nb_ctrs(0); i++)
    cout << " contractor #" << adj.syb_ith_ctr(0,i) << endl;
```

L'affichage produit :

```
the first contractor involves 2 variables, which are:
  variable #0
  variable #1
the first syb (z) is involved in 1 contractors, which are:
  contractor #1
```


Chapitre 5

Bissecteurs

Chapitre 6

Paveurs