Ircam documentation

- Research reports
- Musical works
- Software

OpenMusic

Situation

version 3

Troisième édition, avril 1999

Copyright © 1999 Ircam. Tous droits réservés.
Ce manuel ne doit pas être copié, ni en entier ni partiellement,
sans le consentement écrit de l'Ircam.

Ce manuel a été écrit par Antoine Bonnet et Camilo Rueda, et
produit sous la responsabilité éditoriale de
Marc Battier - Département de la Valorisation, Ircam.

La bibliothèque Situation a été conçue et programmée par
Antoine Bonnet et Camilo Rueda.
Frank Valencia a collaboré pour l'écriture du code.

OpenMusic a été conçu et programmé par Gérard Assayag et Carlos Agon.
Cette documentation correspond à la version 3 de Situation
et à la version 2.01 et supérieure de OpenMusic.

Apple Macintosh est une marque déposée de Apple Computer, Inc.
OpenMusic est une marque déposée de l'Ircam.

Première édition de la version 3.0, avril 1999

Ircam
1, place Igor-Stravinsky
75004 Paris
Tel. 01 44 78 49 62
Fax 01 44 78 15 40
E-mail ircam-doc@ircam.fr

Groupes d'utilisateurs IRCAM

L'utilisation de ce programme et de sa documentation est strictement réservé aux membres des groupes d'utilisateurs de logiciels Ircam. Pour tout renseignement supplémentaire, contactez :

Département de la Valorisation
IRCAM
Place Stravinsky
Paris
France

Courrier électronique : bousac@ircam.fr

Veuillez faire parvenir tout commentaire ou suggestion à :

Courrier électronique : ircam-doc@ircam.fr

Table du contenu

1. Principe de Situation : contraintes et instanciation d'objets musicaux 5
2. Situation, version 3.0. 7
3. La bibliothèque Situation : référence 9
3.1. solvers (moteurs de résolution) 9
3.1.1. csolver (constraints solver = moteur de résolution de contraintes) 9
1° insert : syntaxe générale 10
2° insert : syntaxe des voix 18
3° insert : évaluation, gestion, édition 24
3.1.2. wsolver (weak solver = moteur de résolution de contraintes faibles) 29
3.1.3. add-csolver (additional constraints solver = moteur de résolution de contraintes additionnel) 30
3.2. standard constraints - (contraintes standards) 30
3.2.1. Distances control (contrôle des distances) 31
4° insert : syntaxe des modules _filt 32
5° insert : syntaxe des modules _rnw 37
6° insert : syntaxes complémentaires des modules _filt 41
3.2.2. Points control (contrôle des points) 44
3.2.3. profiles control (contrôle des profils) 48
3.3. user-constraints (contraintes de l'utilisateur) 52
3.3.1. user-cnstr (user defined constraints = contraintes définie par l'utilisateur) 52
7° insert : Explications complémentaires sur a, f, s et i. 53
3.3.2. prev-intances (previous instances = cas précédents) 55
3.3.3. wprev-intances (wprevious instances = cas précédents avec wsolver) 56
3.4. generic problem (problème générique) 56
3.4.1. variable-domains (= domaines des variables) 56
3.4.2. generic-cnstr (= contrainte générique) 56
3.4.3. done-instances (= cas terminés) 57
3.4.4. current-variable (= variable courante) 57
3.5. utilities (modules utilitaires) 57
3.5.1. ch-sol (chords solution = solution d'accords) 57
3.5.2. rtm-sol (rhythmic solution = solution rythmique) 58
3.5.3. part-sol (partial solution = solution partielle) 58
3.5.4. default-fill (default filling = remplissage par défaut) 58
3.5.5. bpf-ambdef (break point function ambitus definition = courbes de définition d'ambitus) 58
3.5.6. all-permutations (= toutes permutations) 59
3.5.7. combinations (= combinaisons) 59
3.5.8. comb-with-reps (combinations with repetition = combinaisons avec répétitions) 59
3.5.9. all-replacements (= tous remplacements) 59

4. Menu de Situation 60
4. Tutoriel 62
4.1. Tutoriel Sit 1 62
4.2. Tutoriel Sit 2 62
4.3. Tutoriel Sit 3 64
4.4. Tutoriel Sit 4 65
4.5. Tutoriel Sit 5 66
4.6. Tutoriel Sit 6 67
4.7. Tutoriel Sit 7 68
4.8. Tutoriel Sit 8 70
Bibliographie 109
Index 110

1. Principe de Situation : contraintes et instanciation d’objets musicaux

Situation est un moteur de satisfaction de contraintes écrit en Common Lisp-CLOS et muni d’un interface graphique en OpenMusic adapté à la construction de problèmes harmoniques et rythmiques.

Situation est particulièrement pertinent pour la génération des objets musicaux lorsque :

- chaque objet dans une séquence est une entité décrite par un ensemble donné de propriétés reliant ses éléments ;
- des objets dans des positions sélectionnées de la séquence doivent satisfaire un certain nombre de contraintes.

Situation permet ainsi la construction de séquences musicales à partir de la description de leurs propriétés - contrairement à l’approche fonctionnelle qui nécessite la spécification précise de l’algorithme de construction.

> La construction et la manipulation d’objets musicaux se trouvent au centre de l’activité compositionnelle. Ces objets se conçoivent généralement en termes de relations entre leurs composantes. Dans les cas les plus simples, la construction d’objets musicaux se fait de façon convenable au moyen de l’élaboration d’un algorithme spécifique, c’est-à-dire qui sache placer chaque composante selon la relation à satisfaire.

Quand la relation est complexe ou quand un nombre important de relations doit être satisfait, la définition de l’algorithme adéquat est lourde et difficile. La meilleure solution est alors de décrire les relations souhaitées afin de laisser la machine construire le bon objet. Les systèmes de résolution de contraintes, tels que Situation, ont cette approche.

Situation est conçu pour un utilisateur non expérimenté en informatique (si l’on fait exception des modules de contraintes non standards, qui sont expérimentaux et réclament une connaissance de la programmation en Common Lisp). L’inconvénient de cette exigence d’accessibilité est le temps parfois très long nécessaire à l’obtention d’un bon résultat. Pour un environnement à vocation interactive, c’est un problème à prendre en considération.

Par souci d’efficacité, Situation a donc été optimisé en fonction des relations harmoniques et rythmiques les plus probables compte tenu du parti pris de départ de placer la notion d’objet au centre du projet. La possibilité de paramétrer les contraintes avec l’interface OpenMusic en témoigne.

Naturellement, les relations y figurant ne sont pas les seules possibles : leur présence est plus une question d'efficacité que de capacité.

Très généralement, on peut envisager le processus de résolution d'un problème de satisfaction de contraintes comme celui des raffinements successifs des solutions partielles. Pour trouver, par exemple, une séquence harmonique particulière, on part d'une « solution » comportant toutes les séquences d'accords possibles et on filtre celles-ci par des contraintes permettant de sélectionner autant de sous-ensembles. Une solution valide est ainsi l'intersection de tous les sous-ensembles.

L'utilisateur doit donc tenir compte du fait que si l'ensemble initial est plus restreint, le processus de résolution est plus efficace. Trouver par exemple des séquences particulières d'accords de sept hauteurs est beaucoup plus efficace si l'on impose un certain registre aux accords que si l'on considère au départ tous les accords possibles. Cet ensemble initial est appelé *domaine* dans le jargon informatique.

2. Situation, version 3.0.

À la différence des versions précédentes, qui restaient optimisées pour des problèmes particuliers (harmoniques notamment), Situation 3.0 se présente sous la forme d'un logiciel de résolution de contraintes général : il permet de résoudre toutes sortes de problèmes, musicaux ou non, pourvu que leurs objets soient appréhendables en termes de points et de distances, et qu'ils évoluent dans des domaines finis.

Cette généralisation a nécessité le changement de nom de tous les modules. L'inconvénient que représente bien sûr ce changement – du moins pour l'utilisateur des versions antérieures - est largement compensé par la rationalisation qu'il a permise. Désormais, tous les noms de modules ne font plus référence qu'aux notions de base : objet, point ou distance (interne ou externe).

Ainsi, dans **csolver** :

- la notion d'accord est généralisée par **n-obj** (nombre d'objets),
- la notion d'ambitus par **p-pts** (possible points = ensemble des points possibles),
- la notion de densité par **n-pts** (nombre de points par objet),
- la notion d'intervalle verticale par **i-dst** (internal distance = distance interne),
- la notion d'intervalle horizontale par **x-dst** (external distance = distance externe).

Pour que les choses soient bien claires dès l'abord, on appellera respectivement dans un problème harmonique ou rythmique :

- objet = accord harmonique ou plan rythmique,
- point = hauteur ou point d'impact,
- distance interne = intervalle vertical dans un accord ou durée entre deux points d'impact dans un plan rythmique,
- distance externe = intervalle horizontal entre accords différents ou durée entre deux points d'impact de plans rythmiques différents.

Quant aux **standard constraints**, elles ne font plus référence, elles aussi, qu'aux notions de base, en liaison avec l'un des trois types de contrainte standard :

- les contraintes **–filt** (filter = filtre) qui permettent de préciser des configurations particulières de points et de distances,
- les contraintes **–rnw** (renewhal = renouvellement) qui permettent de définir des seuils de répétition de points et de distances
- les contraintes **–prof** (profil) qui permettent de dessiner des profils de points et de distances.

Enfin, Situation 3.0 comporte, à titre expérimental, un moteur de résolution de contraintes « faibles » : **Wsolver** (weak solver).

3. La bibliothèque Situation : référence

La bibliothèque Situation est organisée dans un menu hiérarchique comprenant cinq sous-menus : **solvers**, **standard constraints**, **user constraints**, **generic problem** et **utilities**.

3.1. solvers (moteurs de résolution)

Le menu **solvers** contient les modules :

- **csolver** (constraints solver = moteur de résolution de contraintes)
- **wsolver** (weak solver = moteur de résolution de contraintes faibles)
- **add-csolver** (additional constraints solver = moteur de résolution de contraintes additionnel)

3.1.1. csolver ***(constraints solver = moteur de résolution de contraintes)***

[syntaxe : n-obj, p-pts, n-pts, i-dst, x-dst, cnstr
et optionnellement : ***data***, ***x-sol***, ***n-sol, merge***]

Le module **csolver** est le module central de Situation : contenant le moteur du logiciel, il permet de définir ou de recevoir un domaine et de le gérer en fonction des contraintes standards ou personnalisées qu'il accueille.
csolver permet de définir une séquence d'accords ou de durées à partir de cinq paramètres élémentaires correspondant aux cinq premières entrées standards du module :

- **n-obj** (number of objects = nombre d'objets)
- **p-pts** (possible points = points possibles)
- **n-pts** (number of points = nombre de points)
- **i-dst** (internal distances = distances internes)
- **cnstr** (constraints = contraintes) permet de recevoir les modules de contraintes du sous-menu « standard constraints » et « user-constraints »
- **x-dst** (external distances = distances externes) ;

Laissons de côté pour le moment les entrées optionnelles.

3.1.1.1. n-obj (number of objects = nombre d'objets)

Cette entrée permet de définir le nombre d'objets du problème à calculer : nombre d'accords pour un problème harmonique, nombre de plans (ou de voix) pour un problème rythmique.
Il suffit d'y taper un nombre entier.
On notera que, pour aider l'utilisateur, toutes les fenêtres des modules de Situation contiennent un nombre ou une formule par défaut (par exemple : 4 pour la fenêtre **n-obj** de **csolver**).

3.1.1.2. p-pts (possible points = points possibles)

Cette entrée permet de définir l'ensemble des points dans lequel s'inscriront les objets. Pour un problème harmonique, il s'agira de l'ambitus dans lequel s'inscriront les accords ; pour un problème rythmique, de la durée globale dans laquelle s'inscriront les plans rythmiques.
Il faut ici s'arrêter pour préciser la syntaxe générale des modules de Situation, syntaxe à laquelle n'échappent seulement que quelques entrées élémentaires des **solvers** : **n-obj**, **x-sol**, **n-sol** (cf. **3.1.1.1.n-obj**).

1° insert : syntaxe générale

(exemplifiée par **p-pts)**

- Tout d'abord - et avant d'en venir à l'utilisation d'expressions courantes telles que **not**, **or** et **and**, ***** et **?** ou d'expressions spécifiques à Situation tels que **l**, **u**, **interp**, **step**, **f**, **x**, **t**, **app, ints** dont on verra l'utilisation au fur et à mesure des modules qui les emploient - il faut savoir que la syntaxe de Situation combine au moyen d'un jeu de parenthèses :

- des nombres entiers pour les objets ordonnés d'un problème : 0 = le premier accord pour un problème harmonique, 1 = le deuxième plan rythmique pour un problème rythmique etc... ;
- des nombres entiers pour la dénomination de valeurs évaluées en fonction des modules utilisées : par exemple pour l'entrée **n-pts** de **csolver** : 2 = accord(s) de 2 hauteurs pour

un problème harmonique (2 serait donc ici la densité), 3 = plan(s) rythmique(s) de 3 durées pour un problème rythmique (3 serait donc ici le nombre d'impacts) ;

- des nombres entiers pour la dénomination des intervalles (évalués en nombre d'unités) : 0 = unisson (harmonique ou rythmique), 1 = seconde mineure pour un problème harmonique utilisant le demi-ton comme unité, 2 = croche pour un problème rythmique utilisant la double-croche comme unité etc…
- des nombres entiers pour la dénomination des points (évalués en nombre d'unités) : pour un problème harmonique utilisant le demi-ton comme unité, le code MIDI (60 = do3, 61 = réb3 etc...) ; pour un problème rythmique, le nombre d'unités séparant le début du plan du point d'impact de la durée ou des durées à nommer (si, par exemple, un plan utilise la double-croche de quintolet comme unité, les durées (les points) situées une noire et une blanche après le début du plan se nommeront respectivement 5 et 10, c'est-à-dire 5 et 10 doubles-croches de quintolet à partir du début (rappelons que l'on commence toujours à 0 : 0 1 2 3 4 **5** 6 7 8 9 **10**)

Par convention, on commence toujours par les objets, les chiffres venant ensuite en arguments étant interprétés en fonction des modules concernés.

Si par exemple on écrit dans **p-pts** :

(0 3 7 (48_72))

cela signifie, pour un problème harmonique, que le premier, le quatrième et le huitième accord de la séquence (cf. **0 3 7**) s'inscriront obligatoirement dans un ambitus compris entre le do2 et le do4 (cf. **48 72**).

<u>Attention</u> : Pour les problèmes rythmiques, conformément à l'usage, les unités s'expriment ainsi : 1 pour la ronde, 1/2 pour la blanche etc… et par conséquent 1/12 pour la croche de triolet ou 1/28 pour la double-croche de septolet etc…

L'ordinateur interprétant ces données comme des fractions (qu'il effectue) et non des symboles, il est nécessaire d'indiquer à la machine, par **app**, le degré d'approximation souhaité au début de la syntaxe (par exemple **app 1/28**).

- Notons maintenant que Situation accepte la syntaxe d'expression courante **:**

- « ***** » pour « multiplier »
- « **_** » pour « tout élément compris entre »
- « **s2** » pour « tous les deux éléments », « **s3** » pour « tous les trois éléments etc...

Ainsi, dans **p-pts** :

(0_7 (48_72) 8_13s2 (61_83))

signifie, pour un problème harmonique, que les 8 premiers accords de la séquence (cf. **0_7**) seront compris entre le do2 et le do4 (cf. **48_72**) , et un sur deux (cf. **s2**) des 6 suivants (cf. **8_13**) entre le réb3 et le si4 (cf. **61_83**).

Il découle de ce qui a été dit des fractions (que l'ordinateur effectue), que lorsque les points possibles (**p-pts**) d'un problème rythmique sont exprimés par une fourchette (ce qui est évidemment presque toujours le cas), il est nécessaire, même lorsque aucun pas n'est sauté, d'utiliser la syntaxe **s** (« step » mais à ne pas confondre avec le mot **step** que l'on trouvera plus loin).

Ainsi, dans p-pts :

(app 1/24 (7/24_42/24s1/24))

signifie, pour un problème rythmique, que dans l'unique plan (un seul objet est ici spécifié), les durées (dont le nombre serait spécifié dans n-pts de csolver) devront s'inscrire dans tous (cf. **s1/24**, on ne saute donc aucun pas) les points compris entre la 8° et la 43° double-croche de sextolet (cf.**7/24_42/24**).

- Toutefois - et cela vaut pour toutes les entrées de **csolver** et tous les modules de Situation - afin d'alléger la syntaxe, on peut omettre de spécifier les objets ; à ce moment là, la contrainte vaut pour tous les objets de la séquence.

Ainsi, le nombre 18 étant écrit dans **n-obj**, dans **p-pts** :

(60_76)

signifie, pour un problème harmonique, que tous les accords de la séquence (cf. **18** inscrit dans **n-obj**) seront compris entre le do3 et le mi4 (cf. **60_76**).

Autrement dit :

(60_76)

est ici strictement équivalent à

(0_17 (60_76)).

Il s'en suit, quelque soit les modules, qu'il n'est nécessaire de spécifier les objets que lorsque la séquence d'objets n'est pas toute entière concernée.

Attention : Il y a toutefois un petit inconvénient à cette convention (facultative) d'omission, inconvénient que l'on rencontre lorsqu'une expression ne spécifie pas d'objets particuliers et commence tout de même par un chiffre. A ce moment là, pour que Situation fasse bien la différence, on débute l'expression par une double parenthèse (et non une simple), parenthèse que l'on n'oubliera pas de refermer à la fin de l'expression. Cette précaution est notamment nécessaire pour les modules incluant la notion de voix (voir plus loin : **x-dst**, **x-dst filt**, **x-prof** et **x/x prof**).

• En outre, certaines entrées de **csolver** permettent une programmation dynamique, c'est-à-dire incluant une interpolation notée « **interp** ». C'est le cas de **p-pts**, **n-pts**, **i-dst** et **x-dst**.

L'interpolation est spécifiée par deux arguments devant comprendre le <u>même nombre</u> de termes :

- le premier argument définit le point de départ de l'interpolation
- le deuxième argument définit le point d'arrivée de l'interpolation

Ainsi, dans **p-pts** :

(0_89 (interp (24_48) (66_85)))

signifie, pour un problème harmonique, que les 90 premiers accords (cf. **0_89**) s'inscriront dans un ambitus défini par l'interpolation (cf. **interp**) allant de l'ambitus do0-do2 pour le premier accord (cf. **24_48**) à l'ambitus solb3-réb5 pour le quatre-vingt-dixième (cf. **66_85**).
Il s'agit donc ici d'un ambitus dynamique ascendant.
Par défaut, l'interpolation est linéaire. Cependant, il est possible de demander une interpolation plus concave ou convexe en faisant suivre le dernier argument de l'interpolation d'un <u>entier</u> supérieur à 1 ou d'un <u>flottant</u> inférieur à 1 respectivement.
Ainsi, par exemple :

(0_89 (interp (24_48) (66_85) 2))

pour une interpolation plus concave (cf. **2**) ;

(0_89 (interp (24_48) (66_85) .5))

pour une interpolation plus convexe (cf. **.5**).

• Notons maintenant que la lecture des contraintes se fait toujours de gauche à droite par Situation. Les dernières valeurs d'une contrainte (celles du deuxième argument d'une interpolation par exemple) sont donc prises comme valeurs par défaut par les <u>derniers</u> objets de la séquence si celle-ci en contient plus que n'en spécifie la contrainte.

Ainsi, si l'on reprend l'exemple précédent :

(0_89 (interp (24_48) (66_85)))

alors qu'il y a <u>plus</u> de 90 accords dans la séquence, 100 par exemple (**n-obj** = 100), les dix derniers accords prennent la valeur du deuxième argument de l'interpolation des 90 premiers. Autrement dit, l'ambitus de ces 10 accords sera ici de (66_85), et il est strictement équivalent d'écrire :

(0_89 (interp (24_48) (66_85)) **90_99 (66_85)**)

ou simplement :

(0_89 (interp (24_48) (66_85))

Mais ce système ne peut valoir, sans plus de spécifications, que pour les derniers accords (gardons l'exemple d'un problème harmonique) d'une séquence.

Il existe donc pour **p-pts**, comme pour toutes les entrées de **csolver** pouvant inclure une interpolation (**n-pts**, **i-dst** et **x-dst**), un module du menu « utilities » : **default-fill** (default-filling = remplissage par défaut), qui permet d'alléger la syntaxe en faisant prendre, par défaut, aux accords non spécifiés de la séquence :

- les valeurs du départ de l'interpolation (premier argument) aux accords situés avant les accords faisant l'objet de cette interpolation
- les valeurs de l'arrivée de l'interpolation (deuxième argument) aux accords situés après les accords faisant l'objet de cette interpolation

Le module **default-fill** se branche sur l'entrée **p-pts** (s'il s'agit d'une définition d'ambitus) de **csolver**.

Ainsi, avec **default-fill** et **n-obj** = 100 (séquence de 100 accords) :

(20_39 (interp (24_48) (66_85)) 60_79 (interp (66_85) (24_48)))

est strictement équivalent à :

(**0_19 (24_48)** 20_39 (interp (24_48) (66_85)) **40_59 (66_85)** 60_79 (interp (66_85) (24_48)))

qui est beaucoup plus lourd.

- Enfin, **p-pts** peut être défini graphiquement par le module **bpf_lib** du menu « kernel » de OpenMusic.

Ce module **bpf_lib** ne peut toutefois pas être branché directement dans l'entrée **p-pts** de **csolver** : entre les deux doit s'intercaler le module **bpf-ambdef** du sous-menu « utilities » de Situation (**bpf-ambdef** reçoit donc **bpf_lib** en entrée et connecte sa sortie à **p-pts** de **csolver**).

Le module **bpf-ambdef** comporte une fenêtre optionnelle (option_ sur le module sélectionné) permettant d'inscrire le nombre d'accords de la séquence pour échantillonner les courbes du **bpf_lib** et les mettre à l'échelle de la séquence.

La programmation consiste à entrer deux courbes définissant les limites inférieures et supérieures de l'ambitus, à verrouiller puis connecter la 2e sortie (de gauche à droite) du **bpf_lib** à la 1e entrée du **bpf-ambdef**.

3.1.1.3. n-pts **(number of points = nombre de points)**

Cette entrée permet de définir le nombre de points par objet, autrement dit le nombre de hauteurs (et non d'intervalles) des accords (densité harmonique) ou le nombre de durées des plans (densité rythmique).

- Si la densité est la même pour tous les objets, il suffit d'inscrire directement un nombre entier dans la fenêtre comme pour **n-obj**.

Situation résonant à partir de la notion d'objet, remarquons que 1, dans **n-pts**, permet d'obtenir, dans un problème harmonique, une séquence d'accords d'une seule hauteur, c'est-à-dire une mélodie, dont le nombre de notes est le nombre d'objets (inscrit dans **n-obj**).

- Si plusieurs densités sont possibles, il suffit de les inscrire sous forme de liste.

Ainsi :

(2 3 7)

signifie, dans un problème harmonique, que les accords seront de densité 2, 3 ou 7.

- On peut également spécifier la densité par un argument à deux termes séparés par « _ » :

- le premier désigne un nombre minimum de points
- le deuxième désigne un nombre maximum de points

Ainsi :

(12_17)

signifie, dans un problème rythmique, que tous les plans auront un minimum de 12 points d'impact et un maximum de 17 points d'impact.

- On peut encore spécifier la densité d'objets particuliers.

Ainsi :

(0_7 (4 5) 8 10 (6) 9 (2_4))

signifie, dans un problème harmonique, que les huit premiers accords (cf. **0_7**) auront au moins 4 hauteurs et au plus 5 hauteurs (cf. **4 5**), alors que les neuvième et onzième auront exactement 6 hauteurs, et le dixième au moins 2 hauteurs et au plus 4 hauteurs (cf. **2_4**) ;

((5) (7) (4))

signifie, dans un problème rythmique, que le 1° plan comprendra 5 points d'impact, le 2° 7 points d'impact et le 3° 4 points d'impact.

Notons donc que, lorsque les densités sont spécifiées pour des objets ordonnés et conjoints, il n’est pas nécessaire de numéroter les objets (le jeu de parenthèses suffit) ; ((5) (7) (4)) est ici équivalent à (0 (5) 1 (7) 2 (4))

• On peut enfin définir une programmation dynamique ; dans ce cas, le nombre d’intervalles doit être le même pour le départ et l’arrivée de l’interpolation.

Ainsi :

(0_15 (interp (1 3) (6 6)))

signifie, dans un problème harmonique, que les seize premiers accords (cf. **0_15**) auront une densité définie par une interpolation allant de 1 minimum - 3 maximum pour le premier (cf. **1_3**) à 6 exactement (**6** minimum - **6** maximum) pour le seizième.

Attention : comme il s’agit ici d’une interpolation, on doit mettre le même nombre de termes dans les deux arguments : il faut donc écrire pour le deuxième **(6 6)** et non simplement **(6)**.

3.1.1.4. i-dst **(internal distances = distances internes)**

Cette entrée permet de définir les distances internes des objets, autrement dit les intervalles des accords (intervalles verticaux), ou les intervalles des plans rythmiques (durées entre les points d’impact).

• Si les distances internes sont les mêmes pour tous les objets, il suffit de taper directement des nombres entiers entre parenthèses.

Ainsi :

(4 7 11)

signifie, dans un problème rythmique, que tous les accords de la séquence ne comprendront que des tierces majeures (cf. **4**), des quintes justes (cf. **7**), des septièmes majeures (cf. **11**) ou une combinaison de ces intervalles ;

ou :

(1/16_3/16s1/16)

signifie, dans un problème rythmique, que les points d’impact du plan rythmique seront distants d’une double-croche, d’une croche (2 doubles-croches) ou d’une croche pointée (3 doubles-croches).

• Si l’argument est précédé de **f** (force), l’objet comprendra exactement la superposition désignée, et de manière ordonnée (à partir du bas s’il s’agit d’un accord, du début s’il s’agit d’un plan).

Ainsi :

(2 (f (2 3 8)))

signifie, dans un problème harmonique, que le troisième accord comprendra de bas en haut une seconde majeure, une tierce mineure et une sixte mineure.

Notons que, dans ce cas, l'accord sera de densité 4 quelque soit la densité inscrite dans **n-pts** (**f** a donc le pouvoir d'écraser la spécification de **n-pts**) ;

ou :

(0 (f (3/16 1/16)) 1 (f (1/12 2/12)))

signifie, dans un problème rythmique, que le 1° plan rythmique sera croche pointée/ double croche, et le second croche de triolet/noire de triolet.

- Notons également que **i-dst** ne tient pas compte des intervalles par <u>enjambement</u> ou par <u>redoublement</u>.

Ainsi, par exemple, l'accord do-mi-sol est compatible avec la programmation :

(f (4 3)), tierces majeure et mineure, mais pas avec (f (4 3 7)) car la quinte do-sol par enjambement n'est pas comptée.

Quant au redoublement, cette notion n'existe pas non plus au niveau de **csolver**, qui considère les intervalles comme absolus : une dixième majeure, par exemple, n'est pas ici une tierce majeure redoublée mais l'intervalle 16 (cf. plus haut).

Nous verrons plus loin comment les notions d'enjambement et de redoublement interviennent au niveau des « standard constraints » avec le module **i-dst filt**.

- On peut bien sûr spécifier, dans un problème harmonique, les intervalles verticaux d'accords particuliers ou de sous-ensembles de la séquence :

  (0_7 (2 5 6) 8 (1 4) 9 11 (10 13) 10 12_19 (3 4 5))

- ou définir une programmation dynamique :

  (0_9 (interp (2 3 7) (4 6 10))

Dans ce cas, le nombre d'intervalles doit être le <u>même</u> pour le départ et l'arrivée de l'interpolation.

3.1.1.5. cnstr **(constraints = contraintes)**

Cette entrée permet de recevoir les modules du sous-menu « standard constraints » et les constructions de « user constraints » que l'on examinera plus tard.

3.1.1.6. x-dst (external distances = distances externes)

Cette entrée permet de définir les distances externes entre les objets, autrement dit, dans un problème harmonique, les intervalles (horizontaux) formés par des hauteurs d'accords différents ou, dans un problème rythmique, les intervalles formés par des points d'impact de plans différents.

Dans un problème harmonique, **x-dst** implique la notion de voix. Voyons donc ce que l'on entend dans Situation par « voix », sachant que la généralisation de cette notion ne sera abordée que lors de la présentation du module **x-dst_filt**.

(La notion de voix s'applique bien sûr aux problèmes rythmiques ; cependant, dans ces problèmes, elle y est paradoxale puisque c'est **i-dst** qui la représente le plus intuitivement. Nous n'exemplifierons donc la syntaxe des voix qu'avec des problèmes harmoniques et nous contenterons de quelques exemples rythmiques en fin de paragraphe).

2° insert : syntaxe des voix

- Lorsque la densité des accords est invariable, la notion de voix est très simple.

On appelle :

- **l** (lower voice) ou **0** (voix 0) la voix formée par la succession de la première note à partir du bas de chaque accord,
- **1** (voix 1) la voix formée par la succession de la deuxième note à partir du bas de chaque accord,
- **2** (voix 2) la voix formée par la succession de la troisième note à partir du bas de chaque accord etc... jusqu'à :
- **u** (upper voice) pour la voix formée par la succession de la note la plus aiguë de chaque accord. Autrement dit, à quatre voix, l'équivalent de ce qu'on appelait en harmonie classique : la basse, le ténor, l'alto et le soprano.

- Lorsque la densité des accords est variable :

- les voix extrêmes ne posent pas de problème : quelque soit la densité des accords, on désigne par :
- **l** (lower voice) ou **0** : la plus grave,
- **u** (upper voice) : la plus aiguë,

sachant qu'au cas où la densité passe à 1, c'est la voix la plus aiguë qui gère la hauteur restante ;

- pour ce qui est des autres voix, on évide l'accord de l'intérieur et par le bas, en sorte de préserver le plus longtemps possible les voix extrêmes considérées comme prioritaires ; c'est-à-dire que si l'on passe par exemple de la densité 4 à la densité 3 :
- **l** reste **l**
- **2** reste **2**
- **u** reste **u**

en continuant de gérer leurs propres contraintes, tandis que

- **1** disparaît, en précisant toutefois que cette voix (ici le « ténor ») forme, avant de disparaître, un dernier intervalle avec la hauteur de la voix située immédiatement au-dessus d'elle (ici « l'alto ») dans l'accord qui suit (et où elle ne compte donc plus de hauteur propre), et que cet intervalle obéit aux contraintes de la voix 1.

Il en est de même lorsqu'une voix apparaît (passage de la densité 3 à la densité 4 par exemple).

En résumé :

- la voix qui disparaît ou apparaît est toujours la voix située immédiatement au-dessus de la plus basse, ou la plus basse elle-même lorsqu'il n'y a plus que deux voix ;
- toute voix qui disparaît forme un intervalle avec la voix se situant immédiatement au-dessus d'elle dans l'accord qui suit ;
- toute voix qui apparaît forme un intervalle avec la voix se situant immédiatement au-dessus d'elle dans l'accord qui précède ;
- lorsqu'il ne reste plus qu'une voix, c'est toujours la voix supérieure.

Pour que ce système de « décrochage » fonctionne lorsque l'on veut contraindre toutes les voix, il faut donc au départ définir autant de voix qu'il y en a dans l'accord de plus grande densité. Il est clair, toutefois, que contraindre toutes les voix simultanément lorsqu'elle sont nombreuses n'est qu'une possibilité théorique qui - c'est le moins qu'on puisse dire - limite considérablement les possibilités d'opérer en même temps un contrôle vertical de tous les accords.

Quoiqu'il en soit, ce système permet de conserver les notions de voix et d'intervalle horizontal quelque soit la densité des accords.

Revenons maintenant à **x-dst**.

- Si, pour une ou des voix données, les intervalles horizontaux sont les mêmes pour tous les couples d'accords de la séquence, il suffit de taper dans la fenêtre **x-dst** :

- le nom de cette ou de ces voix, suivi
- des intervalles,

sachant que si aucune voix n'est spécifiée, il s'agit par défaut de la voix la plus aiguë (**u** = upper voice).

Ainsi :

(1 3 4)

ou

(u (1 3 4))

signifie :

- que la voix formée par la succession de la note la plus aiguë de tous les accords de la séquence ne comprendra que des secondes mineures, des tierces mineures, des tierces majeures ou une combinaison de ces intervalles ;

(l (1_4) u (2 3))

signifie :

- que la voix formée par la succession de la note la plus grave de tous les accords de la séquence ne comprendra que des intervalles compris entre la seconde mineure et la tierce majeure ou une combinaison de ces intervalles
- et que la voix formée par la succession de la note la plus aiguë de tous les accords de la séquence ne comprendra que des secondes majeures, des tierces mineures ou une combinaison de ces intervalles ;

((0 (4 6) 1 (3_7) u (1 5)))

ou

(l (4 6) 1 (3_7) u (1 5))

signifie :

- que la voix formée par la succession de la note la plus grave (cf. **0** ou **l**) de tous les accords de la séquence ne comprendra que des intervalles de tierce majeure et de quarte augmentée (cf. **4 6**) ou une combinaison de ces intervalles,
- que la voix formée par la succession de la note située immédiatement au- dessus de la note la plus grave (cf. **1**) de tous les accords de la séquence ne comprendra que des intervalles compris entre la tierce mineure et la quinte juste (cf. **3_7**) ou une combinaison de ces intervalles
- et que la voix formée par la succession de la note la plus aiguë (cf. **u**) de tous les accords de la séquence ne comprendra que des secondes mineures, des quartes justes (cf. **1 5**) ou une combinaison de ces intervalles.
  - Important : Dans l'exemple précédent, on remarque que dans la première notation possible, qui commence par **0** pour désigner la voix la plus basse, ce **0** est

précédé de deux parenthèses et non d'une seule comme à l'accoutumée. En effet, cette expression ne spécifie pas d'accords particuliers de la séquence (ils sont donc tous concernés). Or, compte tenue de sa syntaxe générale (cf. 1° insert), il faut que Situation puisse reconnaître que **0** ne désigne pas ici le premier accord mais la voix la plus basse. C'est pourquoi on ajoute une seconde parenthèse (qu'il faut bien sûr refermer à la fin de l'expression), seconde parenthèse qui n'est pas nécessaire si l'on écrit **l** et non **0**. Il est donc préférable d'utiliser **l**, sans que cela puisse bien sûr dispenser de la seconde parenthèse lorsque l'on veut spécifier des voix qui ne sont ni **l** ni **u** : **1**, **2** etc...

• Pour revenir un instant, à propos de ce dernier exemple, sur la notion un peu délicate de voix en cas de densité variable :

- imaginons que la densité des accords passe de 3 à 2 :
- la voix **1** sauterait et Situation fonctionnerait comme si l'on avait écrit pour le sous-ensemble de densité 2 :

  (l (4 6) u (1 5)) ;
- quant au moment précis du passage de la densité 3 à la densité 2, la voix centrale obéirait, pour se fondre dans la voix supérieure, à sa propre contrainte intervallique ;
- et si la densité des accords passait à 1 :
- **l** sauterait et Situation fonctionnerait comme si l'on avait écrit pour le sous-ensemble de densité 1 :

  (u (1 5)) ou simplement (1 5) ;
- quant au moment précis du passage de la densité 2 à la densité 1, **l** obéirait, pour se fondre dans **u**, à sa propre contrainte intervallique

• On peut bien sûr spécifier les intervalles horizontaux de couples d'accords particuliers ou de sous-ensembles de la séquence :

(0_9 (l (1_4) u (2 3)) 6_7 (2 (3 5)))

• et définir une programmation dynamique :

(0_9 (l (interp (1 4) (3 9)) u (interp (2 3) (4 5))) 6_7 (2 (3 5)))

Dans ce cas, le nombre d'intervalles doit être le même pour le départ et l'arrivée de l'interpolation.

• Donnons maintenant quelques exemples d'emploi de **x-dst** dans des problèmes rythmiques. Par commodité, nous conservons la syntaxe **l** (lower voice) et **u** (upper voice).

(0)

strictement équivalent à

(u (0))

puisqu'il s'agit toujours de la « upper voice ! » par défaut (c'est-à-dire, traduit rythmiquement, le dernier point d'impact du plan)

signifie que tous les derniers points d'impacts des plans seront à l'unisson (cf. **0**), autrement dit synchrones.

(1 3 (0 (0 1/4) u (1/8)))

signifie que dans les premier et quatrième plans rythmiques (cf. **1 3**) les premiers points d'impacts (cf. **0** mais on aurait pu écrire **l** (lower voice!)) seront synchrones ou distants d'une noire (cf. **0 1/4**), et les derniers (cf. **u**) distants d'une croche (cf. **1/8**).

3.1.1.7. data **(données)**

Cette entrée, qui est optionnelle et n'apparaît donc qu'en rajoutant une entrée optionnelle à **csolver**, permet d'entrer toutes sortes d'objets déjà constitués (successions d'accords ou superpositions de plans rythmiques), objets calculés ou non par Situation.

• Ces objets se présentent sous forme de listes de listes, en code MIDI pour les problèmes harmoniques ; par exemple :

((60 64 66) (61 64 69) (64 66 71))

Cette entrée (comme tous les modules de Situation, rappelons-le) accepte la syntaxe d'expression courante : *****, **_** et **s.**

• Si, dans un problème harmonique par exemple, on entre une séquence d'accords dans **data**, les accords de cette séquence sont déjà constitués : ils sont un certain nombre, s'inscrivent dans un certain ambitus, ont une certaine densité et sont constitués d'intervalles verticaux.

Ils constituent donc le domaine sur lequel agiront les « standard constraints » ou « user constraints » et certains réglages de **csolver**, lesquels contraintes et réglages n'agiront donc pas pour constituer cette séquence (elle existe déjà) mais pour en trier, réordonner, analyser etc... le contenu.

• Puisque nous sommes dans **csolver**, voyons comment agissent les entrées de ce module lorsqu'une base de données, par exemple harmonique, est entrée dans **data**.

- **n-obj** permet de spécifier combien d'accords de la séquence ont veut retenir, il suffit alors d'inscrire un chiffre dans **n-obj** qui correspondra aux **n** premiers accords de **data**.

Si l'on veut choisir plus précisément des accords de **data**, on utilisera, à la sortie du **buffer** ou de l'éditeur les renfermant, les modules **posn-match** et **expand-lst** du menu « kernel » de OpenMusic : il suffira de choisir les accords dans **expand lst**, de connecter la sortie de ce module à la deuxième entrée de **posn-match**, lequel recevra lui-même en première entrée l'éditeur ou le **buffer**, et de connecter la sortie de **posn-match** à **data** de **csolver**.

- **p-pts** permet :

- de sélectionner des accords en fonction de leur ambitus, par exemple

  (48_72)

  signifie que l'on ne veut retenir que les accords de **data** compris entre le do2 et le do4 ;

- ou de sélectionner et d'ordonner les accords en fonction d'un ambitus dynamique, par exemple :

  (interp (36_60) (60_84))

  signifie que les accords seront choisis et devront s'aligner selon leur appartenance à l'ambitus définie par l'interpolation allant du registre do1- do3 à do3-do5.

- **n-pts** permet :

- de sélectionner des accords en fonction de leur densité, par exemple :

  (2 4)

  signifie que l'on ne veut retenir que les accords de densité 2, 3 ou 4

  (rappel : premier terme de l'argument = densité minimum, deuxième terme de l'argument = densité maximum

- ou de sélectionner et d'ordonner les accords en fonction de leur densité, par exemple :

  (interp (2 2) (4 5))

  signifie que les accords seront choisis et ordonnés selon une densité croissante définie par l'interpolation allant de densité 2 exactement à densité 4 ou 5.

- **i-dst** ne peut être utiliser parce que les intervalles verticaux de **data** sont déjà définis. On peut bien sûr agir sur des accords de la séquence entrée dans **data** en fonction de configurations particulières de ces intervalles, mais c'est là la fonction de la contrainte **i-dst filt** que l'on verra plus loin (attention : ne pas oublier de taper nil dans **i-dst** lorsqu'une séquence est entrée dans **data**).

- **x-dst**, enfin, pourra fonctionner normalement avec une séquence d'accords entrée dans **data**, puisque les intervalles horizontaux y sont virtuellement

présents mais non pris en compte : **x-dst** permettra donc de choisir et d'ordonner la séquence en fonction des intervalles que l'on y inscrira (voir **3.1.1.5.** : **x-dst**)

Nous savons maintenant comment programmer les entrées standards de **csolver** pour définir un domaine ou agir sur un domaine entrée dans **data**. Voyons à présent comment évaluer **csolver**, gérer la ou les solutions et éditer celles-ci.

3° insert : évaluation, gestion, édition

• Pour évaluer le module **csolver** et recueillir la solution, il faut tout d'abord connecter la sortie de **csolver** à l'entrée d'un module chargé de construire la solution :

(13 **ch-sol** (chords solution = solution d'accords) pour un problème harmonique,

(14 **rtm-sol** (rhythmic solution = solution rythmique) pour un problème rythmique ;

ensuite, pour afficher la solution dans un éditeur, il suffit de brancher la sortie de **ch-sol** dans un module éditeur quelconque et **rtm-sol** dans un module **poly-rtm**.

• Le module **ch-sol** permet de construire la solution en hauteurs MIDI. Il comporte une entrée optionnelle permettant de transposer la solution ; il suffit d'y inscrire un nombre en midics, par exemple : 1200 pour transposer d'une octave.

• Le module **rtm-sol** permet de construire la solution rythmique ; il comporte 4 entrées standards :

- **sol** (rhythmic solution = solution rythmique)
- **sign** (time signature = chiffrage des mesures)
- **tempo**
- **u-obj** (unit-objects = unité de distance des objets)

et 4 entrées otpionnelles :

- **h-obj** (harmonic-objects = objets harmoniques)
- **i-pts** (instanciation-points = instanciation des points)
- **c-pts** (correction-points = correction des points)
- **slur ?** (slur ? = liaison ?)
- **sol** (solution = solution rythmique)

  Cette entrée permet de recevoir la solution du problème rythmique calculé dans **csolver**.

- **sign** (signature = chiffrage des mesures)

  Cette entrée permet de définir la ou les mesures dans lesquelles va s'afficher la solution.

- **tempo**

  Cette entrée permet de définir le tempo de la situation.

- **u-obj** (unit-objects = unité de distance des objets)

  Cette entrée permet de définir l'unité de distance des objets : 1/4 pour la noire, 1/8 pour la croche, 1/10 pour la croche de quintolet etc…

  N'est possible qu'une seule unité par objet, sachant que cette unité inclut tous ses multiples entiers ; par exemple, 1/16 (double-croche) inclut 2/16 (croche), 3/16 (croche pointée) etc… mais pas 1/12 (croche de triolet), 1/5 (noire de quintolet) etc...

  N'importe quelle unité est possible (pourvu qu'elle soit cohérente avec la programmation de **csolver**), mais une seule par objet.

  Lorsqu'il y a plusieurs objets, c'est-à-dire plusieurs plans rythmiques, les unités s'inscrivent de manière ordonnée ; ainsi :

  (1/16 1/28 1/12 4/9)

  signifie que le premier plan rythmique aura pour unité la double-croche, le second la double-croche de septolet, le troisième la croche de triolet et le quatrième la ronde de neunolet de noire.

- **h-obj** (harmonic-objects = objets harmoniques)

  Cette entrée permet de recevoir des objets harmoniques (des accords calculés ou non par Situation).

- **i-pts** (instanciation-points = instanciation des points)

  Cette entrée, qui reçoit une liste de listes correspondant aux différents plans rythmiques, permet de régler l'instanciation des points d'impact : leur nombre de valeurs (s'il y en a plusieurs, disons qu'elles forment un « motif ») et la distribution de ces valeurs par rapport aux points d'impact (avant s'indique par « **-** », après par rien, et si des valeurs se distribuent à la fois avant et après un même point, on l'indique par un couple de deux valeurs entre parenthèses). Ainsi :

  ((-2 3 –1) (-4 (-2 6)))

  signifie que :

- dans le premier plan rythmique, les trois points d'impact sont instanciés par des motifs de 2,3 et 1 valeurs et se placent respectivement avant, après et avant leur point d'impact ;
- dans le deuxième plan rythmique, le premier point d'impact est instancié par un motif de 4 valeurs placées avant le point d'impact, et le deuxième point d'impact est instancié par un motif de 8 valeurs, 2 avant et 6 après.

Les durées des motifs sont celles de l'entrée **u-dst** (les unités utilisées par les plans) et les hauteurs sont les accords ou ensembles d'accords importés dans **rtm-sol** par **h-obj**.

- **c-pts** (correction-points = correction des points)

Cette entrée permet de corriger le placement des valeurs instanciées (simples ou multiples) par rapport aux points d'impact des plans.

Les plans sont indiqués par un numéro (0 le premier, 1 le deuxième etc…) et sont suivis d'un argument comprenant autant de couple de nombres qu'il y a de points à corriger ; le premier nombre de ce couple correspond au numéro du point dans le plan (0 le premier, 1 le deuxième etc…), le second indique de combien d'unités du plan le motif instancié doit être déplacé et dans quel sens : « **-** » pour « à gauche » (avant), rien pour « à droite » (après). Ainsi :

((0 (1 –5) (8 2)) (2 (3 –7)))

signifie que :

- dans le premier plan rythmique, 2 valeurs (2 valeurs simples ou 2 motifs) sont à déplacer : le 2° de 5 unités vers la gauche (par exemple, 5 doubles-croches si l'unité est 1/16) et le 9° de 2 unités vers la droite ;
- dans le troisième plan rythmique, 1 valeur est à déplacer : la 4° de 7 valeurs vers la gauche.
- **slur ?** (slur = liaison)

Cette fenêtre permet de dire par **y** (yes) si l'on veut que les silences entre les points d'impact soient remplacés par des tenues liées.

- Si on le désire, on peut suivre dans la fenêtre « lisp » la progression du calcul de la solution. Si une solution est difficile à trouver, cela permet de savoir sur quel objet Situation butte (dans ce cas, l'affichage de la solution piétine sur l'objet précédent).

En fonction de la programmation que l'on a faite, on peut alors en déduire plus facilement la contrainte trop contraignante et éventuellement la modifier. Cependant, s'il n'y a pas de solution, la fenêtre « lisp » indique qu'elle est cette contrainte si elle l'a repérée (il s'agit souvent de **p-pts**).

En fait, à l'usage, on apprend à savoir très vite, d'après la progression du calcul qui s'affiche, si une solution va être trouvée ou non et dans quel délais.

On peut alors décider d'interrompre la recherche de solution en faisant « abort » ou « break » (dans le menu « lisp »), puis recueillir, si l'on veut, la solution partielle avec le module **part-sol** du sous-menu « utilities » de Situation. On connecte alors la sortie de

ce module à l'entrée de **ch-sol** ou de **rtm-sol** et on évalue l'éditeur pour que la solution partielle s'y affiche.

Si l'on a fait « break » (et non « abort »), on peut alors décider de faire « continue » (toujours dans le menu « lisp ») pour poursuivre la recherche de solution.

Si, dans un problème harmonique par exemple, on a fait « break » ou « abort » parce qu'on s'est aperçu dans l'affichage de la progression du calcul que celui-ci buttait sur le chiffre 14 (c'est-à-dire au 15° accord et au seuil d'un 16° introuvable ou difficilement trouvable), on peut demander à Situation de trouver une solution sans se soucier de la contrainte associée à cet accord et l'empêchant d'être trouvée. C'est à cela que sert, dans **csolver**, l'entrée optionnelle **x-sol**.

3.1.1.8. x-sol (x-solution = solution à partir de x objet)

Cette entrée optionnelle permet d'inscrire un numéro d'objet à partir duquel le calcul doit être relancé.

Lorsque l'on fait suivre ce numéro de la lettre **t** (exemple : **15 t**), le calcul se fera sans tenir compte des contraintes associées à ce numéro d'objet.

Ainsi, dans un problème harmonique par exemple, avec **15 t** dans **x-sol** de **csolver** :

- après « abort », il faut réévaluer **csolver** ; à ce moment là, une nouvelle solution va être cherchée depuis le début mais sans que soit tenu compte des contraintes attachées à l'accord 15 ;
- après « break », il faut faire « continue » ; à ce moment là, le calcul va repartir de l'accord 15 sans s'occuper des contraintes qui lui sont attachées.

3.1.1.9. n-sol (n-solutions = nombre de solutions)

Cette entrée optionnelle permet de définir le nombre de solutions demandées en y inscrivant un chiffre. Ce nombre est de 1 par défaut.

- Pour comprendre pourquoi cette entrée n'est qu'optionnelle, il faut savoir comment procède Situation pour trouver une solution.

Après que le domaine des points possibles ait été défini par la programmation de **csolver**, ce domaine est filtré par les « standard constraints » et/ou « user constraints » (si on en a programmées) pour trouver l'un après l'autre les objets de la solution. La plupart du temps, il existe plusieurs possibilités pour chaque objet. Situation en choisit alors un aléatoirement.

Lorsqu'on demande dans **n-sol** trois solutions par exemple, par souci de rapidité Situation va utiliser les mêmes choix pour chaque objet aussi longtemps que possible et se contenter de varier la solution au dernier moment, c'est-à-dire au moment de choisir les derniers objets. Si le nombre de solutions possibles était en fait très grand (ce qui est souvent le cas) ces trois solutions risquent d'être les mêmes sauf pour le dernier objet, ce qui n'est pas très intéressant.
En revanche, lorsqu'on évalue plusieurs fois **csolver** séparément, on obtient des solutions d'autant plus possiblement différentes que le nombre de solutions était important pour chaque objet, puisque Situation relance les dés pour chacun d'entre eux. Ce n'est d'ailleurs pas le moindre intérêt de Situation que de procéder ainsi et de proposer souvent des solutions qu'on n'imaginait pas.
C'est pourquoi **n-sol** n'est intéressant que lorsqu'il existe un très petit nombre de solutions : à ce moment là, Situation est forcé de les trouver même lorsqu'elles sont bien cachées, alors qu'en réévaluant plusieurs fois **csolver**, on risque de ne rapporter à chaque fois que la ou les mêmes solutions : les plus évidentes à trouver.

- Pratiquement, si on utilise **n-sol**, c'est-à-dire si l'on veut plus d'une solution par évaluation, il est nécessaire de verrouiller le module **csolver** avant évaluation, puis de connecter à sa sortie autant de modules **first**, **second**, **third** etc... qu'il y a de solutions demandées. Bien sûr, **first** puise alors dans **csolver** la première solution, **second** la deuxième etc...

Au lieu des modules **first**, **second** etc... , on peut utiliser le module **posn-match** du menu « kernel » de OpenMusic.
Il suffit ensuite de connecter le module **posn-match** ou les modules **first**, **second** etc... à autant de modules **ch-sol** (ou **rtm-sol)** et d'éditeurs pour y afficher les solutions.

3.1.1.10. merge **(= fusionner)**

Cette entrée optionnelle, en fait ces entrées car il y a autant d'entrées optionnelles **merge** que l'on veut, permettent d'accueillir des modules **add-csolver** (un seul par entrée).

3.1.2. wsolver ***(weak solver = moteur de résolution de contraintes faibles)***

Ce module est exactement le même que **csolver** si ce n'est qu'il introduit la notion de contraintes « faibles ».

Attention : **wsolver** n'est encore qu'expérimental et ne fonctionne actuellement que pour les contraintes standards (**standard constraints** et **user constraints**) ; il fonctionnera toutefois pour les **generic constraints** dans la prochaine version (novembre 99).

Wsolver est complètement indépendant de **csolver** : les deux moteurs peuvent être utilisés dans un patch avec exactement les mêmes entrées. La différence est que **wsolver** s'efforce de donner des informations utiles dans le cas de problèmes inconsistants, c'est-à-dire dont il n'existe pas de solution satisfaisant à toutes les contraintes.

L'idée est d'associer un coût à chaque solution. Le coût est fonction du nombre de contraintes violées ; plus précisément, il est la somme des contraintes non satisfaites, pondérée par leur valeur d'importance. Par exemple, une solution qui viole deux contraintes, l'une de valeur 3 et l'autre de valeur 4, a un coût égal à 7. En quelque sorte, le coût est la « distance » entre la solution trouvée et la solution parfaite.

Wsolver donne deux contrôles au programmeur :

- **accept** (contrôle d'acceptation)
- **start** (contrôle d'orientation de la recherche de solutions)

accept donne le coût maximum des solutions que l'on considère comme acceptables ; par exemple, si la valeur est égale à 3, la machine s'arrêtera dès qu'elle aura trouvé une solution dont le coût est inférieur ou égal à 3, ce qui peut vouloir dire que la solution a violé une contrainte de valeur 3 ou trois contraintes de valeur 1 ou une de 2 et une de 1.

Pour comprendre le contrôle **start**, il faut connaître quelques petits choses sur le fonctionnement du moteur.

Wsolver utilise la technique dite **branch&bound**. Il commence ainsi par trouver très rapidement une « solution » même si elle viole toutes les contraintes, puis explore à nouveau le domaine en essayant d'en trouver une de moindre coût, puis une autre et ainsi de suite jusqu'à ce qu'il arrive à trouver une solution acceptable.

Tout cela, bien sûr, peut entraîner beaucoup de calculs et donc de temps. L'intérêt est que l'utilisateur, s'il perd patience, peut arrêter la machine et lui demander de donner la meilleure solution jusqu'ici trouvée.

L'utilisateur peut également aider **wsolver** en lui indiquant par quel genre de solution commencer l'exploration ; c'est la fonction de **start** de donner le coût des solutions par lesquelles il faut commencer l'exploration. Si, par exemple, **start** = 6, **wsolver** va éliminer tous les chemins d'exploration conduisant aux solutions dont le coût est supérieur à 5 ; il va donc très probablement commencer par trouver une solution dont le coût est égal à 5.

Notons que si **start** = 1 et **accept** = 0, **wsolver** agit exactement comme **csolver** puisqu'il va essayer de commencer par une solution dont le coût est inférieur à 1, c'est-à-dire satisfaisant à toutes les contraintes (il peut être alors intéressant de comparer l'efficacité des deux moteurs).

Toutes les contraintes, dans Situation 3.0, ont une entrée optionnelle supplémentaire permettant de leur donner une valeur d'importance. Cette entrée est ignorée par **csolver** mais prise en compte par **wsolver**.

Les valeurs des contrôles **start** et **accept** doivent être données dans l'entrée **p-pts** de **wsolver** avant la syntaxe habituelle ; par exemple :

(start 3 accept 1 0_24 (48_72))

Pour un exemple, voir le patch *weak constraints* dans le dossier Situation 3.0 examples.

3.1.3. *add-csolver* ***(additional constraints solver = moteur de résolution de contraintes additionnel)***

Ce module est exactement le même que **csolver** si ce n'est qu'il est prévu pour inscrire la solution qu'il calcule à la suite de celle de **csolver**.

Si **csolver** calcule par exemple une séquence de 25 accords, le premier accord de **add-csolver** sera donc le 26°. On peut ainsi composer une chaîne de séquences d'accords en reliant **csolver** à autant de **add-csolver** que l'on veut, via les entrées **merge**, et éditer le résultat global dans le même éditeur.

3.2. standard constraints - (contraintes standards)

Le menu « standard constraints » contient quatre sous-menus :

- **Distances control** (contrôle des distances)
- **Points control** (contrôle des points)
- **Profiles control** (contrôle des profils).

Tous les modules de ces sous-menus sont des contraintes se connectant à l'entrée **cnstr** de **csolver**.

Lorsqu'on utilise plus d'un de ces modules, on utilise pour les réunir (dans n'importe quel ordre) le module extensible **list** du menu « kernel » de OpenMusic, lequel doit alors être connecté à l'entrée **cnstr** de **csolver**.

Rappelons ici que **csolver** permet de définir un domaine où pourront s'inscrire les solutions ou de recevoir un domaine entré dans **data**. Dans le premier cas, le rôle des contraintes sera donc de filtrer ce domaine pour trouver une solution plus précise, dans le deuxième de sélectionner, ordonner, analyser etc... une solution déjà existante.

Pour alléger cette documentation déjà copieuse, on ne donnera, sauf exceptions ici ou là, que des exemples du premier cas.

3.2.1. Distances control **(contrôle des distances)**

Le sous-menu **Distances control** contient sept modules :

- **i-dst_filt** (internal distances filter = filtre des distances internes)
- **i-dst/reg_filt** (internal distances/region filter = filtre des distances internes par région)
- **i/i-dst_filt** (internal/internal distances filter = filtre des distances internes les unes par
  rapport aux autres)
- **i-dst_rnw** (internal distances renewhal = renouvellement des distances internes)
- **x-dst_filt** (external distances filter = filtre des distances externes)
- **x-dst/reg_filt** (external distances/region filter = filtre des distances externes par région)
- **x-dst_rnw** (external distances renewhal = renouvellement des distances externes)

Tous les modules de **Distances control** permettent de préciser les configurations de distances dont, rappelons-le, seuls les configurations génériques ont été jusque là spécifiés dans **csolver** par **i-dst** et **x-dst** ; ils sont de deux types :

- **_filt** (filter = filtre)
- **_rnw** (renewhal = renouvellement)

chaque type admettant une syntaxe propre.

4° insert : syntaxe des modules _filt

Les modules **_filt** admettent :

- la syntaxe Lisp standard :
- **not** = interdiction
- **and** = obligation
- **or** = alternative
- la syntaxe d'expression courante :
- = n'importe combien et n'importe quel caractère y compris rien
- **?** = un seul caractère et n'importe lequel
- et une syntaxe propre à Situation :
- **ints** : lorsque un chiffre est précédé de **ints**, la distance à laquelle il est associé tient compte des enjambements ;
- **t** : lorsque un chiffre est suivi de **t**, la distance à laquelle il est associé tient compte de tous ses redoublements (à l'octave) ; **t** est inconsistant pour les problèmes rythmiques ;
- **step** : permet d'interpréter des contraintes externes en fonction d'ensembles d'objets distincts et non successifs ;
    - en effet, toute contrainte externes standard examine des couples, triplets, quadruplets etc... d'objets consécutifs, par exemple : 1°- 2°, 2°-3° etc... ou 1°-2°-3°, 2°-3°-4° etc...
    - avec **step**, en revanche, la contrainte tiendra compte de couples, triplets, quadruplets etc... distincts, par exemple : 1°-2°, 3°-4° etc... ou 1°-2°-3°, 4°-5°-6° etc...

    autrement dit, **step** permet la formation de patterns ;
- **x** = distance (ou point) quelconque (comme **?** donc) mais considéré comme répétition n fois du même (il se décline donc toujours ainsi : (**x x**), (**x x x**) etc... :
    - (**x x**) concerne la répétition de toute distance quelconque une fois ou plus,

- (**x x x**) concerne la répétition de toute distance quelconque deux fois ou plus (mais pas une fois) ;

le nombre de **x** désigne donc un seuil.

- Il faut savoir enfin :

- que les objets se lisent toujours de bas en haut pour les accords et de gauche à droite pour les rythmes, et toujours de gauche à droite dans la syntaxe ;
- et que lorsqu'un chiffre se trouve à l'extrême gauche ou à l'extrême droite d'un argument sans * ni **?**, il désigne une distance interne se situant respectivement en première ou en dernière position dans l'objet, ou une distance externe se situant obligatoirement en première ou en dernière position dans un ensemble d'objets.

3.2.1.1. i-dst_filt (internal distances filter = filtre des distances internes)

i-dst_filt permet de filtrer les configurations de distances internes.

- Ainsi, pour un problème harmonique par exemple, si on écrit dans **i-dst_filt** :

(not (or (ints 12 t) (2 2 *) (? ? 5 *)))

ne seront interdits (cf. **not**) que les accords obéissant à l'une ou l'autre (cf. **or**) des trois configurations spécifiées :

- **(ints 12 t)** = les octaves sous toutes leurs formes : directs (cf. **12**), par enjambement d'autres intervalles (cf. **ints**) ou par redoublement (cf. **t**),
- **(2 2 *)** = les superpositions de deux secondes majeures (cf. **2 2**) lorsque la première d'entre elles se situe en première position, c'est à dire dans le grave de l'accord (cf. rien à gauche) et quelque soit ce qui se trouve en intervalle et en nombre au-dessus (cf. * à droite),
- **(? ? 5 *)** = les quartes justes (cf. **5**) lorsqu'elles se trouvent au-dessus de deux autres intervalles quelqu'ils soient (cf. **? ?**), c'est-à-dire en troisième position à partir du bas, et quelque soit ce qui se trouve en intervalle et en nombre au-dessus (cf. * à droite).

Autrement dit :

- l'accord do-sol-réb-sol est impossible (octave par emjambement du réb)
- fa-sol-la-mib-fa# également (deux secondes majeures superposées dont la première est en première position : fa-sol-la)
- la-do-ré-sol-sib-do# également (une quarte en troisième position : ré-sol).

- Si on reprend le même exemple en remplaçant **or** par **and** :

(not (and (ints 12 t) (2 2 *) (? ? 5 *)))

ne seront <u>interdits</u> (cf. **not**) que les accords obéissant <u>à la fois</u> (cf. **and**) aux trois configurations, par exemple : do-ré-mi-la-do

Si on remplace **not** par **and** :

(and (or (ints 12 t) (2 2 *) (? ? 5 *)))

ne seront <u>autorisés</u> (cf. **and**) que les accords obéissant <u>à l'une ou l'autre</u> (cf. **or**) des trois configurations, par exemple : si-do#-ré-sol

Et si l'on remplace **not** par **and** et **or** par **and** :

(and (and (ints 12 t) (2 2 *) (? ? 5 *)))

ne seront <u>autorisés</u> (cf. **and**) que les accords obéissant <u>à la fois</u> (cf. **and**) aux trois configurations, par exemple : do-ré-mi-la-do.

- Enfin :

(not (x x))

signifiant que toute répétition, au moins une fois, du même intervalle est interdite :

(and (not (x x)) (and (? ? 5 *)))

signifie qu'il est obligatoire (cf. **and**) dans tous les accords de la séquence d'avoir une superposition de deux intervalles quelconques, d'une quarte et de n'importe quel intervalle en genre et en nombre (cf. **(and (? ? 5 *)**)), mais que ces intervalles ne pourront pas être une quarte d'une part, et devront tous être différents d'autre part (cf. **(not (x x)**)).

- Un exemple rythmique maintenant :

(or (0 (not (3/16 1/16 **))** **1** **(and (** 1/2 ?))))

signifie que :

- ou bien (cf. **or**) le premier plan rythmique (cf. **0**) ne devra pas (cf. **not**) comporter le rythme croche-pointée/double-croche (cf. **3/16 1/16**) pour commencer (cf.rien à gauche, * à droite),
- ou bien (cf. **or**) le deuxième plan rythmique (cf. **1**) devra (cf. **and**) comporter une blanche (cf. **1/2**) n'importe où sauf en dernière position (cf. * à gauche et **?** à droite).

Cette syntaxe générale est extrêmement puissante puisqu'elle permet de spécifier n'importe quelle configuration. Et ce d'autant qu'elle se combine avec la syntaxe générale de Situation qui permet de spécifier des ensembles d'objets ou des objets particuliers, par exemple :

(0_7 (not (and (2 2 *) (? ? 5))) **8 (and (ints 12 t)) 9_46 (not (or (3 ?)** (1 1 1 *))))

3.2.1.2. i-dst/reg_filt **(internal distances/regions filter = filtre des distances internes en fonction des régions)**

Ce module est le même que **i-dst_filt** à ceci près qu'il permet de filtrer les configurations de distances internes en fonction des régions, c'est-à-dire des registres définis par des fourchettes de hauteurs en code MIDI pour les problèmes harmoniques, et des zones de plans définis par des fourchettes de points d'impact compte tenu d'une unité (d'une impulsion) pour les problèmes rythmiques.

La contrainte est spécifiée par un argument à deux termes donnant successivement :

- la limite inférieure de la région
- la limite supérieure de la région.

- Ainsi, dans un problème harmonique par exemple :

  (0_7 (36 61 (not (and (* 2 2 *) (? 5 *)))))

signifie que pour les 8 premiers accords (cf. **0_7**), il est interdit (cf. **not**) d'avoir à la fois (cf. **and**) dans un même accord les configurations d'intervalles verticaux (*** 2 2 ***) et (**? 5 ***) si ces accords se situent dans un registre compris entre le do1 et le do#3 (cf. **36 61**).

- Dans un problème rythmique :

  ((0_14/20 (not (* 2/20 3/20 *))))

signifie que, dans une région correspondant aux trois premiers temps (cf. **0_14/20** = 15 doubles-croches de quintolet à partir du début, donc les trois premiers temps), il est interdit d'avoir le rythme croche/croche pointée de quintolet.

Remarquons la double parenthèse parce qu'on ne spécifie pas d'objets particuliers ; la contrainte vaut donc pour tous les plans rythmiques (cf. 1° insert).

3.2.1.3. i/i-dst_filt **(internal/internal distances filter = filtre des distances internes les unes par rapport aux autres)**

Ce module permet de contrôler la succession des distances internes en fonction de leur position dans les objets.

Pour mieux saisir d'emblée ce dont il est question, disons que ce module, dans un problème harmonique, inclut la règle d'interdiction qu'on appelle en harmonie classique : interdiction des intervalles parallèles aux voix extrêmes (soprano-alto ou basse-ténor).

- **i/i-dst_filt** généralise cette relation en introduisant les notions :

- d'obligation et d'alternative : **and** et **or** en plus de **not** (en notant que les éléments de syntaxe *****, **ints** et **t** n'ont pas de sens pour ce module **-filt**) ;
- de distance interne pouvant se former entre n'importe quel point d'un objet : on numérote les points formant une distance avec la syntaxe exposée dans **x-dst** ; ainsi :
- **l u** ou **0 u** signifie : distance formée par les points extrêmes d'un objet,
- **l 1** ou **0 1** signifie intervalle formé par les deux premier points,
- **1 3** signifie intervalle formé par les deuxième et quatrième points etc...
- de succession qui permet, au moyen de l'élément de syntaxe **step**, de prendre en compte des alternatives de successions de distances internes d'objets distints et non successifs (voir **3.2.2.1.x-dst-filt** un peu plus loin) ;
- de répétition qui permet de gérer la répétition de distances internes au sein d'ensembles d'objets (voir **x** dans **3.2.1. Distances control**).

Remarque : c'est un souci de méthode qui nous fait présenter les modules en fonction de leur classement dans les menus.

Cependant, pour mieux suivre la présentation du module **i/i-dst_filt**, nous demandons au lecteur de lire en préliminaire les explications relatives au module **x-filt** où sont longuement détaillés les problèmes posés par la notion de distance externe (notion sur laquelle i/i-dst_filt empiète même s'il ne s'y agit pas de distance externe mais de relation entre des distances internes d'objets différents).

• Nous supposons, en tout cas, qu'il procèdera ainsi et proposons directement quelques exemples harmoniques :

(u l (not (13)))

signifie qu'on interdit (cf. **not**) une neuvième mineure (cf. **13**) parallèle pour l'intervalle formé par les voix extrêmes des accords (cf. **l u**) dans chaque couple d'accords successifs de la séquence entière, c'est-à-dire : 1°-2°, 2°-3° etc...).

(0_3 (1 2 (and (8 8))))

signifie qu'on oblige (cf. **and**) deux sixtes mineures (cf. **8 8**) parallèles pour l'intervalle formé par les deuxième et troisième voix à partir du bas des accords (cf. **1 2**) dans chaque triplet d'accords successifs (il y a 2 intervalles spécifiés donc 3 accords) formés par les quatre premiers accords de la séquence (cf. **0_3**), c'est-à-dire 1°-2°-3° et 2°-3°-4°.

(5_24 (step 0 1 (and (or (8 9 4) (8 ? 9))) (not (x x x))))

signifie que du sixième au vingt-cinquième accord (cf. **5_24**), on oblige (cf. **and**) pour tout intervalle formé par les deux voix les plus graves (cf. **0 1**) :

- ou bien (cf. **or**) la succession d'une sixte mineure, d'une sixte majeure et d'une tierce majeure (cf. **8 9 4**) dans tout quadruplet (**3** intervalles donc **4** accords) d'accords distincts (cf. **step** donc quadruplets distincts et non successifs) ;
- ou bien la succession d'une sixte mineure, d'un intervalle quelconque, et d'une sixte majeure (cf. **ints 8 ? 9**), sachant que ces intervalles seront différents d'une part et ne seront ni une sixte mineure ni une sixte majeure d'autre part (cf. **not (x x x)**), dans tout quadruplet (**3** intervalles donc **4** accords) d'accords distincts (cf. **step** donc quadruplet distints et non successifs).

On remarque qu'il est nécessaire que les arguments formant les alternatives comportent le même nombre d'intervalles.

Attention à la double parenthèse lorsque l'expression commence par une voix qui n'est ni **u** ni **l** (cf. 1° insert).

- Cette contrainte peut également avoir un sens dans des problèmes rythmiques ; par exemple :

((0 1 (and (x x))))

signifie que tous les plans rythmiques (cf. **double parenthèse**) commenceront (cf.**0 1**) par une distance identique quelqu'elle soit (cf. **xx**).

Remarque : Dans les problèmes rythmiques, l'usage de cette contrainte est limitée lorsque les unités de distance des objets (des plans) ne sont pas les mêmes.

5° insert : syntaxe des modules _rnw

Les modules **–rnw** admettent un argument à deux termes représentant respectivement :

- un nombre de points (modules **dst**) ou d'objets (modules **pts**)
- un nombre autorisé de renouvellement de distances (modules **dst**) ou de points (modules **pts**), sachant que :
- si ce nombre est absolu, il sera simple
- si ce nombre est un seuil supérieur, il sera précédé de **max**
- si ce nombre est un seuil inférieur, il sera précédé de **min**

Certains de ces modules admettent, en plus de la syntaxe générale, la syntaxe des voix et, pour les problèmes harmoniques, la syntaxe **t**.

3.2.1.4. i-dst_rnw (internal distances renewal = renouvellement des distances internes)

Ce module permet de contrôler le renouvellement des distances internes en fonction du nombre de points des objets.

La contrainte est spécifiée par un argument à deux termes donnant successivement :

- le nombre de points des objets
- le nombre de renouvellement autorisé d'une même distance dans chaque objet.

Ainsi, dans un problème harmonique par exemple :

(0_9 (4 1))

signifie que dans chaque accord de densité 4 (c'est-à-dire comprenant 4 notes et donc 3 intervalles) faisant partie des dix premiers accords de la séquence, chaque intervalle ne pourra être renouveler qu'une seule fois.

On remarque que pour que cette contrainte soit cohérente, il faut que **i-dst** de **csolver** permette au moins 3 intervalles verticaux différents. Si **i-dst** de **csolver** en autorise exactement 3, chaque accord de densité 4 comprendra exactement ces 3 intervalles dans n'importe quel ordre.

Dans un problème rythmique :

(1 (18 4))

signifie que si le deuxième plan rythmique comporte 18 points d'impact, la même distance entre deux points conjoints ne pourra être renouvelée que 4 fois.

i-dst_rnw admet également des listes d'arguments pour définir le renouvellement des distances dans des objets de nombre de points variable ; par exemple :

((3 1) (4 1) (5 2) (6 2))

Indépendamment de son intérêt musical, ce module permet de réduire considérablement la combinatoire possible des distances au sein d'un même objet ; sa programmation est donc un gain de temps pour la machine.

3.2.1.5. x-dst_filt (external distances filter = filtre des distances externes)

Ce module, qui est le symétrique de **i-dst_filt**, permet de spécifier, pour des objets donnés, des configurations de distances externes, c'est-à-dire de distances entre des points d'objets différents.

- Il utilise toutes les syntaxes déjà introduites :

- la syntaxe générale de Situation ;
- la syntaxe propre aux modules **-filt** (en notant toutefois que les éléments *****, **ints** et **t** n'ont ici aucun sens)
- la syntaxe de la notion de voix précédemment expliquée (**l**, **u**, **0** , **1** etc...), syntaxe incluant la notion particulière de voix dans le cas d'une séquence d'objets de densité variable ;

et introduit deux syntaxes complémentaires :

- une syntaxe permettant de mieux préciser les distances entre les points : distances absolues (positives ou négatives) ou relatives (supérieures ou inférieures) ;
- une syntaxe permettant de généraliser la notion de « voix », une « voix » pouvant être ici constituée par la mise en relation de n'importe quel point d'un objet avec n'importe quel point d'un autre objet ;

ces deux syntaxes complémentaires seront étudiées à la fin de ce paragraphe (cf. 6° insert).

- Ainsi, dans un problème harmonique par exemple :

  (u (not (or (8 2) (4 9))))

signifie que dans toute la séquence d'accords, il ne pourra y avoir à la voix supérieure des configurations telles que sixte mineure/seconde majeure ou tierce majeure/sixte majeure, pour tout triplet pris un à un (1°-2°-3° notes, 2°-3°-4° notes etc...).

Cependant, on voit que si **not** est remplacé par **and** :

  (u (and (or (8 2) (4 9))))

et que le nombre d'accords de la séquence est supérieur au minimum nécessaire (ici 3) à utiliser le nombre d'intervalles spécifiés dans les arguments, la contrainte devient incohérente : les accords étant pris deux à deux, le deuxième intervalle ne peut être à la fois, pour la première alternative par exemple, le deuxième terme du premier couple d'intervalles (2) et le premier terme du deuxième couple d'intervalle (8).

Il faut donc avoir recours au petit mot **step** (pas), que l'on placera avant tous les arguments de la syntaxe, pour que celui-ci lise le nombre d'intervalles spécifiés dans chaque argument afin de définir tous les couples, triplets, quatruplets etc... nécessaires à la lecture exhaustive de la contrainte, couples, triplets etc... qu'il considèrera comme distincts et non successifs (par exemple, pour des couples : 1°-2°, 3°-4° et non 1°-2°, 2°-3°) ; **step** rend ainsi la contrainte cohérente et permet la formation de pattern intervalliques.

Par exemple (toujours pour un problème harmonique) :

  (step u (and (or (7 1 ? ?) (? 6 ? 9))))

signifie que pour la voix aiguë formée par tous les accords de la séquence :

- ou bien les quadruplets d'intervalles horizontaux seront formés par une quinte, une seconde mineure et deux intervalles quelconques ;
- ou bien les quadruplets d'intervalles horizontaux seront formés par un intervalle quelconque, un triton, un intervalle quelconque et une sixte majeure ;
- ou bien, encore, la voix aiguë sera constituée par une combinaison des deux types de quadruplets ainsi définis.

Attention : il faut que les arguments formants les alternatives comportent le même nombre d'intervalles.

- Enfin, on peut utiliser **x** pour définir la notion de répétition du même intervalle, nombre de répétition égale au nombre de **x** écrit.

Ainsi, si l'on reprend l'exemple harmonique précédent en y ajoutant (not (x x x x)) :

(step u (not (x x x x)) step u (and (or (7 1 ? ?) (? 6 ? 9))))

signifie la même chose que précédemment sauf que les intervalles quelconques des quadruplets ne pourront pas être les mêmes d'une part, et devront être différents des intervalles de leur quadruplet respectif d'autre part :

- ou bien les quadruplets d'intervalles horizontaux seront formés par une quinte, une seconde mineure et deux intervalles quelconques mais différents d'une part, et ne pouvant pas être une quinte ou une seconde mineure d'autre part ;
- ou bien les quadruplets d'intervalles horizontaux seront formés par un intervalle quelconque, un triton, un intervalle quelconque et une sixte majeure, sachant que ces deux intervalles quelconques devront être différents d'une part, et ne pourront être un triton ou une sixte majeure d'autre part ;
- ou bien, encore, la voix aiguë sera constituée par une combinaison des deux types de quadruplets ainsi définis.

Attention à la double parenthèse si l'expression ne spécifie pas d'accords particuliers et commence par la spécification d'une voix qui n'est ni **u** ni **l** (cf. 1° insert).

- Dans un problème rythmique :

((l (not (* 1/8 *))))

signifie qu'il est interdit d'avoir une distance d'une croche entre les premiers points d'impact de tous les plans rythmiques.

Venons-en maintenant aux deux syntaxes complémentaires.

6° insert : syntaxes complémentaires des modules _filt

- La première syntaxe complémentaire permet de mieux préciser les distances entre les points.

  En effet, comme il s’agit ici de distances externes, c’est-à-dire de relations entre des points d’objets différents, les distances peuvent être absolues ou définies par des seuils ;

- si les distances sont absolues, elles pourront être :
- positives ou négatives ; elles ne seront alors précédées de rien (c’est le cas général jusqu’ici rencontré)
- positives ; elles seront alors précédées de **^** (ce qui désignera, dans un problème harmonique, un intervalle (horizontal) **ascendant** et désignera, dans un problème rythmique, une durée située **après**) ;
- négatives ; elles seront alors précédées de **-** (ce qui désignera un intervalle (horizontal) **descendant** dans un problème harmonique et désignera, dans un problème rythmique, une durée située **avant**) ;
- si les distances sont définies par des seuils, elles pourront être :
- supérieures (ou supérieures ou égales) ; elles seront alors précédées de **>** ou ≥ (ce qui désignera, dans un problème harmonique, un intervalle (horizontal) d’**au moins** la distance désignée et désignera, dans un problème rythmique, une durée d’**au moins** la distance désignée) ;
- inférieures (ou inférieures ou égales) elles seront alors précédées de **<** ou ≤ (ce qui désignera, dans un problème harmonique, un intervalle (horizontal) d’**au plus** la distance désignée et désignera, dans un problème rythmique, une durée d’**au plus** la distance désignée).

  Les notions positives/négatives peuvent se combiner avec les notions de seuil ; par exemple, pour désigner un intervalle ascendant d’une distance au moins égale à une seconde mineure : ≥ **^ 2**.

- La seconde syntaxe complémentaire permet de généraliser la notion de « voix ».

  Il est en effet possible de définir une contrainte entre n’importe point d’un objet avec n’importe quel point d’un autre objet ; il suffit, au moyen du jeu habituel de parenthèses, de spécifier successivement :

- les objets concernés
- les points concernés de ces objets
- la contrainte elle-même

Ainsi, dans un problème rythmique :

(0 1 (2 7 (or (-9/24) (≥ ^ 3/10))))

signifie que le 8° points d'impact (cf.**7**) du deuxième plan rythmique (cf. **1**) devra être distant du 3° point d'impact (cf.**2**) du premier plan rythmique (cf. **0**)

- ou bien de 9 doubles-croches de sextolet avant (cf. **–9/24**)
- ou bien de au moins 3 croches de quintolet après (cf. ≥ **^ 3/10**).

Dans un problème harmonique :

(3 6 (l u (or (^8) (-5))))

signifie que la hauteur la plus basse (cf.**l**) du 4° accord (cf.**3**) devra former avec la hauteur la plus haute (cf.**u**) du 7° accord(cf.**6**)

- ou bien une sixte mineure ascendante(cf.**^8**)
- ou bien une quarte descendante(cf.**-5**).

3.2.1.6. x-dst/reg_filt **(external distances/regions filter = filtre des distances externes en fonction des régions)**

Ce module est le même que **x-dst_filt** à ceci près qu'il permet de filtrer les configurations de distances externes en fonction des régions, c'est-à-dire des registres définis par des fourchettes de hauteurs en code MIDI pour les problèmes harmoniques, et des zones de plans définis par des fourchettes de points d'impact compte tenu d'une unité (d'une impulsion) pour les problèmes rythmiques.

La contrainte est spécifiée par un argument à deux termes donnant successivement :

- la limite inférieure de la région
- la limite supérieure de la région.
  - Ainsi, dans un problème harmonique par exemple :

((36 61 (1 (not (5 ? 7)))))

signifie que pour tous les accords de la séquence, il est interdit (cf. **not**) d'avoir à la deuxième voix à partir du bas (cf. **1**) la succession d'une quarte, d'un intervalle quelconque et d'une quinte (cf. **5 ? 7**), <u>si</u> les notes de cette voix se situent dans un registre compris entre le do1 et le do#3 (cf. **36 61**).

<u>Attention</u> à la double parenthèse si l'expression ne spécifie pas d'accords particuliers et commence par une spécification de registre (cf. 1° insert).

- Dans un problème rythmique :

(2 3 (5/20_14/20 (0 1 (not (or (2/20) (3/20))))))

signifie que le premier point (cf. **0**) du troisième plan (cf. **2**) et le deuxième point (cf. **1**) du quatrième plan (cf. **3**) ne devront pas être séparés d'une distance égale à une croche ou à une croche pointée de quintolet (cf. **(not (or (2/20) (3/20))))** s'ils sont situés dans une région correspondant aux deuxième et troisième temps (cf. **5/20_14/20**).

3.2.1.7. x-dst_rnw **(external distances renewal = renouvellement des distances externes)**

Ce module, qui est le symétrique de **i-dst_rnw**, permet de spécifier le renouvellement des distances externes en fonction d'un nombre d'objets.

Sa fonction est proche de celle de **x-dst_filt** lorsque ce module utilise **step** et **x** (voir plus haut), sauf qu'il ne s'agit pas ici d'objets dont les distances sont spécifiées : **x-dstr_rnw** définit génériquement des seuils de répétition valant pour toutes les distances en fonction d'un nombre donné d'objets.

En outre, **x-dst_rnw** considère des objets successifs et non distincts ; par exemple, pour des couples : 1°-2°, 2°-3° etc... et non 1°-2°, 3°-4° etc...

Après avoir spécifié les points concernés dans les différents objets (spécification incluant la notion de « voix » pour les problèmes harmoniques (l, u, 0, 1 etc...), sachant qu'on applique également ici la notion particulière de voix dans le cas d'une séquence d'accords de densité variable), la contrainte est spécifiée par un argument à deux termes donnant successivement :

- le nombre d'objets à considérer
- le nombre de renouvellement autorisé d'une même distance dans chaque groupement d'objet compte tenu d'une « voix ».

Ce module admet la notion de seuil propre aux modules **_rnw** : **max** et **min** (cf. 5° insert).

Ainsi, dans un problème harmonique par exemple :

(0_9 (u (4 1))

signifie qu'à la voix supérieure faisant partie des dix premiers accords de la séquence, on considèrera chaque fragment de 4 hauteurs (cf. **4**) en s'assurant que dans chacun des fragments de 4 hauteurs consécutives chacun des 3 intervalles n'est utilisé qu'une fois (cf. **1**).

On remarque que pour que cette contrainte soit cohérente, il faut que **x-dst** de **csolver** permette au moins 3 intervalles horizontaux différents. Si **x-dst** de **csolver** en autorise

exactement 3, chaque fragment de 4 notes pris un à un comprendra exactement ces 3 intervalles et toujours dans le <u>même</u> ordre.

3.2.2. Points control **(contrôle des points)**

Ce menu contient quatre modules :

- **pts_filt** (points filter = filtre des points)
- **pts_rnw** (points renewhal = renouvellement des points)
- **x-pts_filt** (external points filter = filtre des points externes)
- **x-pts_rnw** (external points renewhal = renouvellement des points externes).

Avec ces modules, il ne s'agit plus de définir des contraintes sur les distances entre les points mais directement sur les points.

3.2.2.1. pts_filt **(points filter = filtre des points)**

Ce module permet de spécifier des configurations particulières de points dans les objets. Il admet :

- la syntaxe générale des modules **–filt**
- la syntaxe complémentaire des modules **–filt** (expliquée au 6° insert).
  - Ainsi, dans un problème harmonique :
    (not (* 60 *))

signifie qu'il est interdit, dans tous les accords de la séquence, d'avoir la hauteur do3 où qu'elle soit dans les accords.

(≥60 *)

signifie qu'il est obligatoire que la hauteur la plus grave de chaque accord de la séquence soit supérieure ou égale au do3.

(? <60 *)

signifie qu'il est obligatoire que la hauteur immédiatement au-dessus de la plus grave de chaque accord de la séquence soit strictement inférieure au do3.

(7_18 s2 (or (and (* 60 ? 64 **))** **(not** (60 *)))) signifie que tous les <u>deux</u> accords de la sous-séquence allant du huitième au dix-neuvième accord :

- ou bien comprendront la superposition de hauteurs quelconques en n'importe quel nombre, du do3, d'une hauteur quelconque, du mi3 et de hauteurs quelconques en n'importe quel nombre ;
- ou bien ne contiendront pas de do3.

Autrement dit, ces accords ne pourront comprendre le do3 que s'ils comprennent également le mi3 avec une autre hauteur intercalée.

(0 17 (and (or (* 49 * 60 * 62) (46 * 53 * 59 * 64 *)))) signifie que les premiers et dix-huitième accords contiendront obligatoirement :

- ou bien les hauteurs do#2, do3 et ré3 quelque soit ce qui les sépare y compris rien ;
- ou bien les hauteurs sib1, fa2, si2 et mi3 quelque soit ce qui les sépare y compris rien.
- Dans un problème rythmique :
  (0 (≤9/28 *))

signifie que le 1° point du 1° plan rythmique se situera à au plus 9 doubles-croches de septolet du début.

(1 (? ? ? ? 3 *)) signifie que le 5°point du 2° plan se situera à 3 rondes du début.

(* 7/4 ?) signifie que les avant-derniers point de tous les plans rythmiques se situeront à 7 noires du début.

• Rappelons que les contraintes ne sont prises en compte qu'au moment de la recherche de solutions où elles interviennent.

Dans le dernier exemple harmonique, la contrainte portant sur le premier accord ne pose aucun problème : parce que c'est le premier, Situation peut la prendre en compte comme si c'était une simple donnée. En revanche, la contrainte portant sur le dix-huitième est particulièrement forte : se situant en plein milieu du réseau de contraintes, elle risque fort de faire piétiner longtemps Situation au dix-septième accord, car la probabilité de tomber sur une disposition de hauteurs aussi spécifique est, c'est le moins que l'on puisse dire, faible (on peut même attendre longtemps avant de s'entendre dire, le cas échéant, qu'il n'y a pas de solution).

En cas de piétinement (on s'en rendra compte en suivant le déroulement du calcul de la solution dans la fenêtre « lisp »), on pourra :

- ou bien adopter une attitude radicale en interrompant le calcul (drague-cliquer « break »), taper **17** ou **17 t** dans la fenêtre optionnelle **x-sol** de **csolver**, puis drague-cliquer « continue » ; à ce moment là, le calcul se poursuivra à partir du dix-huitième accord en tenant compte ou non des contraintes de cet accord selon que l'on aura ou non fait suivre le **17** de **t** (cf. **3.1.1.8. x-sol**) ;
- ou bien adopter une attitude plus fine en réglant l'ambitus en sorte qu'il agisse comme un entonnoir dirigeant la séquence vers le dix-huitième accord tel qu'on l'a décrit ; à ce moment là, il faudra bien veiller aux autres contraintes : imaginons que **x-dst** de **csolver** n'autorise pour cet accord que des tierces mineures à la voix supérieure et que le module **x-pts_prof**, que l'on verra toute à l'heure, impose un mouvement contraire à cette voix pour chaque accord ; il faudra alors laisser une marge d'ambitus au-dessus du dix-huitième accord au moins égale à cet intervalle.
- Rappelons enfin que le module **pts_filt**, comme toutes les contraintes, permet d'agir sur les bases de données comme un outil d'analyse. Si, en effet, une base de données est connectée à l'entrée

data de **csolver**, **pts_filt** permet de dire si la contrainte qu'il spécifie vaut ou non pour des accords de cette base de données ; autrement dit, de sélectionner les accords de la base de données comportant les configurations de hauteurs qu'il spécifie.

3.2.2.2. pts_rnw (points renewhal = renouvellement des points)

Ce module permet de contrôler le renouvellement global des points en fonction d'un nombre donné d'objets conjoints. Le calcul tient compte du nombre de points des objets.

On spécifie la contrainte par un argument à deux termes donnant successivement :

- le nombre d'objets à considérer
- le nombre autorisé de points répétés pour l'ensemble des points de ces objets :
- si ce nombre est absolu, il sera simple
- si ce nombre est un seuil désignant un maximum, il sera précédé de **max**
- dans les problèmes harmoniques, si on veut tenir compte du redoublement aux octaves des points (des hauteurs), ce nombre sera suivi de **t**.
- Ainsi, dans un problème harmonique :
  (4 0 t)

signifie que la séquence d'accords entière sera examinée par quadruplets successifs (1°-2°-3°-4°, 2°-3°-4°-5° etc...) et que chaque quadruplet ne devra comporter aucune hauteur répétée en tenant compte des redoublements aux octaves de ces hauteurs.

(6_8 (3 max 3))

signifie que l'unique triplet d'accords du sous-ensemble 6_8 de la séquence (7°-8°-9°) comprendra au maximum 3 hauteurs répétées (il pourra donc en comporter 0, 1, 2 ou 3) sans compter les redoublements aux octaves (c'est-à-dire que les hauteurs ré3 et ré4, par exemple, seront comptées comme différentes).

- Dans un problème rythmique :
  (5 0)

signifie que dans tous les plans rythmiques on considèrera les quintuplets de points successifs (1°-2°-3°-4°-5°, 2°-3°-4°-5°-6° etc...) et que ceux-ci ne seront pas identiques ; autrement dit, les points ne tomberont jamais ensemble.

3.2.2.3. x-pts_filt **(external points filter = filtre des points externes)**

<u>Remarque</u> : un point, bien sûr, ne peut être externe : il appartient à un objet et est donc interne. Entendons donc « points externes » dans le sens de « points considérés dans la dimension externe ».

Ce module admet :

- la syntaxe générale des modules **-filt** (sachant que *****, **ints** et **t** n'ont ici aucun sens),
- la syntaxe des voix (cf. **x-dst** de **csolver**),
- la syntaxe complémentaire des modules **–filt**.
  - Ainsi, dans un problème harmonique :
    (not (60))

signifie qu'il est interdit d'avoir la hauteur do3 à la voix supérieure formée par tous les accords de la séquence (rappelons que lorsque aucune voix n'est spécifiée, il s'agit toujours de la voix supérieure).

(step 0 (and (or (? 60 ? 64) (61 63 67 71))))

signifie que la voix formée par la note la plus grave de tous les accords de la séquence sera considérée par quadruplets distincts et que ces quadruplets :

- ou bien comprendront la succession d'une hauteur quelconque, du do3, d'une hauteur quelconque et du mi3
- ou bien comprendront la succession do#3, ré#3, sol3 et si3
- Dans un problème rythmique :
  ((3 (not (≥5/4))))

signifie que les quatrièmes points de chaque plan rythmique ne pourront pas se situer sur ou après la cinquième noire à partir du début.

3.2.2.4. x-pts_rnw **(external points renewhal = renouvellement des points externes)**

Entendons là encore « points externes » dans le sens de « points considérés dans la dimension externe ». Quant à l'usage du mot « voix », nous renvoyons à ce que nous en avons dit à propos de **x-dst** et **x-dst_filt.**

Ce module permet de contrôler le renouvellement global des points d'une ou de plusieurs « voix » en fonction d'un nombre donné d'objets conjoints. Ce module est le même que **pts_rnw** sauf que les objets sont considérés par « voix » et qu'il n'y a donc plus de notion de « densité ».

On spécifie la contrainte par un argument à deux termes donnant successivement :

- le nombre d'objets à considérer pour une ou des « voix » données

- le nombre autorisé de points répétés pour l'ensemble des points de cette ou ces « voix » :
- si ce nombre est absolu, il sera simple
- si ce nombre est un seuil désignant un maximum, il sera précédé de **max**
- dans les problèmes harmoniques, si on veut tenir compte du redoublement aux octaves de ces points (ces hauteurs), ce nombre sera suivi de **t**.
- Ainsi, dans un problème harmonique :
  (u (4 max 2))

signifie que la séquence d'accords sera examinée par quadruplets de hauteurs successives à la voix supérieure (1°-2°-3°-4°, 2°-3°-4°-5° etc...) et que chaque quadruplet devra comporter au maximum 2 hauteurs répétées sans tenir compte des redoublements aux octaves.

ou encore :

(6_12 (0 (7 5 t) 2 (7 5 t)))

signifie que l'unique sextuplet (cf. **6_12** et **7**) de hauteurs successives de la voix inférieure (cf. **0**) ainsi que l'unique sextuplet de hauteurs successives de la troisième voix à partir du bas (cf. **2**) du sous-ensemble 6_12 de la séquence (7°-8°-9°-10°-11°-12°-13° accords) comprendront chacun exactement 5 hauteurs répétées (5 fois la même hauteur) en tenant compte des redoublements aux octaves (cf. **5 t**).

- Dans un problème rythmique :
  (1 (2 0))

signifie que les plans rythmiques seront examinés par couples successifs (1°-2°, 2°-3° etc...) en considérant les troisièmes points de chaque plan, et que ces troisièmes points pris deux à deux ne seront pas synchrones.

3.2.3. profiles control (contrôle des profils)

Ce sous-menu comprend trois modules :

- **x-pts_prof** (external points profile = profil des points externes)
- **x/x-pts_prof** (external/external points profile = profil des points externes les uns par rapport aux autres)
- **bpf-x-pts_prof** (break point function external points profile = profil des points externes déterminé par des courbes)

Avec ces modules, il ne s'agit plus de définir des contraintes sur les distances ou sur les points mais de définir le profil des relations externes au moyen d'une syntaxe (**x-pts_prof** et **x/x-pts_prof)** ou d'une courbe (**bpf-x-pts_prof**).

La notion de « voix » étant paradoxale et n'ayant qu'un sens limité (si ce n'est théorique) dans les problèmes rythmiques, nous ne donnerons pour les modules de *profiles control* que des exemple harmoniques.

Remarque : La prochaine version de Situation (novembre 99 probablement) comportera des modules *profile control* pour la dimension interne qui, en revanche, seront d'un grand intérêt pour les problèmes rythmiques.

3.2.3.1. x-pts_prof **(external points profile = profil des points externes)**

La contrainte peut être spécifiée de deux manières :

- nominativement, par un argument pouvant comporter plusieurs termes définissant le profil exact que l'on désire, profil s'appliquant alors à une ou des « voix » dans un nombre défini d'objets ;
- par un seuil désignant à partir de combien de distances le profil changera de direction, profil s'appliquant alors à une ou des « voix » dans un nombre indéfini d'objets.
- Si on définit nominativement le profil, l'argument pourra comprendre autant de termes que l'on veut, sachant que ces termes seront :
- des nombres simples pour désigner un nombre absolu de directions ascendantes (rappelons que l'équivalent de « directions ascendantes » est « situés après » pour les problèmes rythmiques),
- des nombres précédés de **-** pour désigner un nombre absolu de directions descendantes (« situés avant »),

et que ces nombres définiront le profil d'une « voix » dans un nombre défini d'objets, nombre défini d'objets que Situation déduira en additionnant le nombre de distances spécifiées dans l'argument plus un. Ces objets constitueront donc des couples, triplets etc... considérés comme distincts et non successifs.

Remarquons que cette manière de procéder permet de définir des pattern de profil.

Ainsi, dans un problème harmonique :

(u (2 -3))

signifie que la voix supérieure de la séquence d'accords entière sera considérée par sextuplets d'accords (cf. **2 -3** : 2 intervalles ascendants (cf. **2**) suivi de 3 intervalles descendants (cf. **-3**) font un total de 5 intervalles et impliquent donc 6 accords), sextuplets d'accords distincts (1°-2°-3°-4°-5°-6°, 7°-8°-9°-10°-11°-12°) et non successifs, et comportant chacun deux intervalles ascendants suivis de trois intervalles descendants.

- Si l'on définit le profil par un seuil, l'argument comprendra un seul terme, sachant que ce terme pourra être :

- un nombre simple désignant un seuil absolu
- un nombre précédé de **max** désignant un seuil maximum

et que ce nombre, correspondant au nombre de distances de même direction, définira un seuil à partir duquel le profil d'une ou de plusieurs « voix » dans un nombre indéfini d'accords changera de direction ; il n'y a donc pas de notion de couples, triplets etc..., et la contrainte s'applique à l'ensemble des objets de manière successive.

Cette limitation s'explique par ceci que si plusieurs nombres représentaient des seuils, on aurait affaire à un très grand nombre de combinaisons possibles. Situation pourrait bien sûr en venir à bout, mais au prix d'un temps de calcul nous paraissant disproportionné par rapport à l'intérêt des solutions, lesquelles seraient alors choisies aléatoirement parmi ce grand nombre de possibilités.

- Ainsi, dans un problème harmonique :
  (0_9 (1 (3)) 3_14 (3 (max 4)))

  signifie que la deuxième voix à partir du bas des 10 premiers accords de la séquence changera de direction exactement tous les trois intervalles, c'est-à-dire tous les quatre accords, et que la quatrième voix à partir du bas, du quatrième au quinzième accords de la séquence, changera de direction au maximum tous les quatre intervalles, c'est-à-dire tous les cinq accords (on aura donc au maximum 4 intervalles de même direction).
- Dans un problème rythmique :
  (0_4 (1 (5)))

  signifie que les plans rythmiques (les 5, cf. **0_4**) commenceront les uns après les autres en commençant par le premier (il s'agit en effet des premiers points de chaque plan (cf. **l**)).

3.2.3.2. x/x-pts_prof **(external/external points profile = profil des points externes les uns par rapport aux autres)**

Ce module permet de contrôler la direction d'une « voix » par rapport à une autre. Il généralise la notion, bien connue en harmonie classique, de mouvement parallèle entre les voix.

Ce module reprend une partie de la syntaxe expliquée pour **x-pts_prof**, en prenant bien sûr une signification différente, et désigne les « voix » à considérer l'une par rapport à l'autre comme dans **i/i-dst_filt** : **l u** = voix extrêmes, **1 3** : la deuxième et la quatrième à partir du bas etc...

On spécifie la contrainte par un argument à un terme donnant le nombre de directions parallèles autorisées pour un couple de voix défini, sachant que ce terme pourra être :

- un nombre simple désignant un seuil absolu

- un nombre précédé de **max** désignant un seuil maximum

Ainsi :

(0_9 (l u (max 3)) 3_14 (1 2 (0)))
signifie :

- que les voix extrêmes des 10 premiers accords de la séquence ne pourront pas être parallèles plus de trois fois, autrement dit elles pourront éventuellement prendre une direction contraire après avoir formé 1 ou 2 intervalles parallèles mais devront obligatoirement prendre une direction contraire après avoir formé 3 intervalles parallèles ;
- et que les deuxième et troisième voix à partir du bas, du quatrième au quinzième accord de la séquence, seront parallèles zéro fois exactement, c'est-à-dire qu'elles avanceront toujours par mouvement contraire.

3.2.3.3. bpf-x-pts_prof (break point function external points profile = profil des points externes déterminé par des courbes)

Ce module permet de définir le profil d'une « voix » à partir d'une courbe. Sa fonction est, si l'on veut, la même que **x-pts_prof**, sauf qu'il ne connaît pas la notion de seuil.
bpf-x-pts_prof comporte trois entrées :

- la première permet de recevoir un **bpf** où l'on aura dessiné la courbe du profil ;
- la deuxième permet d'inscrire le numéro de la « voix » concernée : 0, 1, 2 etc... (attention, ce module n'accepte ni **l** ni **u**) ;
- la troisième permet de spécifier si l'on interprète les parties plates de la courbe comme une absence de contrainte (dans ce cas on tape nil dans la fenêtre) ou comme une contrainte de profil plat (c'est-à-dire de notes répétées dans un problème harmonique) ; dans ce cas on tape **t** dans la fenêtre (rappelons, en effet, que les **bpf** de OpenMusic ne peuvent fragmenter une courbe : ses différentes parties sont donc reliées par un trait horizontal qu'il faut interpréter) ;

bpf-x-pts_prof comporte en option une quatrième entrée qui permet d'inscrire les objets sur lesquels agira la courbe afin d'échantillonner celle-ci et de la mettre à l'échelle ; cette fenêtre accepte la syntaxe générale, ainsi : 0_9 signifie 10 objets, les dix premiers, et 10_39s3 signifie dix objets, un sur trois du onzième au quarantième.

Cette programmation d'un **bpf** et de **bpf-x-pts_prof** est à faire pour chaque « voix » que l'on veut contraindre par une courbe.

3.3. user-constraints (contraintes de l'utilisateur)

Contrairement aux **standard constraints** dont les fonctions sont prédéfinies, les **user-constraints** sont à définir par l'utilisateur ; elles s'adressent donc à un utilisateur plus expérimenté.

Le menu « user constraints » comporte trois modules spécifiques à Situation :

- **user-cnstr** (user defined constraints = contrainte définie par l'utilisateur)
- **prev-instances** (previous instances = cas précédents)
- **wprev-instances** (previous instances = cas précédents avec **wsolver**)

3.3.1. user-cnstr ***(user defined constraints = contraintes définie par l'utilisateur)***

Ce module, qui se connecte à l'entrée **cnstr** de **csolver** ou au module **list** du menu « kernel » de OpenMusic s'il y a d'autres contraintes, comprend deux entrées : **const** et **exp**.

• **const** (constraint = contrainte) est un prédicat (c'est-à-dire une fonction qui permet de dire si le résultat du patch est vrai ou faux). C'est en général un sous-patch de OpenMusic verrouillé en mode lambda (voir manuel de référence de OpenMusic).

• **exp** (expression) est une liste qui spécifie les objets subissant la contrainte. Cette liste combine par un jeu de parenthèses :

- des régions du problème ;
- un choix d'objets dans ces régions (choice = choix). Il peut y avoir trois types de choix, désignés par **a**, **f** et **s** :
- **a** (all combinations = toutes les combinaisons) : la contrainte porte sur toutes les combinaisons différentes d'objets d'une région donnée ;
- **f** (fixed and ordered chords = objets fixés et ordonnés) : la contrainte porte sur les objets désignés dans une région et dans l'ordre où ils sont désignés ;

- **s** (sub-sequence = sous-séquence) : la contrainte porte sur une ou plusieurs sous-régions de la région ; contrairement à **a** et **f**, **s** peut être associé à un nombre qui définit un pas : ce pas est de 1 par défaut, c'est-à-dire que **s** est équivalent à **s 1**.

Chaque choix (**a**, **f** ou **s**) peut être suivi de **i**. Dans ce cas, le prédicat (c'est-à-dire la fonction définie dans **const**), doit avoir deux fois plus d'arguments que d'objets que contraint le prédicat :

- les numéros des objets (c'est-à-dire les numéros d'ordre qu'ils occupent dans le problème), qui constituent le premier paramètre ;
- les objets eux-mêmes, qui constituent le deuxième paramètre.

Par exemple :

(0_10 (s))

équivalent à

(0_10 (s 1))

signifie que la contrainte (le prédicat ou fonction) s'applique sur toutes les sous-régions de la région en fonction du nombre d'arguments de la contrainte.

Si la contrainte contient deux arguments, 0_10 sera examiné ainsi : 1°-2°, 2°-3° etc...

(0_10 (s 2))

signifie que la contrainte (le prédicat ou fonction) s'applique sur toutes les sous-régions de la région en fonction du nombre d'arguments de la contrainte.

Si la contrainte contient trois arguments, 0_10 sera examiné ainsi : 1°-2-3°, 3°-4°-5° etc…

(0_10 (s 2 i))

signifie la même chose que précédemment sauf que l'on spécifie également le numéro d'ordre des objets.

7° insert : Explications complémentaires sur a, f, s et i.

- s (= séquence)

Dans **user-cnstr**, il est nécessaire de préciser la manière dont les objets vont être considérés par la contrainte. La machine regarde d'abord le patch branché sur **user-cnstr**. Là, il trouve le nombre d'arguments de la contrainte. Si, par exemple, le patch a deux entrées, cela veut dire une contrainte entre deux objets. Mais, lesquels ?.

Si l'entrée dans **user-cnstr** est :

(1_15 (s 2))

il faut appliquer la contrainte aux objets numérotés de 1 à 15. Cependant, il y a plusieurs façons de considérer des couples d'objets parmi les objets numérotés de 1 à 15 : **(s 2)** indique qu'ils sont pris en séquence avec des déplacements de pas. La contrainte porte donc sur les couples d'objets :

(1,2)
(3,4)
(5,6).....etc…

De même, si l'entrée dans **user-cnstr** est :

(1_15s3 (s 2))

équivalent à :

(1 4 7 10 13 (s 2))

les couples d'objets à contraindre seront :

(1,4)
(7,10)

- f (= fixed)

Avec **f**, la contrainte porte exactement sur les objets indiqués. Par exemple :

(13 9 72 (f))

indique une contrainte à trois arguments portant sur les objets 13, 9 et 72, dans l'ordre.

En fait, si le patch branché sur **user-cnstr** avait exactement 3 arguments, **f** pourrait très bien être éliminé car l'exemple précédent serait alors équivalent à :

(13 9 72 (s 1))

- a (= all)

Avec **a**, les objets sont pris ici dans toutes les combinaisons (croissantes) possibles.

Par exemple, si le patch a deux arguments et si l'entrée de **user-cnstr** est :

(1_15 (a))

la contrainte s'applique aux couples d'objets :

(1, 2)
(1, 3) ......
(1, 15)
(2, 3)
(2, 4)

(2, 15) ....

(3, 4) ....

(3, 5) ....

(14, 15)

- i (= include object indexes)

Il y a une autre complication. On pourrait en effet bel et bien vouloir utiliser non pas seulement les objets mais aussi leur numérotation ; par exemple, dire que deux objets séparés par 4 places dans la séquence doivent avoir entre eux une distance supérieure à 4. Pour pouvoir contraindre ce genre de choses, le patch a besoin de connaître non pas seulement les objets mais aussi leur position dans la séquence. C'est l'affaire de **i** ; ainsi, l'entrée :

(1_15 (s 2 i))

signifie que le patch branché sur **user-cnstr** doit s'attendre à recevoir des indices (en premier) et des objets. Si, par exemple, ce patch a 4 arguments, cela veut dire que les deux premiers sont des indices pour la position des objets (dans l'ordre), lesquels objet constituent les deux autres arguments.

La contrainte portera alors sur les couples précédemment mentionnés, c'est-à-dire :

(1,2)

(3,4)

(5,6).....etc…

mais le patch recevra également la position en argument ; autrement dit :

Argument 1 : 1

Argument 2 : 2

Argument 3 : l'objet dans la position 1 de la séquence

Argument 4 : l'objet dans la position 2 de la séquence

et ensuite,

Argument 1 : 3

Argument 2 : 4

Argument 3 : l'objet dans la position 3 de la séquence

Argument 4 : l'objet dans la position 4 de la séquence

etc…

3.3.2. prev-intances ***(previous instances = cas précédents)***

Ce module permet de prendre en compte dans la contrainte les objets du problème ayant déjà été calculés. Il peut intervenir en toute place du patch.

Pour plus de précisions, voir **Tutoriel Sit 7b**.

*3.3.3. wprev-intances **(wprevious instances = cas précédents avec wsolver)***

Ce module est le même que **prev-instances** mais marche avec **wsolver** (et non **csolver**).

3.4. generic problem (problème générique)

Ce menu comprend quatre modules :

- **variables-domains** (variables-domains = domaines des variables)
- **generic-cnstr** (generic constraints = contrainte générique)
- **done-instances** (cas terminés)
- **current-variable** (= variable courante)

Ces quatre modules, qui s'adressent à un utilisateur expérimenté, illustrent la généralité de *Situation* : ils permettent de résoudre, sur des domaines finis, une grande variété de problèmes de résolution de contraintes, musicaux ou autres.

*3.4.1. variable-domains (**= domaines des variables)***

Ce module permet de définir le domaine sur lequel agiront les contraintes ; il se connecte à l'entrée **data** de **csolver**.

variable-domains contient des listes de listes et les sous-listes se présentant dans l'ordre des variables. Voir **Tutoriel 8a**.

*3.4.2. generic-cnstr (**= contrainte générique)***

Ce module permet de recevoir, dans sa première entrée, une **user constraint** et de la gérer en fonction des variables (2° entrée) et des modalités de choix des variables (3°entrée) (cf. **user constraints**). **Voir Tutoriel 8a.**

3.4.3. done-instances (= cas terminés)

3.4.4. current-variable (= variable courante)

3.5. utilities (modules utilitaires)

Le menu utilities comporte neuf modules :

- **ch-sol** (chords solution = solution d'accords)
- **rtm-sol** (rhythmic solution = solution rythmique)
- **part-sol** (partial solution = solution partielle)
- **default-fill** (default filling = remplissage par défaut)
- **bpf-ambdef** (break point function ambitus definition = courbes de définition d'ambitus)
- **all-permutations** (= toutes permutations)
- **combinations** (= combinaisons)
- **comb-with-reps** (combinations with repetition = combinaisons avec répétitions)
- **all-replacements** (= remplacements)

3.5.1. ch-sol ***(chords solution = solution d'accords)***

Ce module permet de construire une solution d'accords en code MIDI pour l'édition.
Dans **csolver**, en effet, le problème est résolu indépendamment de toute instanciation : ce n'est qu'à la sortie que le résultat est interprété, avec **ch-sol** en terme de séquence harmonique.

ch-sol reçoit en entrée **csolver** et connecte sa sortie à l'entrée « chords » d'un module éditeur de OpenMusic : **voice ou chord-seq** par exemple.
ch-sol possède une entrée optionnelle permettant de transposer la séquence en Midics, par exemple 2400 pour transposer de deux octaves.

3.5.2. rtm-sol ***(rhythmic solution = solution rythmique)***

Ce module permet de construire une solution rythmique pour l'édition.
Lire le **3° insert** et noter que **rtm-sol** doit être branché dans un **poly** éditeur.

3.5.3. part-sol ***(partial solution = solution partielle)***

Ce module permet d'évaluer une solution partielle après avoir clické « break » ou « abort » dans le menu *lisp*.
Ce module connecte sa sortie à l'entrée des modules **ch-sol** ou **rtm-sol**.
Lire ce document : **3.1.1.8 x-sol** et paragraphe précédent.

3.5.4. default-fill ***(default filling = remplissage par défaut)***

Ce module permet d'alléger la syntaxe en déduisant les valeurs non spécifiées des valeurs spécifiées.
Plus précisément, **default-fill** fait prendre :

- les valeurs du départ de l'interpolation (premier argument) aux objets situés avant les objets faisant l'objet de cette interpolation
- les valeurs de l'arrivée de l'interpolation (deuxième argument) aux objets situés après les objets faisant l'objet de cette interpolation

default-fill se branche à l'entrée souhaitée de **csolver.**
Lire ce document : **3.1.1.2. p-pts**.

3.5.5. bpf-ambdef ***(break point function ambitus definition = courbes de définition d'ambitus)***

Ce module permet de définir un ensemble de points possibles (**p-pts**) avec des courbes dessinées dans un module **bpf-lib** de openMusic.

Il reçoit la deuxième sortie d'un module **bpf-lib** en entrée et connecte sa sortie à **p-pts** de **csolver**, et comporte une entrée optionnelle permettant d'inscrire le nombre d'objets à considérer pour échantillonner les courbes et les mettre à l'échelle.

3.5.6. all-permutations (= toutes permutations)

Ce module permet de calculer toutes les permutations d'une liste.

3.5.7. combinations (= combinaisons)

Ce module permet de calculer toutes les combinaisons de deux listes.

3.5.8. comb-with-reps (combinations with repetition = combinaisons avec répétitions)

Ce module permet de calculer toutes les combinaisons avec répétitions d'une liste. Le nombre de répétitions s'inscrit dans la deuxième entrée du module.

3.5.9. all-replacements (= tous remplacements)

4. Menu de Situation

Situation 3.0

solvers

csolver
wsolver
add-csolver

standard constraints

distances control

i-dst_filt
i-dst/reg_filt
i/i-dst_filt
i-dst_rnw
x-dst_filt
x-dst/reg_filt
x-dst_rnw

pitches control

pts_filt
pts_rnw
x-pts_filt
x-pts_rnw

profiles control

x-pts_prof
x/x-pts_prof
bpf- x-pts_prof

user constraints

user-cnstr
prev-instances
wprev-instances

generic problem

variable-domains
generic-cnstr

done-instances

current-variables

utilities

ch-sol

rtm-sol

part-sol

default-fill

bpf-ambdef

all-permutations

combinations

comb-with-reps

all-replacements

4. Tutoriel

Les Tutoriels se trouvent dans le dossier Situation 3.0 Tutoriels du dossier *OMsituation3.0* de *User Library* (faire glisser l'icône sur OMWorkspace) :

- Tutoriels 1 à 3 : **standard constraints** dans problèmes harmoniques
- Tutoriels 4 à 6 : **standard constraints** dans problèmes rythmiques
- Tutoriel 7 : user constraints
- Tutoriel 8 : generic problems

Il existe en outre de nombreux exemples d'application musicale de Situation 3.0 dans le dossier Situation 3.0 examples du dossier *OMsituation3.0* de *User Library* (faire glisser l'icône sur OMWorkspace). Contrairement aux Tutoriels, ces exemples ne sont pas commentés dans cette documentation mais dans les patchs eux-mêmes.

4.1. Tutoriel Sit 1

- Tutoriel Sit 1

Ce patch montre les différentes constructions possibles pour l'édition harmonique.

4.2. Tutoriel Sit 2

Ces neuf patchs montrent le raffinement progressif d'un problème harmonique et l'interdépendance des contraintes.

Tutoriel Sit 2a Evaluation avec les valeurs par défaut de **csolver** :

- **n-obj** = 8 : 8 accords
- **p-pts** = (48_72) : do2-do4
- **n-pts** = (3) : densité 3
- **i-dst** = (2 7) : intervalles verticaux autorisés : seconde majeure et quinte juste
- **cnstr** = nil : pas de contrainte.
- **x-dst** = (1 3) : intervalles horizontaux autorisés pour la voix supérieure : seconde mineure et tierce mineure

La solution proposée (il en existe bien sûr un très grand nombre) montre que par souci de rapidité de calcul Situation se contente de transposer le même accord en fonction des intervalles horizontaux autorisés de la voix supérieure.

On peut remédier à cela de différentes manières.

- Tutoriel Sit 2b

Par exemple en imposant des intervalles horizontaux pour la voix inférieure avec **x-dst**. S'ils sont différents de ceux de la voix supérieure (cf**. l (2 4)**), Situation ne pourra plus transposer le même accord.

- Tutoriel Sit 2c

Ou en interdisant avec le module **i/i-dst_filt** une quinte parallèle (cf. **7 7**) aux deux voix les plus graves (cf. **0 1**).

- Tutoriel Sit 2d

Remarquons que si l'on interdit la quinte parallèle à tous les couples de voix (c'est-à-dire sans spécifier un couple particulier contrairement à l'exemple précédent), il y a pas mal de chances pour que Situation choisisse de ne pas utiliser de quinte.

- Tutoriel Sit 2[e]

On peut alors, avec le module **i-dst_rnw**, forcer Situation à utiliser dans chaque accord les deux intervalles verticaux autorisés (cf. **3 1 =** dans un accord de densité 3, il ne pourra y avoir que 1 seul intervalle de même type ; en effet, comme il faut deux intervalles pour former un accord de densité 3, la seconde majeure et la quinte juste seront forcément utilisées).

- Tutoriel Sit 2f

On peut maintenant inscrire cette séquence dans un ambitus ascendant.

On remarque que la pente de celui-ci étant assez raide - de do2-do3 (cf. **48 60**) pour le 1° accord à do4-do5 (cf. **72 84**) pour le dernier - la voix supérieure est obligée d'utiliser son intervalle le plus grand : la tierce mineure.

- Tutoriel Sit 2g

On peut d'ailleurs s'en assurer : si l'on interdit avec **x-dst_filt** la succession de deux tierces mineures (cf**. (not (3 3)**) à la voix supérieure (cf. **u**)), Situation dira qu'il n'y a pas de solution (cf. « there is no solution » dans la fenêtre Lisp, laquelle indique d'ailleurs que l'on peut en trouver une si l'on modifie l'ambitus : cf. « (change_level (append (make_ambitus)... »).

- Tutoriel Sit 2h

Si on impose un ambitus toujours ascendant mais moins raide - de do2-do3 (cf. **48 60**) pour le 1° accord à fa#3-fa#4 (cf. **66 78**) pour le dernier - une solution pourra être trouvée malgré la contrainte **x-dst_filt**.

- Tutoriel Sit 2i

Enfin, corsons un peu le problème en interdisant non seulement les quintes parallèles mais également les secondes mineures parallèles avec **i/i-dst_filt** (cf. (**not (7 7)**)), et demandons 3 solutions différentes (cf. **n-sol** = 3).

La fenêtre Lisp indique que Situation en a trouvées deux mais qu'il n'y en a pas d'autres.

En les éditant, on constate que les deux uniques solutions sont très proches l'une de l'autre : les hauteurs des voix extrêmes sont les mêmes et seule change la voix intérieure qui permute les deux intervalles qu'elle forme avec les autres voix.

4.3. Tutoriel Sit 3

Ces cinq patchs donnent des exemples plus axés sur la dimension mélodique.

- Tutoriel Sit 3a

Dans ce patch, on cherche une mélodie de 12 hauteurs (cf. **n-obj = 12**) toutes différentes - cf. **pts_rnw** : **12** hauteurs avec exactement **0** répétition et en tenant compte des redoublements à l'octave (cf. **t**) - et comportant des intervalles compris entre la seconde mineure et la septième majeure (cf. **(1_11)**).

Autrement dit, une série dodécaphonique.

Si on veut également imposer le total intervallique, plusieurs possibilités s'offrent.

- Tutoriel Sit 3b

Si on désire le total intervallique sans ordre particulier des intervalles, on peut utiliser **x-dst_rnw** qui permet d'obliger que pour la séquence de **12** accords (ou notes puisque la densité est **1**), chaque intervalle autorisé par **x-dst** (cf. **(1_11)**) ne soit utilisé que **1** fois (cf. **(12 1)**). Comme 11 intervalles sont autorisés, c'est exactement le nombre nécessaire.

Puisqu'on garde la contrainte **pts_rnw**, on obtient à la fois le total intervallique et le total chromatique.

- Tutoriel Sit 3c

Si on désire un total chromatique ordonné, plusieurs solutions sont possibles.

On peut spécifier un ordre particulier des 11 intervalles différents dans **x-dst** de **csolver** en forçant cet ordre avec **f** (rappelons que *sans* **f** les intervalles sont simplement des intervalles possibles, et que *avec* **f** ils sont obligatoires et dans l'ordre indiqué).

Par exemple :

(f (3 2 6 10 7 1 5 4 8 11 9))

- Tutoriel Sit 3e

On peut enfin utiliser, du moins en théorie, le module **x-dst_filt** avec **step** (pour que la séquence soit considérée comme un pattern).

Mais il y a là un problème de durée de calcul : en effet, contrairement à **x-dst** qui agit dès le départ pour la définition du domaine (la succession d'intervalles est donc quasiment une donnée), **x-dst_filt** est une contrainte et agit donc après la définition du domaine.

x-dst_filt devra donc filtrer ce domaine jusqu'au moment de trouver la solution. Comme la succession d'intervalles exigée est longue, l'exploration du domaine sera long car chaque nouvelle exploration sera déterminée par la rencontre d'une impossibilité, impossibilité qui interviendra le plus souvent, qui plus est, en fin de séquence.

On peut certes contourner le problème en fragmentant la contrainte et écrire par exemple dans **x-dst_filt** :

(0_3 (step u (and (3 2 6))) 3_6 (step u (and (10 7 1)))
6_9 (step u (and (5 4 8))) 9_11 (step u (and (11 9))))

au lieu de :

(step u (3 2 6 10 7 1 5 4 8 11 9))

qui est strictement équivalent.

Mais, indépendamment de la lourdeur de la syntaxe, le calcul demeurera tout de même plus lent (environ 2 minutes au lieu de 10 secondes sur un Power Mac 7500).

Attention : toutes les combinaisons de tous les intervalles ne sont pas compatibles avec le total chromatique. Lorsqu'il n'y a pas de solution, Situation doit attendre d'avoir explorer toutes les combinaisons pour le dire. Avec **x-dst_filt**, cela peut prendre une heure contre 20 secondes avec **x-dst**! (toujours sur Power Mac 7500).

Il vaut donc mieux réserver **x-dst_filt** avec **step** pour d'autres types de problèmes (cf. ce document : **x-dst_filt**, **i/i-dst_filt**, **x-pts_filt** par exemple).

4.4. Tutoriel Sit 4

Ces trois patchs montrent de façon élémentaire le fonctionnement de Situation dans un problème monorythmique.

- Tutoriel Sit 4a

Dans l'ordre de ses entrées, **csolver** demande 1 plan rythmique, dans un « ambitus de temps » de 20 doubles-croches de quintolet échantillonnées à la double-croche de quintolet, et contenant 7 points d'impact distants les uns des autres d'une durée comprise entre 2 et 5 doubles-croches de quintolet (il

n’y a pas de contrainte et, bien sûr, pas de spécifications de distance externe puisqu’il n’y a qu’un seul plan rythmique.

Pour l’affichage de la solution, **rtm-sol** demande d’inscrire le résultat dans une mesure à 4/4, avec un tempo de 60 à la noire, et une unité d’impulsion valant une double-croche de quintolet ; en outre, les points d’impact devront être instanciés avec des accords importés.

En l’absence de plus de précisions, les solutions sont très nombreuses (probablement une solution différente à chaque lancement du calcul).

- Tutoriel Sit 4b

La contrainte **pts-filt** demande que le premier point (cf. « rien » avant **0**) ait la valeur **0** (c’est-à-dire la première double-croche de quintolet) et que l’avant-dernier point (cf. une fois **?** à droite donc un avant la fin) ait la valeur **17/20** (c’est-à-dire la dix-huitième double-croche de quintolet, ce qui veut dire que le dernier point n’a plus que deux solutions : la dix-neuvième ou la vingtième double-croche de quintolet).

- Tutoriel Sit 4c

La contrainte **i-dst_filt** interdit maintenant la succession de 2 croches de quintolet, où qu’elles soient placées (cf. **not (* 1/10 1/10 *)**).

4.5. Tutoriel Sit 5

Ces six patchs montrent de façon élémentaire le fonctionnement de Situation dans un problème polyrythmique.

- Tutoriel Sit 5a

Dans l’ordre de ses entrées, **csolver** demande 2 plans rythmiques, le premier dans un « ambitus de temps » de 10 doubles-croches de quintolet échantillonnées à la double-croche de quintolet, le second dans un « ambitus de temps » de 6 croches de triolet échantillonnées à la croche de triolet, le premier comportant 6 points d’impact, le second 4, le premier admettant des distances de 1 à 3 doubles-croches de quintolet, le second de 1 à 2 croches de triolet (ni contrainte ni spécification de distances externes).

Pour l’affichage de la solution, **rtm-sol** demande d’inscrire le résultat dans une mesure à 2/4, avec un tempo de 60 à la noire, et une unité d’impulsion valant une double-croche de quintolet pour le premier plan et croche de triolet pour le second ; en outre, les points d’impact des deux plans devront être instanciés avec des accords importés.

Là encore, en l’absence de plus de précisions, les solutions sont très nombreuses (probablement une solution différente à chaque lancement du calcul).

- Tutoriel Sit 5b

La contrainte **x-dst_filt** interdit les « unissons » entre les premiers, seconds etc… points d'impact de chaque plan rythmique.

La contrainte **pts_filt** interdit au premier plan rythmique d'avoir un point d'impact sur la valeur 0 (c'est-à-dire sur la première double-croche de quintolet) et sur la valeur 1/4 (c'est-à-dire sur le deuxième temps).

- Tutoriel Sit 5c

Maintenant, la contrainte **pts_filt** interdit aux deux plans rythmiques d'avoir un point d'impact sur la valeur 0 (c'est-à-dire sur la première double-croche de quintolet pour le premier plan et sur la première croche de triolet pour le deuxième plan) et sur la valeur 1/4 (c'est-à-dire sur le deuxième temps).

- Tutoriel Sit 5d

On rajoute à la contrainte **pts_filt** du *Tutoriel 5c* une contrainte **i-dst_rnw** limitant le premier plan rythmique (cf. **0**) à 3 répétitions d'une même distance entre deux points (cf. **(6 3)**, 6 parce que 6 points dans le plan) et le second plan (cf. **1**) à 2 répétitions (cf. **(4 2)**, 4 parce que 4 points dans le plan).

- Tutoriel Sit 5[e]

    Une troisième contrainte vient interdire la succession de 2 doubles-croches de triolet au premier plan rythmique.

- Tutoriel Sit 5f

On demande alors 5 solutions différentes obéissant à ces contraintes (cf. **5** dans la 9° entrée de **csolver**).

Le *listener* nous apprend qu'il n'y en a que 4 et on les affiche.

4.6. Tutoriel Sit 6

Ces cinq patchs illustrent l'intérêt d'utiliser les points comme marques temporelles susceptibles d'être instancier par d'autres objets rythmiques ; autrement dit, ils illustrent la dimension hiérarchique de Situation.

- Tutoriel Sit 6a

Dans l'ordre de ses entrées, **csolver** demande 3 plans rythmiques, le premier dans un « ambitus de temps » allant de la 9° à la 63° doubles-croche de septolet (soit 54 doubles-croches de septolet échantillonnées à la double-croche de septolet), le second dans un « ambitus de temps » allant de la 8° à la 54° doubles-croche de sextolet (soit 46 croches de sextolet échantillonnées à la croche de sextolet), le troisième dans un « ambitus de temps » allant de la 5° à la 27° croche de triolet (soit 22 croches de triolet échantillonnées à la croche de triolet), le premier comportant 5 points d'impact, le

second et le troisième 4, le premier admettant des distances de 12 à 16 doubles-croches de septolet, le second de 7, 12 et 13 croches de sextolet, le troisième de 6 croches de triolet (ni contrainte ni spécification de distances externes).

Pour l'affichage de la solution, **rtm-sol** demande d'inscrire le résultat dans des mesures à 3/4, avec un tempo de 112 à la noire, et une unité d'impulsion valant une double-croche de septolet pour le premier plan, une double-croche de sextolet pour le second et une croche de triolet pour le troisième ; en outre, les points d'impact des deux plans seront instanciés avec des accords importés.

Là encore, en l'absence de plus de précisions, les solutions sont très nombreuses (probablement une solution différente à chaque lancement du calcul).

- Tutoriel Sit 6b

La programmation reste la même mais l'on instancie les points des trois plans rythmiques avec des motifs de plusieurs notes.

La 6° entrée de rtm-sol donne, par plan rythmique, le nombre de notes de chaque motif et le placement de ces motifs par rapport aux points (**-** pour avant, rien pour après) ; ainsi, pour le premier plan : 1° point : 9 notes avant ; 2° point : 6 notes avant etc…).

- Tutoriel Sit 6c

La contrainte **pts_filt** précise que, dans le premier plan rythmique, le premier point devra se situer au plus à 11 doubles-croches de septolet du début et le troisième sur la 34° exactement, que, dans le second plan, aucun point ne devra se situer sur le troisième temps (la 18° double-croche de sextolet), et que, dans le troisième plan, le premier point devra se situer sur la 4° croche de triolet.

- Tutoriel Sit 6d

On ajoute une **user constraint**, **points-cnstr**, interdisant tous les recouvrements de points et de motifs des trois plans ; la contrainte nécessite donc en *input* la taille des motifs et les unités de chaque plan.

- Tutoriel Sit 6^{e}

Pour ajuster manuellement la solution, on dispose, à la 7° entrée de **rtm-sol**, d'une entrée permettant, une fois **csolver** verrouillé, de déplacer les motifs par rapport aux points.

Dans le premier plan (cf. **0**), le premier motif est déplacé d'une unité (d'une double-croche de septolet) vers la droite (après) par rapport au premier point (cf. **(0 1)**), le deuxième motif d'une unité vers la gauche (avant) par rapport au deuxième point (cf. **(1-1)**) etc…

Enfin, la 8° entrée de **rtm-sol** permet de remplacer les silences par des tenues (cf. **yes**).

4.7. Tutoriel Sit 7

Ces deux patchs donnent des exemples de **user-constraints**.

- Tutoriel Sit 7a

Dans ce patch, **csolver** demande 8 accords, dans un ambitus 48 72, de densité 3 ou 4, admettant les intervalles verticaux 3, 5, 7, 9, 13 et 15, et les intervalles horizontaux 1 et 2.

cnstr de **csolver** reçoit une **user-cnstr** définissant :

- un prédicat (cf. **const** = constraint) contenu dans une abstraction appelée **homog** (homogénéité)
- et une manière de considérer les accords de la séquence (cf. **exp** = expression).

homog désigne une contrainte portant sur l'homogénéité des accords (on définit l'homogénéité d'un accord par la différence entre le plus petit et le plus grand intervalle vertical qu'il contient).

> Si un accord comprend deux intervalles identiques, par exemple deux tierces mineures, son homogénéité est égale à 0.

Si un accord comprend trois intervalles : seconde mineure, tierce mineure et quinte juste, son homogénéité sera égale à 7 (quinte juste) - 1 (seconde mineure) = 6.

exp - ici **(0_7 (s i))** - signifie que les accords seront examinés individuellement (cf. **s**, c'est-à-dire **s1**) et que l'on prendra en compte deux paramètres (cf. **i**) :

- le numéro d'ordre d'un accord (cf. **parameter 1** dans l'abstraction)
- et l'accord lui-même (cf. **parameter 2** dans l'abstraction).

Ouvrons cette abstraction (double-clicker dans **homog**) : les modules proviennent du « kernel » de OpenMusic.

On voit apparaître les deux paramètres en haut du patch :

- le premier input 1 définit le numéro d'ordre d'un accord ;
- le deuxième input 2 définit un accord.

Dans la partie du patch située sous le deuxième paramètre, on calcul l'homogénéité.

- Tutoriel Sit 7b

Dans ce patch, on demande les 6 accords possibles de densité 3 et contenant une structure intervallique verticale différente, sachant que trois intervalles verticaux sont autorisés.

cnstr reçoit deux contraintes :

- une **standard constraint**, **i-dst_rnw**, interdissant d'avoir plus de **1** fois le même intervalle dans un accord de densité **3** (cf. **3 1**) ;
- une **user constraint** interdisant qu'il y ait une répétition de la même structure d'intervalle dans les accords de la séquence. L'abstraction définissant cette contrainte utilise le module **prev-instances** qui permet de vérifier à chaque nouvel accord trouvé si un accord déjà trouvé contient ou non cette structure intervallique.

4.8. Tutoriel Sit 8

Ce patch illustre un problème de résolution de constraintes non musical.

- Tutoriel Sit 8

Ce patch pose un problème bien connu sous le nom de « N Queen Problem ».

Il consiste à trouver un placement de N Reines dans un échiquier de NxN, en sorte qu'aucune Reine ne menace l'autre.

Choisissons un échiquier normal de 8 lignes et 8 colonnes. Il s'agit donc de trouver le placement de 8 Reines sur cet échiquier (cf. 8 dans **n-obj** de **csolver**).

On veut 5 solutions : **n-sol** de **csolver** est donc égal à 5. Les autres entrées de **csolver** n'ont pas de sens ; on y inscrit donc <u>nil</u>.

On entre dans **data** de **csolver** le domaine représentant l'échiquier : 8 fois une colonne de 8 cases, soit **(8* (0_7)).**

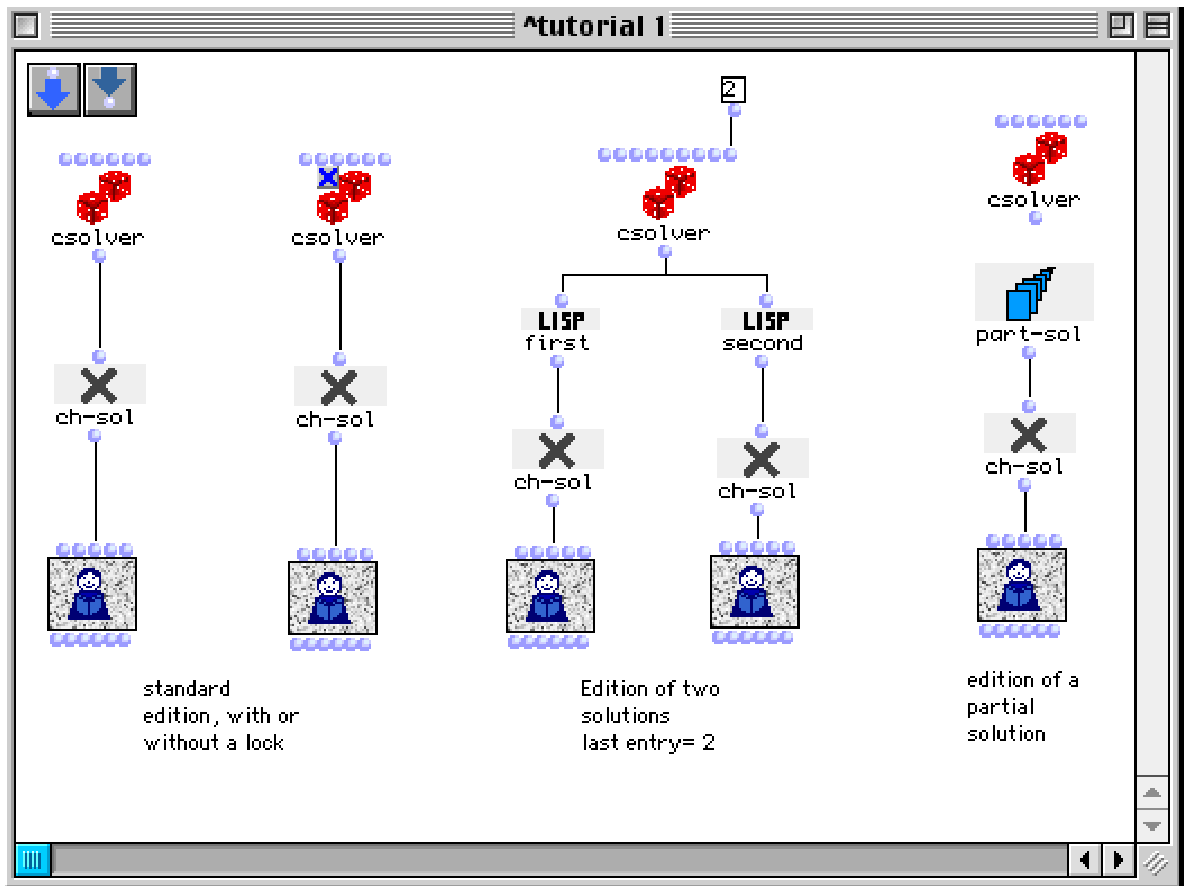
Dans **cnstr** de **csolver**, on entre le module **generic-cnstr**. Celui-ci reçoit :

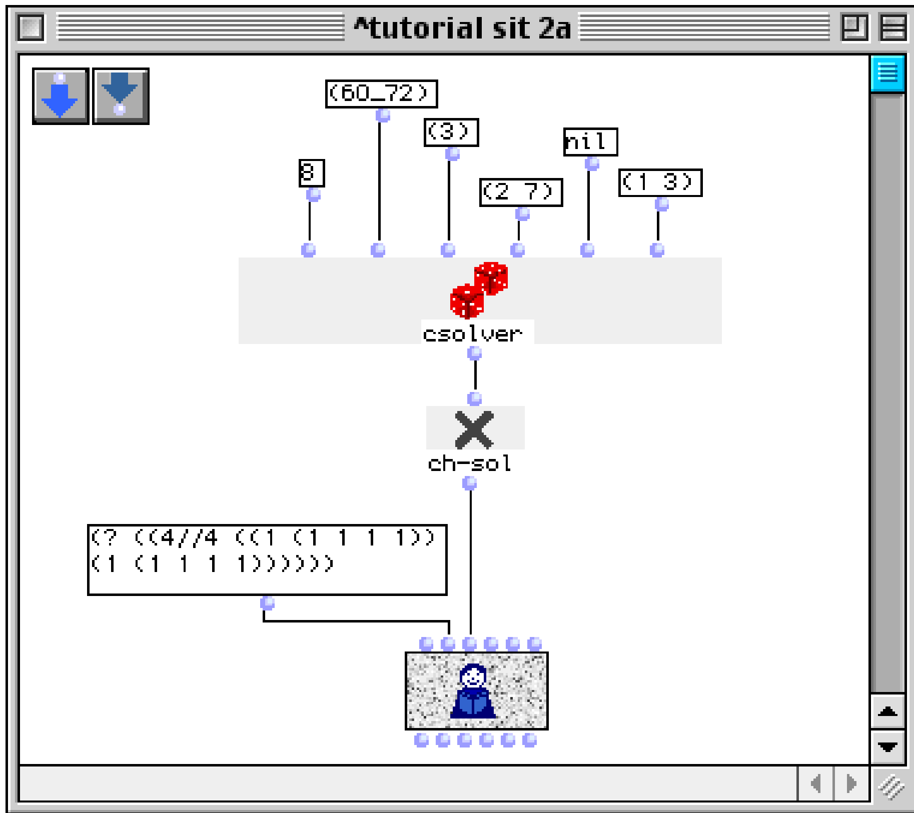
- première entrée : la **user constraint** ;
- deuxième entrée : les variables : **(0_7)** pour les 8 variables représentant les 8 Reines) ;
- troisième entrée : les modalités de choix de ces variables : toutes les combinaisons entre les deux variables (cf. **a**) en tenant compte du numéro d'ordre de ces variables (cf. **i**), sachant que les numéros d'ordre des variables représentent les colonnes et la valeur d'une variable sa position dans la colonne.

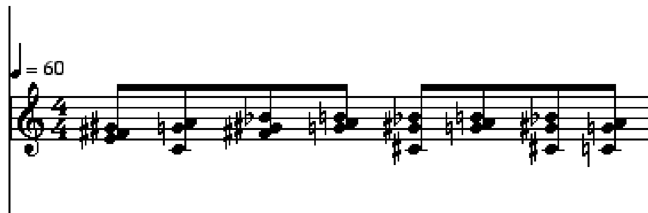
Quatre inputs, numérotés de 1 à 4, sont nécessaires puisqu'il s'agit de la <u>même</u> contrainte pour chaque <u>paire</u> de Reines :

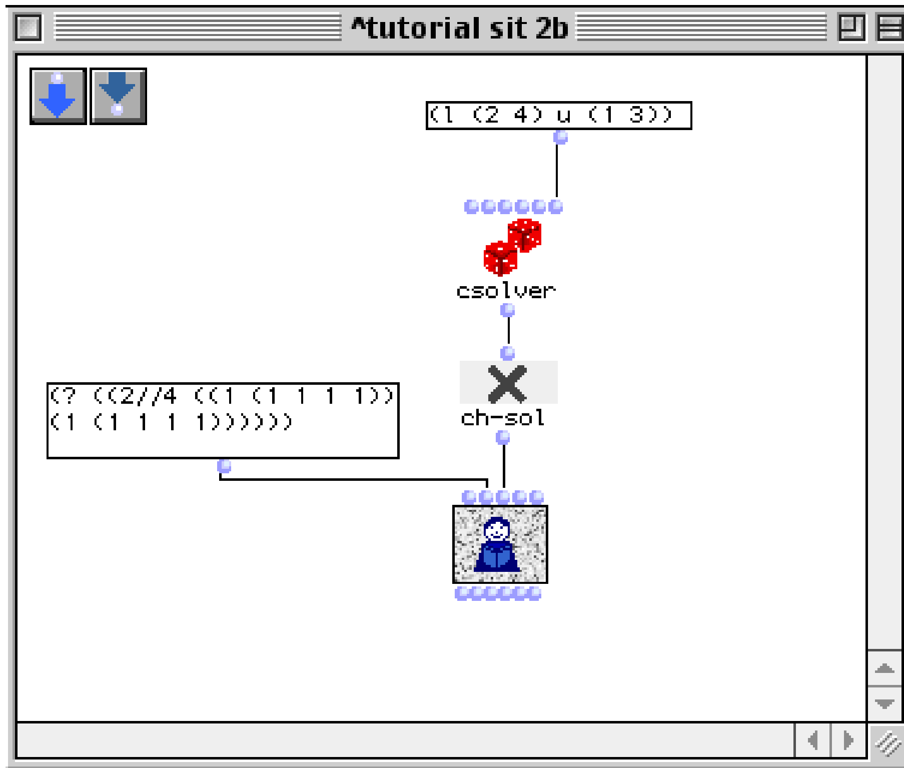
- les deux premiers représentent deux colonnes pour deux Reines ;
- les deux autres la position de ces deux Reines dans leurs deux colonnes.

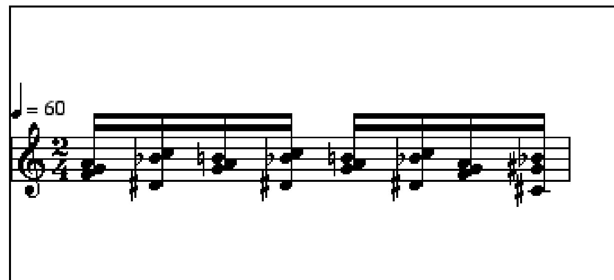
En évaluant **csolver**, on lance le calcul dont le résultat s'affiche dans la fenêtre Lisp ; il donne, pour chaque solution (ici 5 solutions demandées), le numéro de la case que chaque Reine doit occuper dans sa colonne pour ne jamais en rencontrer une autre dans ces déplacements possibles.

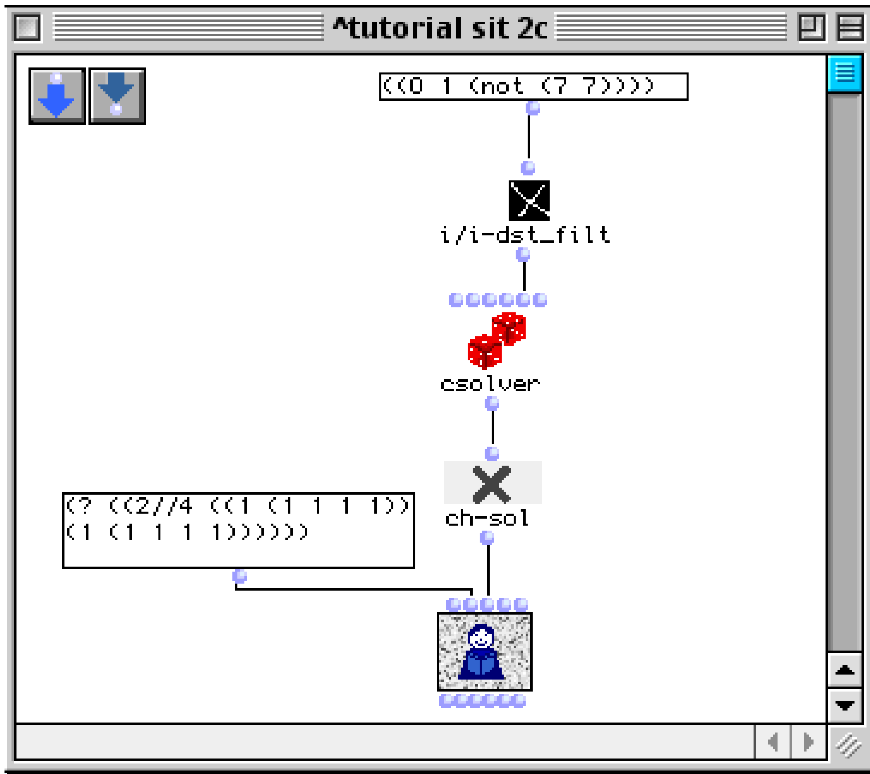

^tutorial 1
2
csolver
csolver
csolver
csolver
LISP
first
LISP
second
part-sol
ch-sol
ch-sol
ch-sol
ch-sol
ch-sol
standard
edition, with or
without a lock
Edition of two
solutions
last entry= 2
edition of a
partial
solution

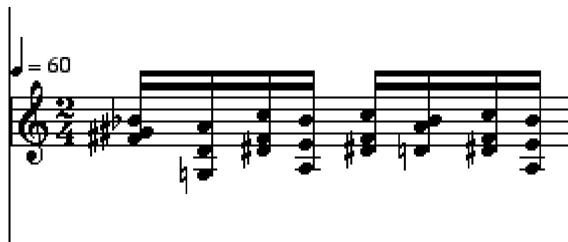

^tutorial sit 2a
(60_72)
(3)
nil
8
(2 7)
(1 3)
csolver
ch-sol
(? ((4//4 ((1 (1 1 1 1))
(1 (1 1 1 1)))))))

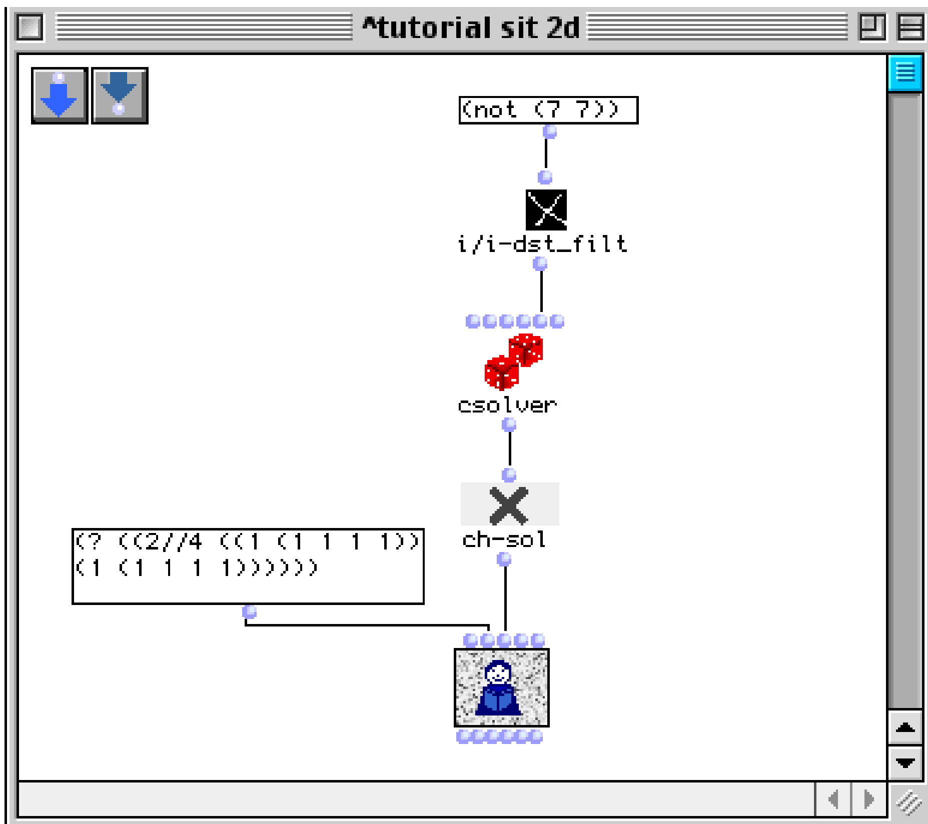

= 60

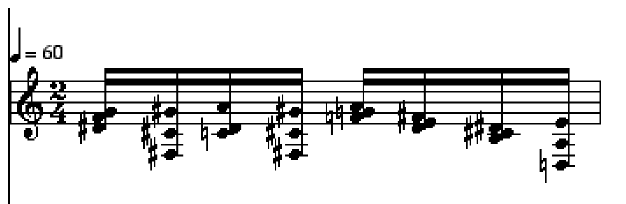

^tutorial sit 2b
(1 (2 4) u (1 3))
csolver
(? ((2//4 ((1 (1 1 1 1))
(1 (1 1 1 1))))))
ch-sol

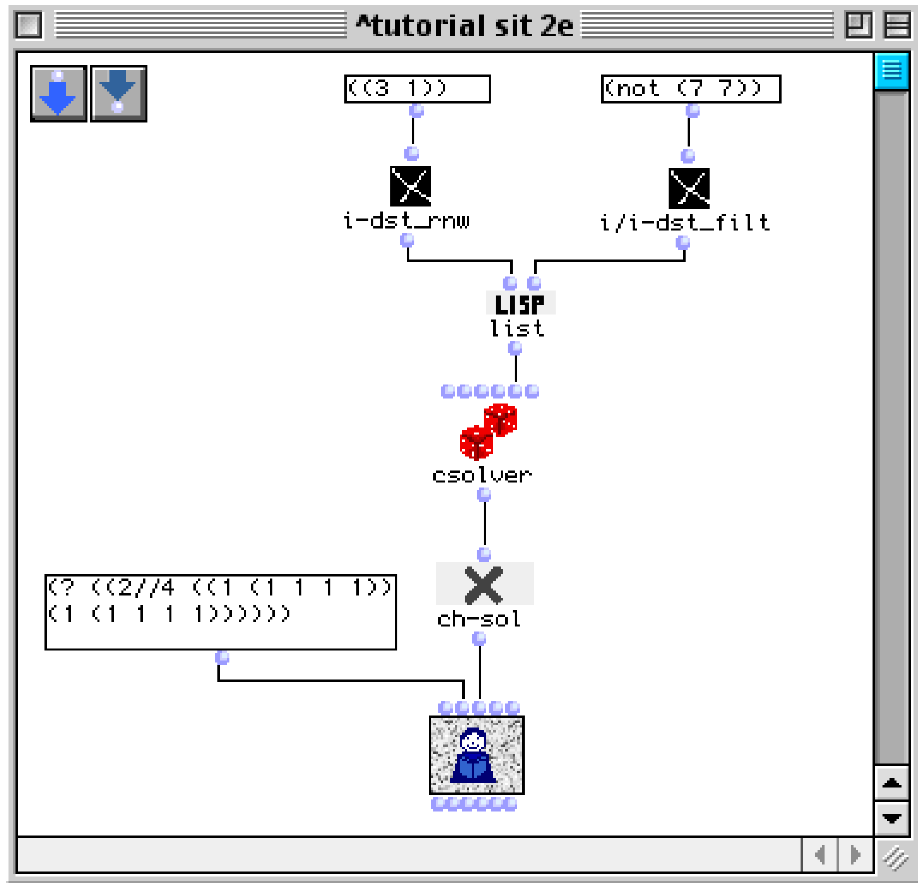

= 60

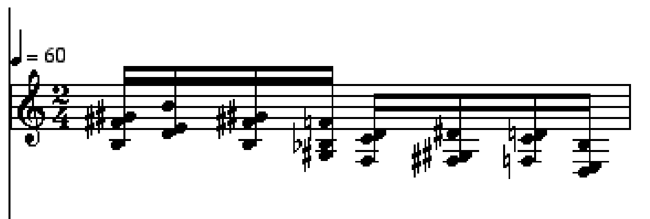

^tutorial sit 2c
((0 1 (not (7 7))))
i/i-dst_filt
csolver
(? ((2//4 ((1 (1 1 1 1))
(1 (1 1 1 1))))))
ch-sol


= 60


^tutorial sit 2d
(not (7 7))
i/i-dst_filt
csolver
ch-sol
(? ((2//4 ((1 (1 1 1 1))
(1 (1 1 1 1)))))))


= 60


^tutorial sit 2e
((3 1))
(not (7 7))
i-dst_rnw
i/i-dst_filt
LISP
list
csolver
ch-sol
(? ((2//4 ((1 (1 1 1 1))
(1 (1 1 1 1)))))))


= 60

tutorial sit 2f

((3 1))

(not (7 7))

i-dst_rnw

i/i-dst_filt

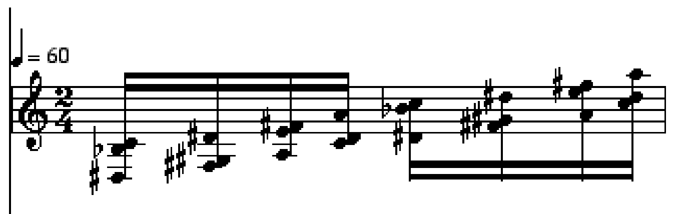
(0_7 (interp (48 60)
(72 84)))

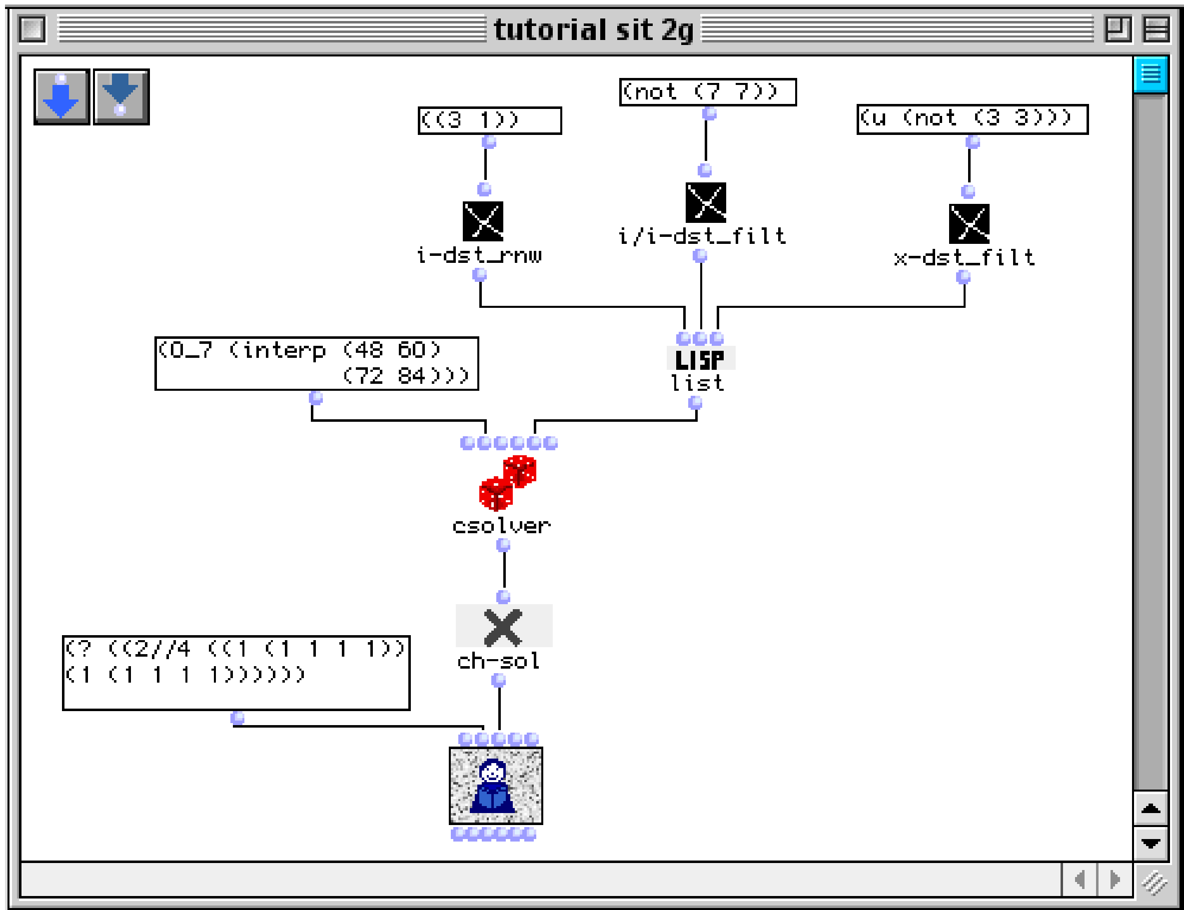
LISP
list

csolver

ch-sol

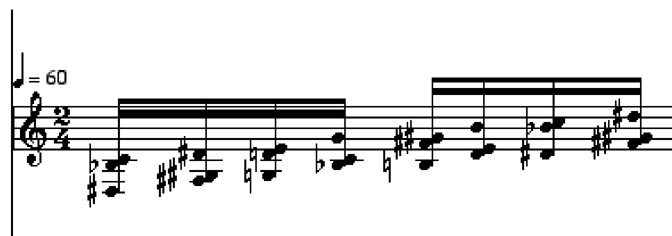
(? ((2//4 ((1 (1 1 1 1))
(1 (1 1 1 1))))))

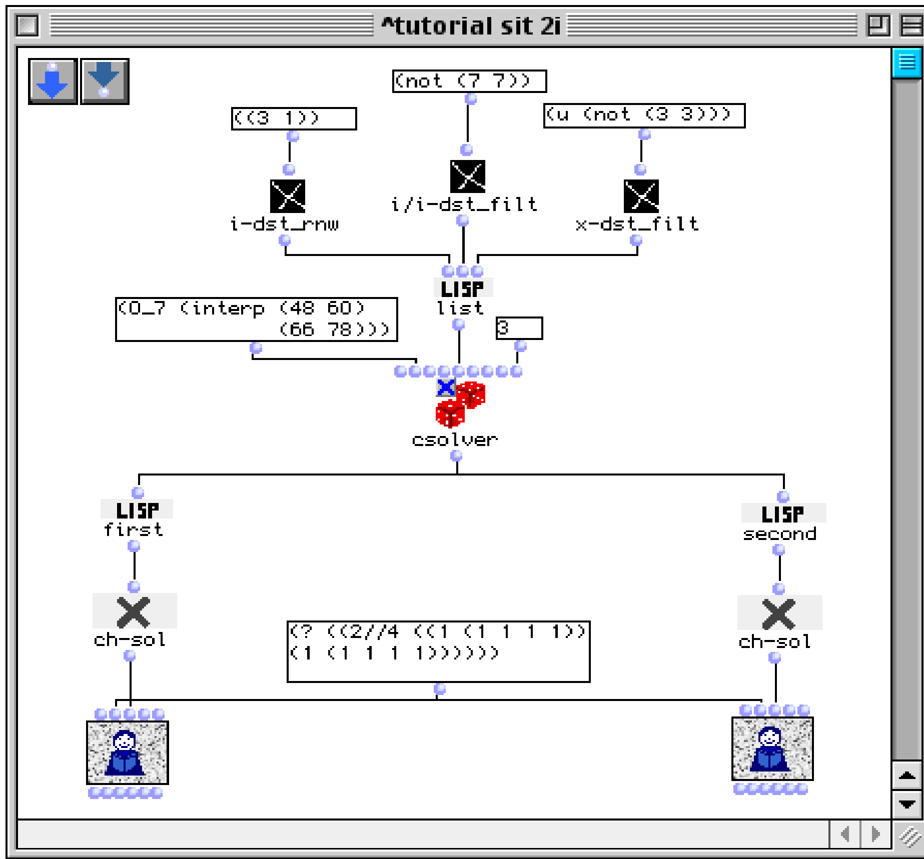


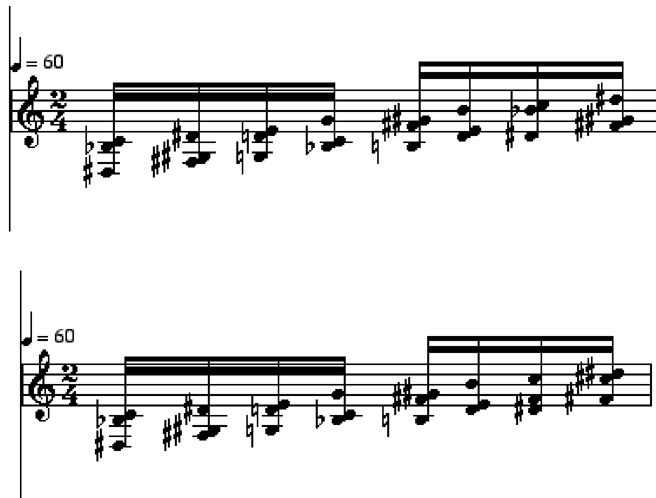

tutorial sit 2g
((3 1))
(not (7 7))
(u (not (3 3)))
i-dst_rnw
i/i-dst_filt
x-dst_filt
(0_7 (interp (48 60)
(72 84)))
LISP
list
csolver
ch-sol
(? ((2//4 ((1 (1 1 1 1))
(1 (1 1 1 1))))))

tutorial sit 2h

((3 1))
(not (7 7))
(u (not (3 3)))
i-dst_rnw
i/i-dst_filt
x-dst_filt
LISP
list
(0_7 (interp (48 60)
(66 78)))
csolver
ch-sol
(? ((2//4 ((1 (1 1 1 1))
(1 (1 1 1 1)))))))




^tutorial sit 2i
(not (7 7))
((3 1))
(u (not (3 3)))
i-dst_rnw
i/i-dst_filt
x-dst_filt
LISP
list
(0_7 (interp (48 60)
(66 78)))
3
csolver
LISP
first
LISP
second
ch-sol
ch-sol
(? ((2//4 ((1 (1 1 1 1))
(1 (1 1 1 1))))))


= 60
= 60

tutorial sit 3a

(12 0 t)

pts_rnw

12

(1)

(48_72)

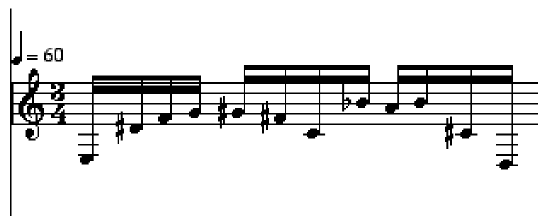
(2 7)

(1_11)

csolver

(? ((3//4
((1 (1 1 1 1))
(1 (1 1 1 1))
(1 (1 1 1 1))))))

ch-sol



tutorial sit 3b

(11 1)

(12 0 t)

x-dst_rnw

pts_rnw

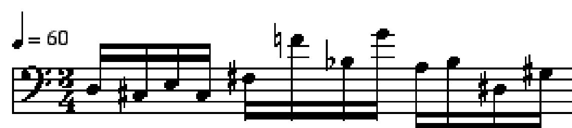
LISP
list

(1_11)

csolver

(? ((3//4
((1 (1 1 1 1))
(1 (1 1 1 1))
(1 (1 1 1 1))))))

ch-sol



tutorial sit 3c

(12 0 t)

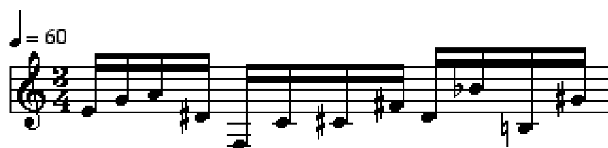
pts_rnw

(f (3 2 6 10 7 1 5 4 8 11 9))

csolver

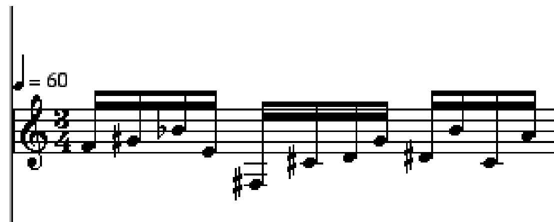
(? ((3//4
((1 (1 1 1 1))
(1 (1 1 1 1))
(1 (1 1 1 1))))))

ch-sol



^tutorial sit 3d

```
(0_3 (step u (and (3 2 6)))
3_6 (step u (and (10 7 1)))
6_9 (step u (and (5 4 8)))
9_11 (step u (and (11 9))))

(12 0 t)

x-dst_filt        pts_rnw

LISP
list

(1_11)

csolver

(? ((3//4
 ((1 (1 1 1 1))
(1 (1 1 1 1))
(1 (1 1 1 1))))))

ch-sol
```



^tutorial sit 4a

(app 1/20 (0_19/20s1/20))

(2/20_5/20s1/20)

1

(7)

nil

nil

csolver

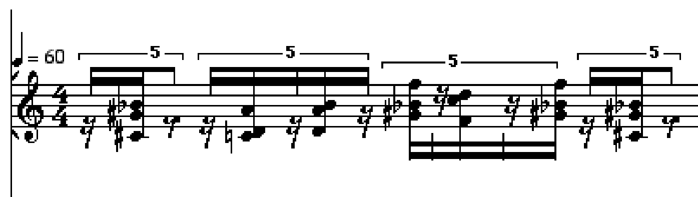
(1/20)

(4 4)

60

(((6100 6800 7000) (6000 6200 6900) (6200 6900 7100) (6800 7000 7700) (6500 7200 7400) (6800 7000 7700) (6100 6800 7000) (6200 6400 7100) (6300 7000 7200) (7000 7200 7900) (6300 7000 7200) (6000 6200 6900) (6500 7200 7400) (6000 6200 6900) (6700 7400 7600) (6000 6200 6900) (6300 7000 7200) (6000 6200 6900) (6100 6800 7000) (6200 6400 7100)))

rtm-sol



^tutorial sit 4b

(app 1/20 (0_19/20s1/20))

(0 * 17/20 ?)

(2/20_5/20s1/20)

1

(7)

pts_filt

nil

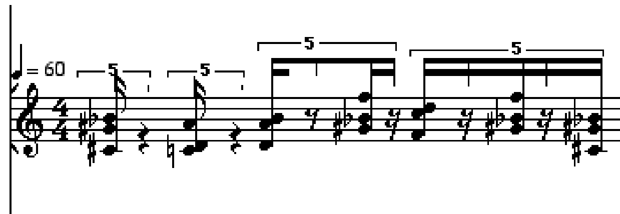
csolver

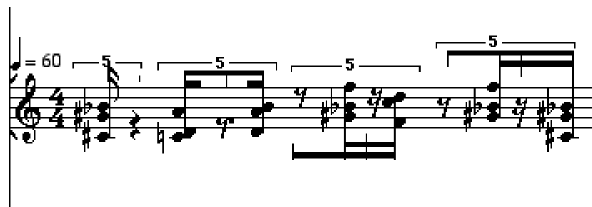
(1/20)

(4 4)

60

rtm-sol




= 60
5
5
5
5

tutorial sit 5a

(app 1/20 (0_9/20s1/20) (0_5/12s1/12))

((1/20_3/20s1/20) (1/12 1/6))

2

((6) (4))

nil

nil

csolver

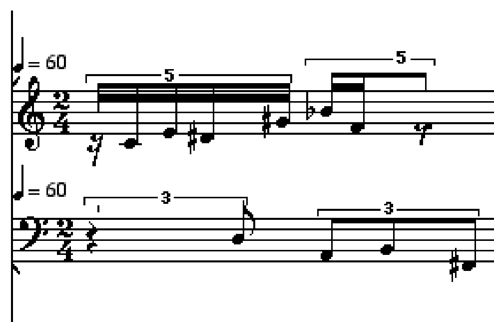
(1/20 1/12)

(2 4)

60

(((6000) (6400) (6300) (6800) (7000) (6500)
(6700) (6200) (6100) (6600))
((5000) (4500) (4700) (4200) (4100) (4600)))

rtm-sol



^tutorial sit 5b

(0 (not (or (0 *) (* 1/4 *))))

(0_1 (0 (not (0)) 1 (not (0)) 2 (not (0)) 3 (not (0))))

x-dst_filt

pts_filt

(app 1/20 (0_9/20s1/20) (0_5/12s1/12))

LISP
list

((1/20_3/20s1/20)
(1/12 1/6))

2

((6) (4))

nil

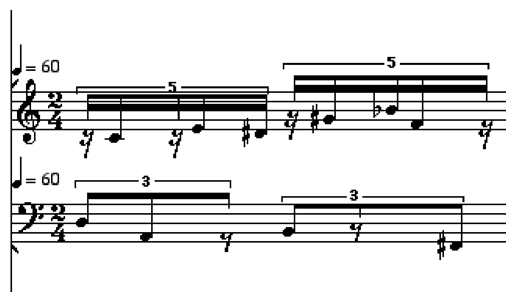
csolver

(1/20 1/12)

(2 4)

60

rtm-sol



^tutorial sit 5c

(0 1 (not (or (0 *) (* 1/4 *))))

pts_filt

(app 1/20 (0_9/20s1/20) (0_5/12s1/12))

((1/20_3/20s1/20)
(1/12 1/6))

2

((6) (4))

nil

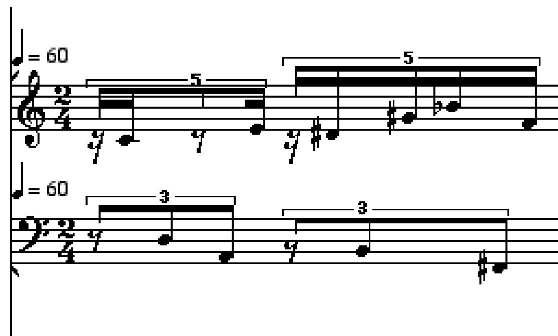
csolver

(1/20 1/12)

(2 4)

60

rtm-sol



tutorial sit 5d

(0 1 (not (or (0 *) (* 1/4 *))))

(0 ((6 3)) 1 ((4 2)))

pts_filt

i-dst_rnw

LISP
list

(app 1/20 (0_9/20s1/20) (0_5/12s1/12))

((1/20_3/20s1/20)
(1/12 1/6))

2

((6) (4))

nil

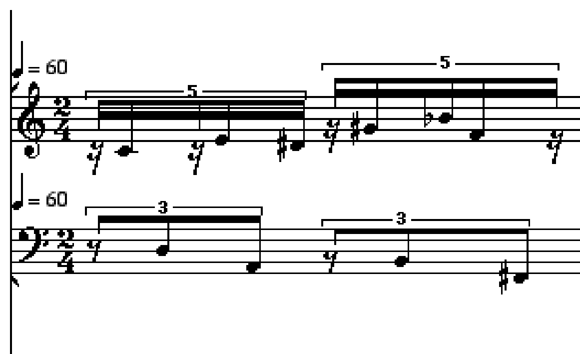
csolver

(1/20 1/12)

(2 4)

60

rtm-sol



^tutorial sit 5e

(0 1 (not (or (0 *) (* 1/4 *))))

(0 ((6 3)) 1 ((4 2)))

(0 (not (* 1/20 1/20 *)))

pts_filt

i-dst_rnw

i-dst_filt

(app 1/20 (0_9/20s1/20) (0_5/12s1/12))

LISP list

((1/20_3/20s1/20) (1/12 1/6))

2

((6) (4))

nil

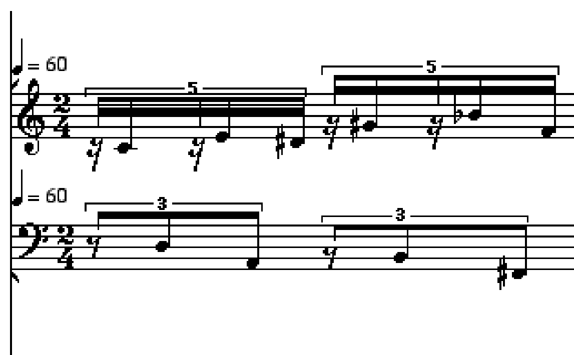
csolver

(1/20 1/12)

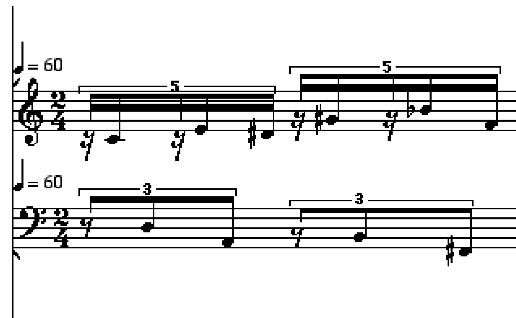
(2 4)

60

rtm-sol



tutorial sit 5f
(0 1 (not (or (0 *) (* 1/4 *))))
(0 ((6 3)) 1 ((4 2)))
(0 (not (* 1/20 1/20 *)))
pts_filt
i-dst_rnw
i-dst_filt
LISP
list
(app 1/20 (0_9/20s1/20) (0_5/12s1/12))
((1/20_3/20s1/20)
(1/12 1/6))
2
((6) (4))
nil
nil
nil
5
csolver
LISP
first
LISP
second
LISP
third
LISP
fourth
rtm-sol
rtm-sol
rtm-sol
rtm-sol


= 60
5
5
= 60
3
3

tutorial sit 6a

((5) (4) (4))

(app 1/28
(8/28_62/28s1/28)
(7/24_53/24s1/24)
(4/12_26/12s1/12))

((12/28_16/28s1/28)
(7/24 12/24 13/24)
(6/12))

3

nil

nil

csolver

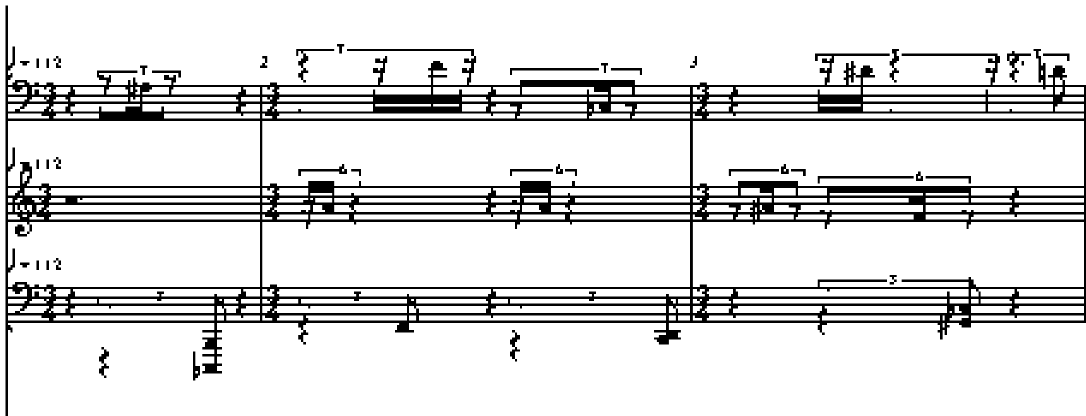
(((5600) (6400)
(4600) (6300) (6200))
((6700) (6700) (6800)
(6400)) ((2200 3300)
(4000) (3500 3600)
(4200 4600)))

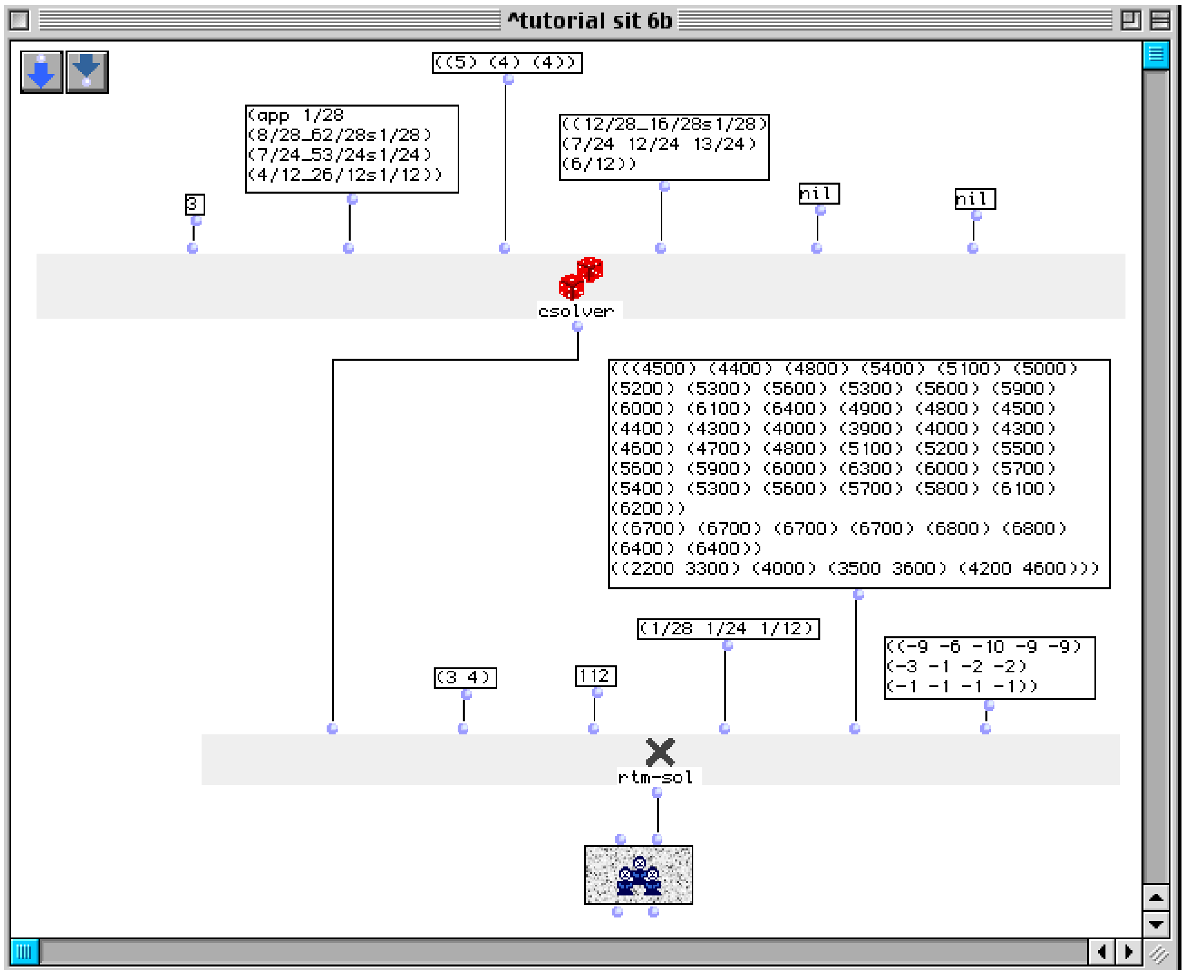
(1/28 1/24 1/12)

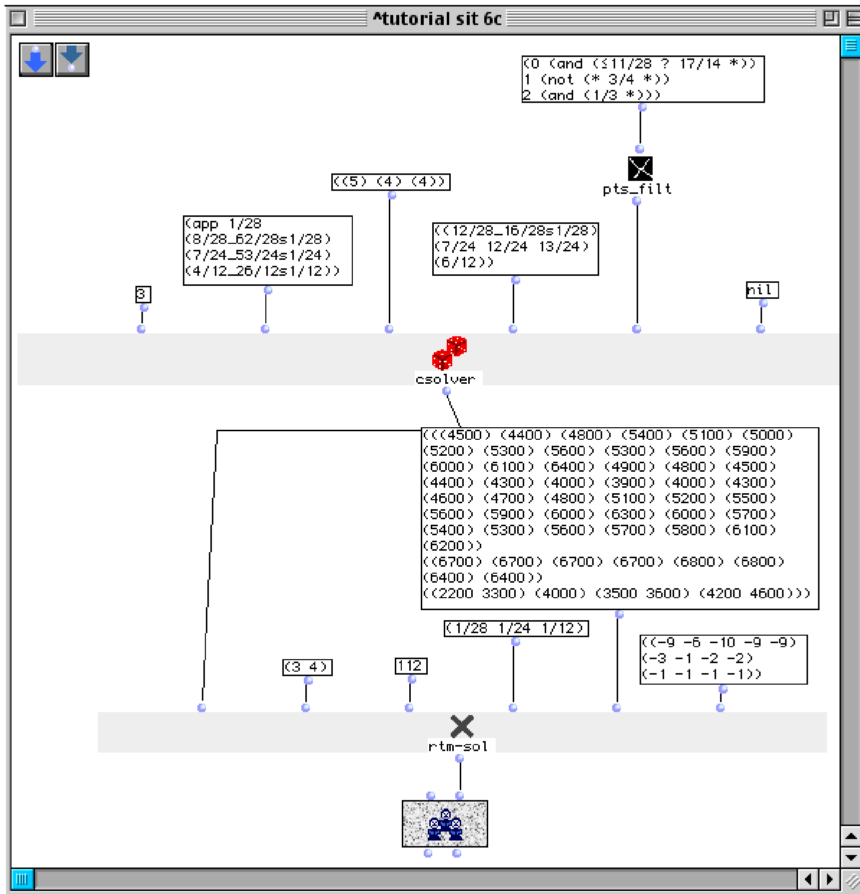
(3 4)

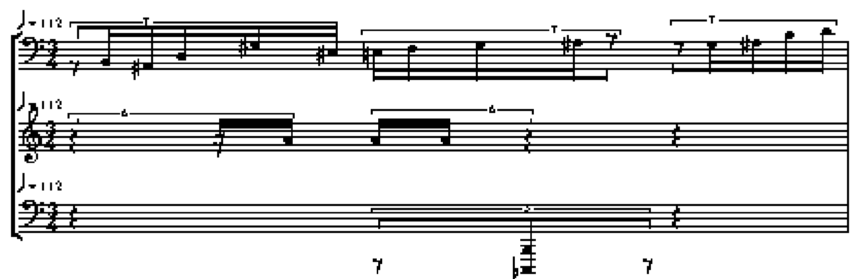
112

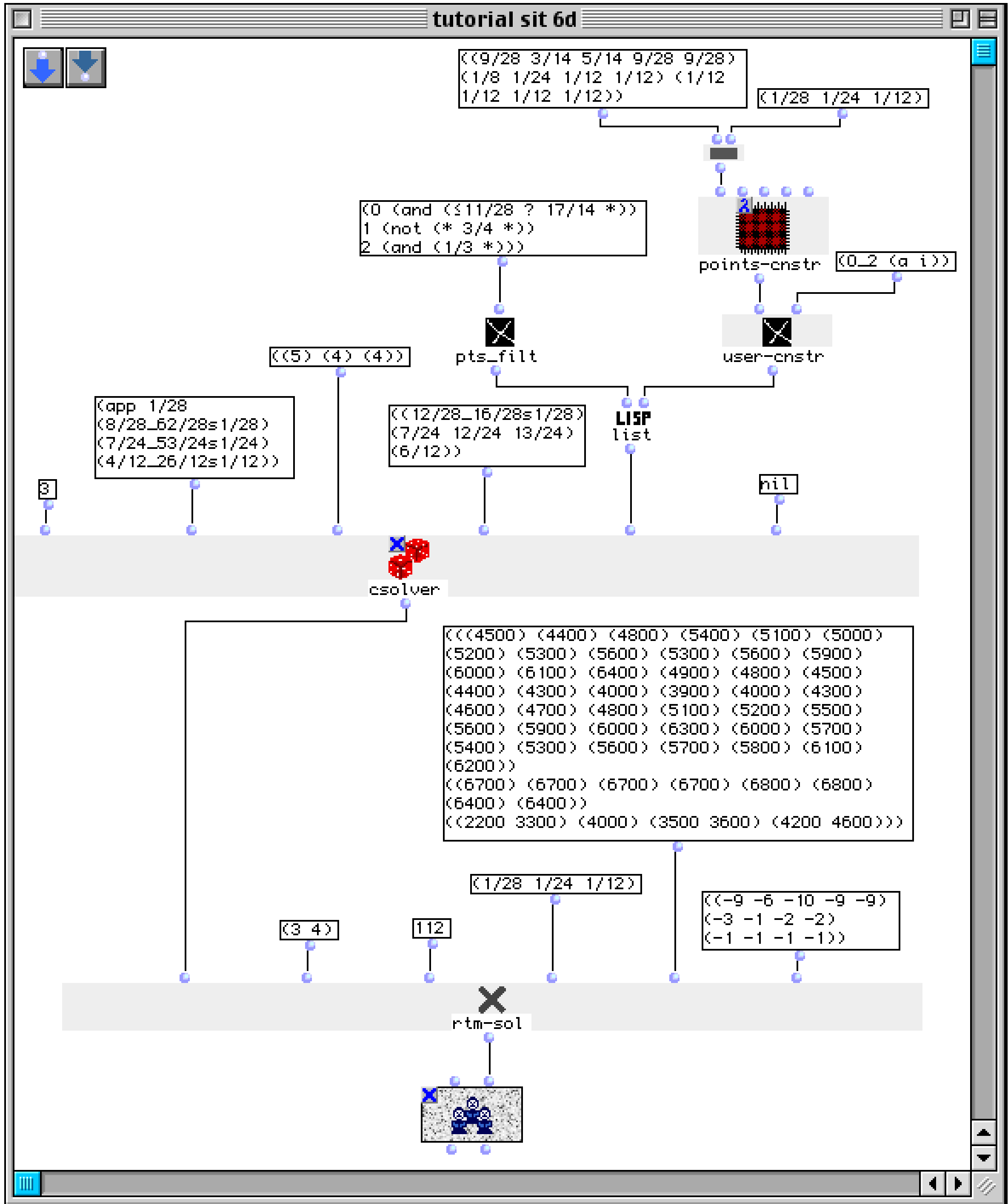
rtm-sol



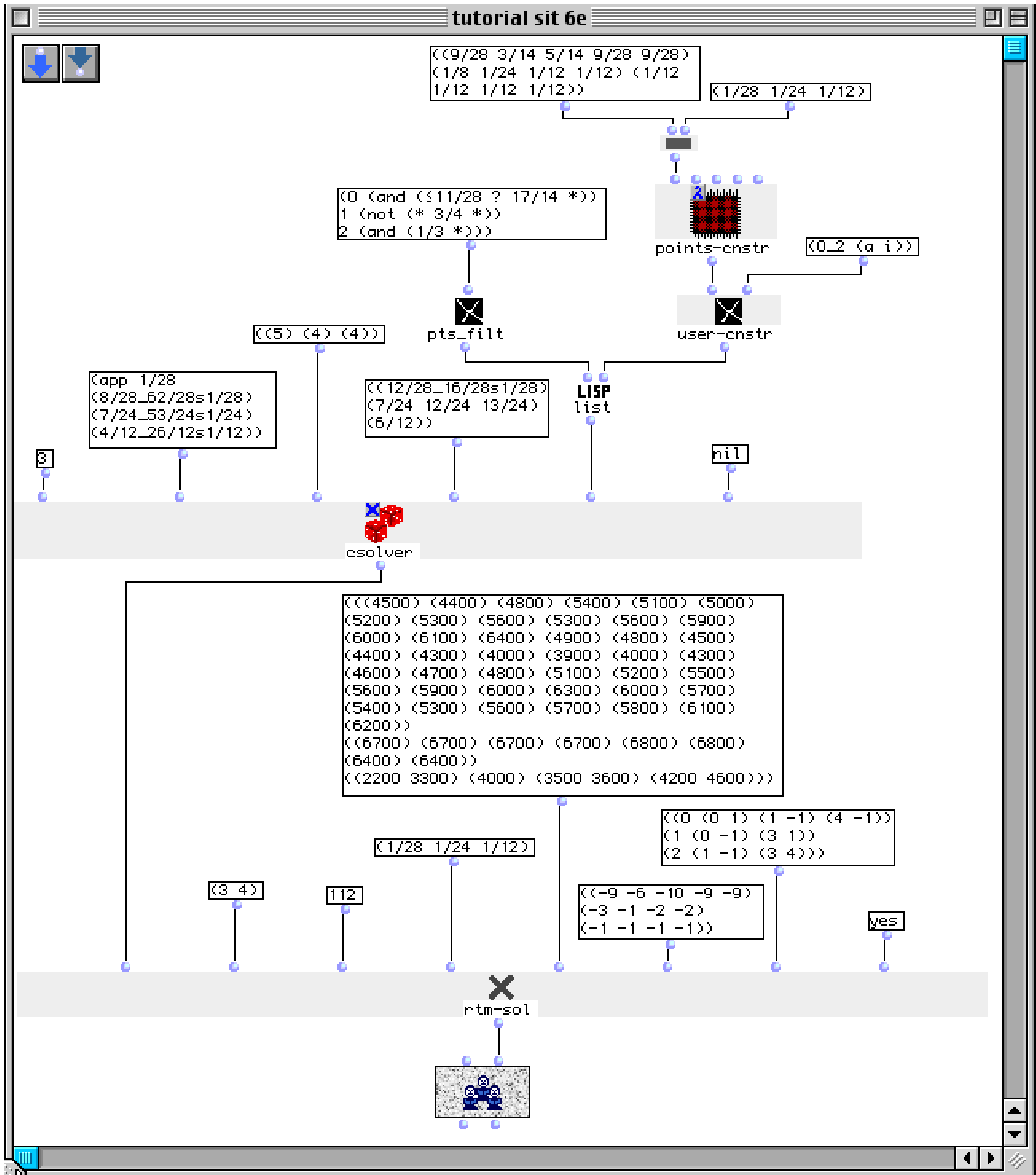

^tutorial sit 6b
((5) (4) (4))
(app 1/28
(8/28_62/28s1/28)
(7/24_53/24s1/24)
(4/12_26/12s1/12))
((12/28_16/28s1/28)
(7/24 12/24 13/24)
(6/12))
3
nil
nil
csolver
(((4500) (4400) (4800) (5400) (5100) (5000)
(5200) (5300) (5600) (5300) (5600) (5900)
(6000) (6100) (6400) (4900) (4800) (4500)
(4400) (4300) (4000) (3900) (4000) (4300)
(4600) (4700) (4800) (5100) (5200) (5500)
(5600) (5900) (6000) (6300) (6000) (5700)
(5400) (5300) (5600) (5700) (5800) (6100)
(6200))
((6700) (6700) (6700) (6700) (6800) (6800)
(6400) (6400))
((2200 3300) (4000) (3500 3600) (4200 4600)))
(1/28 1/24 1/12)
(3 4)
112
((-9 -6 -10 -9 -9)
(-3 -1 -2 -2)
(-1 -1 -1 -1))
rtm-sol


^tutorial sit 6c
(0 (and (≤11/28 ? 17/14 *))
1 (not (* 3/4 *))
2 (and (1/3 *)))
pts_filt
((5) (4) (4))
(app 1/28
(8/28_62/28≤1/28)
(7/24_53/24≤1/24)
(4/12_26/12≤1/12))
((12/28_16/28≤1/28)
(7/24 12/24 13/24)
(6/12))
3
nil
csolver
(((4500) (4400) (4800) (5400) (5100) (5000)
(5200) (5300) (5600) (5300) (5600) (5900)
(6000) (6100) (6400) (4900) (4800) (4500)
(4400) (4300) (4000) (3900) (4000) (4300)
(4600) (4700) (4800) (5100) (5200) (5500)
(5600) (5900) (6000) (6300) (6000) (5700)
(5400) (5300) (5600) (5700) (5800) (6100)
(6200))
((6700) (6700) (6700) (6700) (6800) (6800)
(6400) (6400))
((2200 3300) (4000) (3500 3600) (4200 4600)))
(1/28 1/24 1/12)
((-9 -6 -10 -9 -9)
(-3 -1 -2 -2)
(-1 -1 -1 -1))
(3 4)
112
rtm-sol

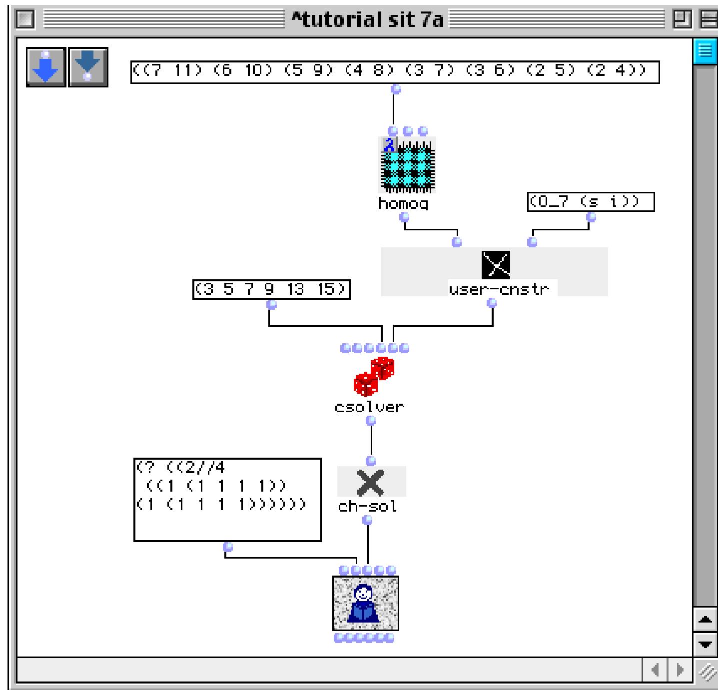


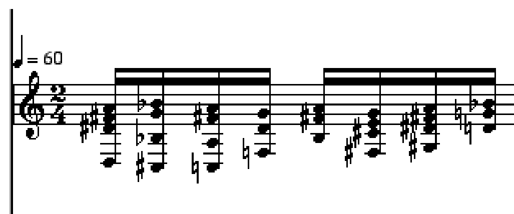


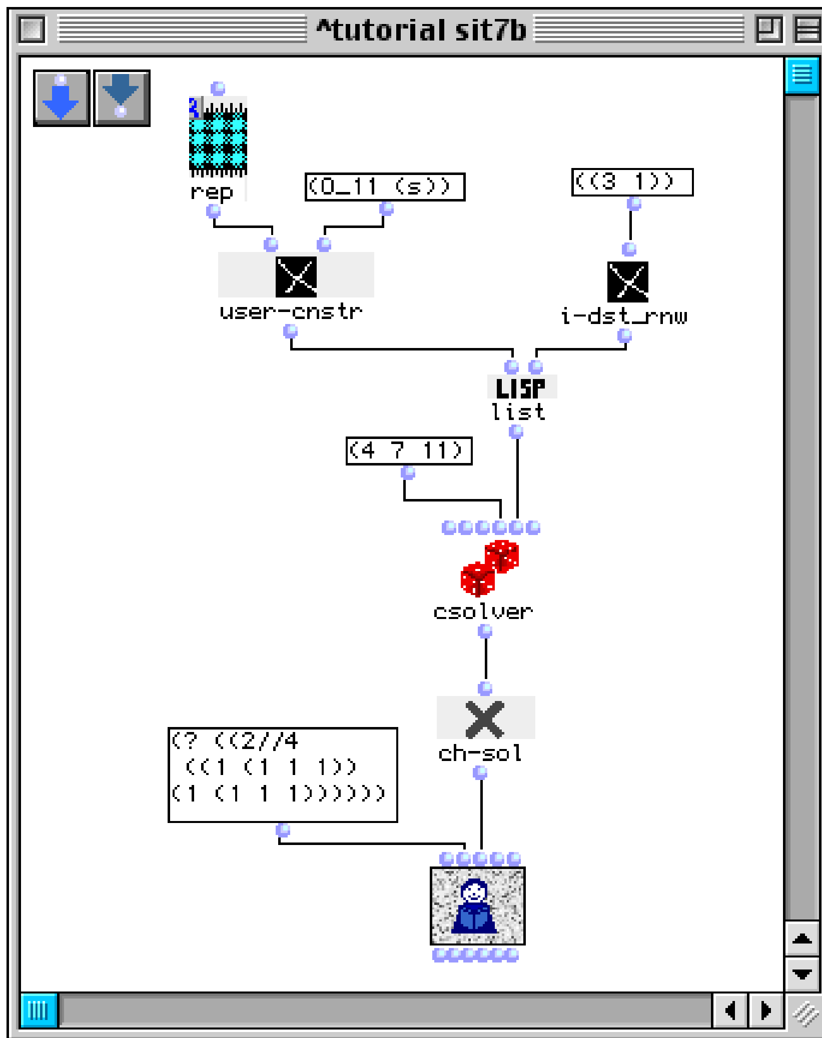
tutorial sit 6d
((9/28 3/14 5/14 9/28 9/28)
(1/8 1/24 1/12 1/12) (1/12
1/12 1/12 1/12))
(1/28 1/24 1/12)
(0 (and (≤11/28 ? 17/14 *))
1 (not (* 3/4 *))
2 (and (1/3 *)))
points-cnstr
(0_2 (a i))
((5) (4) (4))
pts_filt
user-cnstr
(app 1/28
(8/28_62/28s1/28)
(7/24_53/24s1/24)
(4/12_26/12s1/12))
((12/28_16/28s1/28)
(7/24 12/24 13/24)
(6/12))
LISP
list
3
nil
csolver
(((4500) (4400) (4800) (5400) (5100) (5000)
(5200) (5300) (5600) (5300) (5600) (5900)
(6000) (6100) (6400) (4900) (4800) (4500)
(4400) (4300) (4000) (3900) (4000) (4300)
(4600) (4700) (4800) (5100) (5200) (5500)
(5600) (5900) (6000) (6300) (6000) (5700)
(5400) (5300) (5600) (5700) (5800) (6100)
(6200))
((6700) (6700) (6700) (6700) (6800) (6800)
(6400) (6400))
((2200 3300) (4000) (3500 3600) (4200 4600)))
(1/28 1/24 1/12)
(3 4)
112
((-9 -6 -10 -9 -9)
(-3 -1 -2 -2)
(-1 -1 -1 -1))
rtm-sol



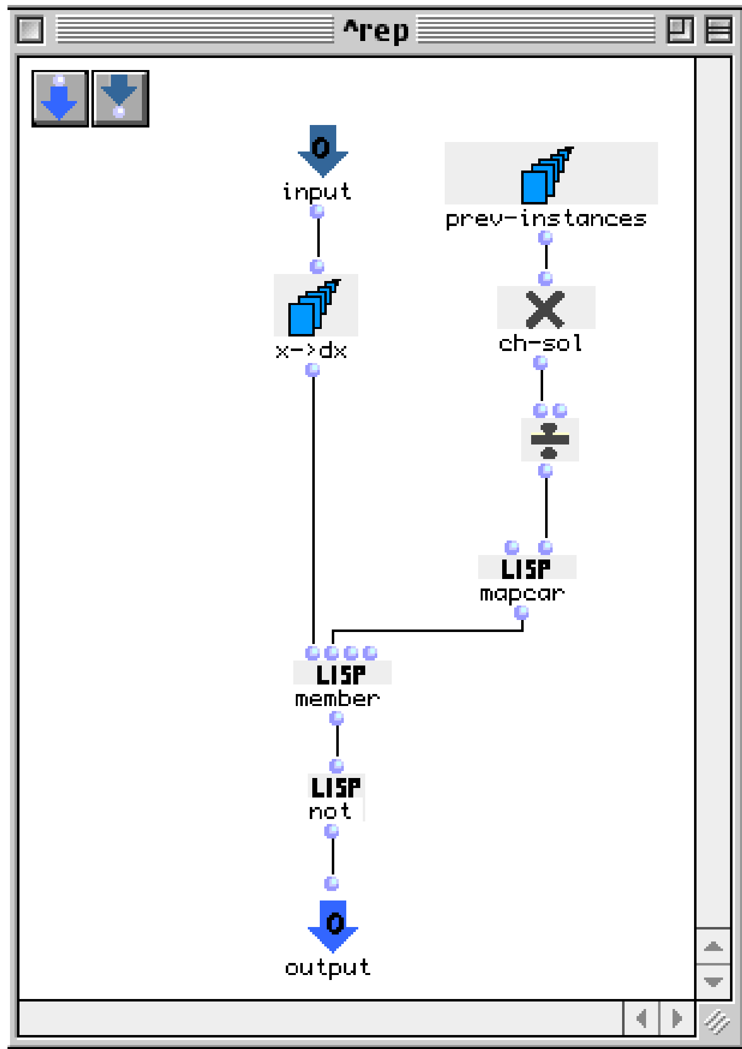
tutorial sit 6e
((9/28 3/14 5/14 9/28 9/28)
(1/8 1/24 1/12 1/12) (1/12
1/12 1/12 1/12))
(1/28 1/24 1/12)
(0 (and (≤11/28 ? 17/14 *))
1 (not (* 3/4 *))
2 (and (1/3 *)))
points-cnstr
(0_2 (a i))
pts_filt
user-cnstr
((5) (4) (4))
(app 1/28
(8/28_62/28s1/28)
(7/24_53/24s1/24)
(4/12_26/12s1/12))
((12/28_16/28s1/28)
(7/24 12/24 13/24)
(6/12))
LISP
list
3
nil
csolver
(((4500) (4400) (4800) (5400) (5100) (5000)
(5200) (5300) (5600) (5300) (5600) (5900)
(6000) (6100) (6400) (4900) (4800) (4500)
(4400) (4300) (4000) (3900) (4000) (4300)
(4600) (4700) (4800) (5100) (5200) (5500)
(5600) (5900) (6000) (6300) (6000) (5700)
(5400) (5300) (5600) (5700) (5800) (6100)
(6200))
((6700) (6700) (6700) (6700) (6800) (6800)
(6400) (6400))
((2200 3300) (4000) (3500 3600) (4200 4600)))
((0 (0 1) (1 -1) (4 -1))
(1 (0 -1) (3 1))
(2 (1 -1) (3 4)))
(1/28 1/24 1/12)
(3 4)
112
((-9 -6 -10 -9 -9)
(-3 -1 -2 -2)
(-1 -1 -1 -1))
yes
rtm-sol

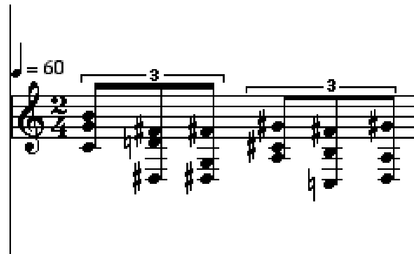

^tutorial sit 7a
((7 11) (6 10) (5 9) (4 8) (3 7) (3 6) (2 5) (2 4))
homog
(0_7 (s i))
user-cnstr
(3 5 7 9 13 15)
csolver
(? ((2//4
((1 (1 1 1 1))
(1 (1 1 1 1))))))
ch-sol

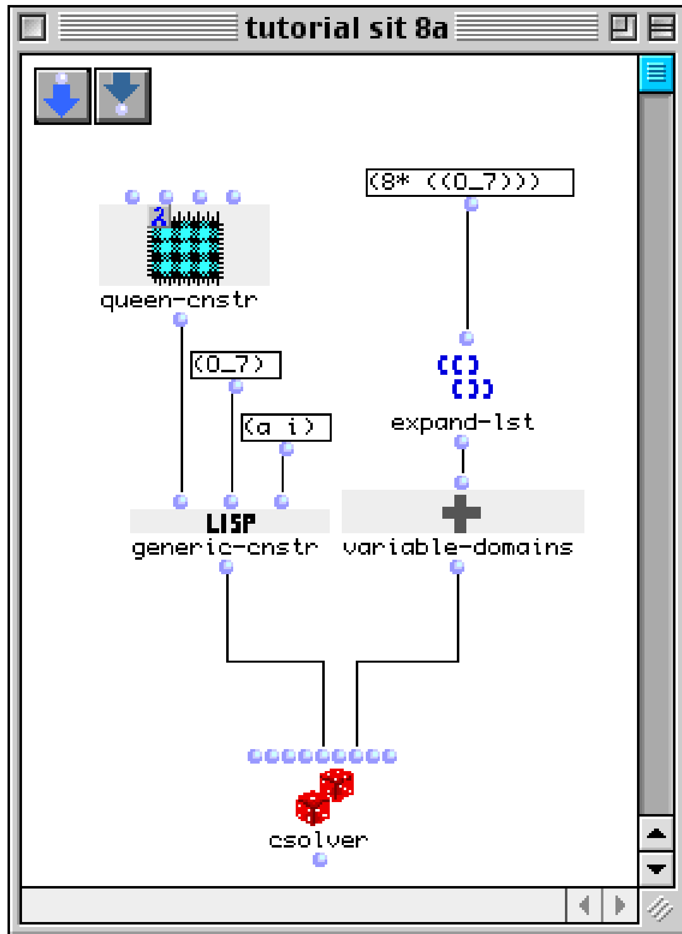

= 60



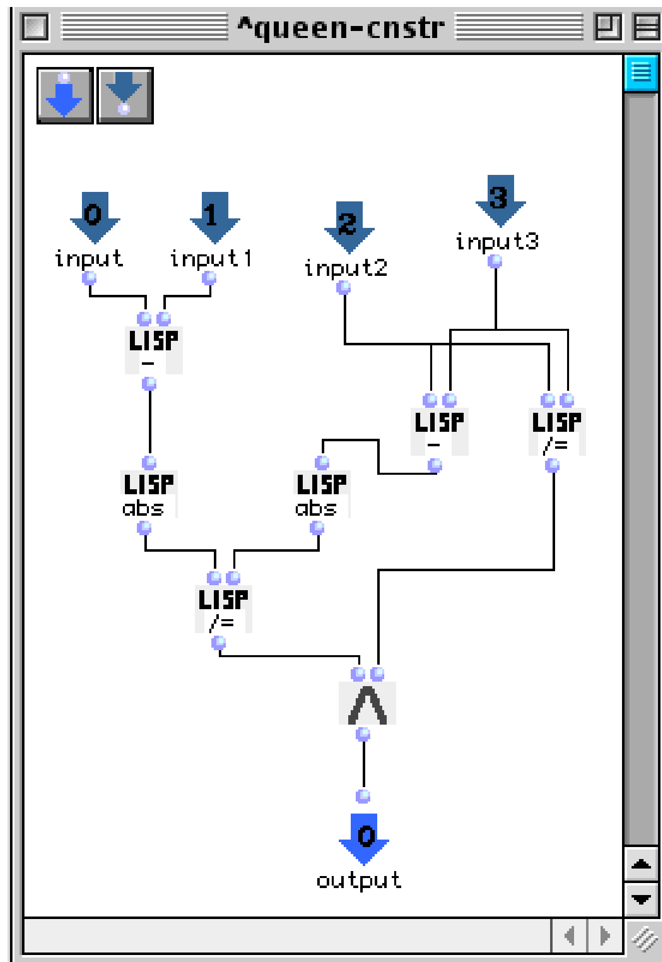
^tutorial sit7b
rep
(0_11 (s))
((3 1))
user-cnstr
i-dst_rnw
LISP
list
(4 7 11)
csolver
ch-sol
(? ((2//4
((1 (1 1 1))
(1 (1 1 1))))))


^rep
input
prev-instances
x->dx
ch-sol
LISP
mapcar
LISP
member
LISP
not
output


= 60
3
3



tutorial sit 8a
(8* ((0_7)))
queen-cnstr
(0_7)
(a i)
expand-lst
LISP
generic-cnstr
variable-domains
csolver


^queen-cnstr
input
input1
input2
input3
LISP
-
LISP
-
LISP
/=
LISP
abs
LISP
abs
LISP
/=
output

Bibliographie

1992 : A. Bonnet et C. Rueda : « Projet de Langage Informatique d'Assistance à la Composition : Représentation Symbolique de Situations Musicales »,

IRCAM publication interne.

1995 : C. Rueda : « A Visual Environment for Constraints Based Musical Composition »

1° congrès brésilien de musique par ordinateur, Caxambu, Brésil.

1997 : C. Rueda et F. Valencia : « Improving Forward Checking with Delayed Evaluation »

CLEI 97 Valparaiso, Chili.

1998 : A. Bonnet et C. Rueda : « Situation : Un Langage Visuel Basé sur les Contraintes pour la Composition Musicale », JIM, Bordeaux.

1999 : A. Bonnet et C. Rueda : « Logiciel Situation 3.0. », CDROM Forum IRCAM, avril 99.

Index

_filt,32,33,34,36,37,38,40,42,44,45,53,64,67,68,69,70,71
_rnw,32,33,39,40,45,64,67,68,71,74
accept,29,30
Agon C.,1
all-permutations,61,63,65
all-replacements,61,63,65
Assayag G.,1
Bonnet A.,1,114
bpf_lib,13
bpf-ambdef,13,61,63,65
break point function ambitus definition,61,63
chords solution,24,61
ch-sol,24,27,29,61,62,65
CLOS,3
cnstr,7,8,17,31,55,57,58,59,60,65,66,73,74,75
combinations,56,61,63,65
Common Lisp,3
constraints,7,8,17,22,28,29,31,55,59,60,65,66
Contraintes de l'utilisateur,55
Contraintes standards,8,29,31
Contrôle des distances,31,32
Contrôle des points,46
Contrôle des profils,51
Courbes de définition d'ambitus,63
c-pts,24,26
csolver,5,7,8,9,10,11,12,13,16,21,22,23,24,25,27,28,29,30,31,32,33,40,45,47,48,49,55,59,60,61,62,63,64,66,69,70,71,72,73,74,75
current-variable,59,60
data,7,21,22,23,32,48,60,75
default filling,62
default-fill,12,13,61,62,65
Distances control,31,32,33,37
Distances externes,17
Distances internes,15
done-instances,59,60,65
Edition,24
Evaluation,24
external distances,8,17,32,33,40,44,45
external distances filter,32,40
external distances renewal,45
external distances/regions filter,44
external points profile,51,53,54
Filtre des distances externes,32,40,44
Filtre des distances internes en fonction des régions,36
Filtre des distances internes les unes par rapport aux autres,37
Filtre des points externes,49
Fusionner,29
generic problem,7,59,65
Gestion *Voir*
h-obj,24,25,26
i-dst,5,7,8,11,12,15,16,17,23,32,33,34,36,37,38,39,40,45,53,64,66,67,68,69,70,71,74
internal distances,8,15,32,34,36,37,39
internal distances renewal,39
i-pts,24,25
Menus,64
merge,7,29,31
Modules utilitaires,60
Moteur de résolution de contraintes,7
Moteur de résolution de contraintes additionnel,7,31
Moteur de résolution de contraintes faibles,7,29

Moteurs de résolution,7
n-obj,8
Nombre d'objets,8
Nombre de points,14
Nombre de solutions,27
n-pts,5,7,8,9,10,11,12,14,16,23,66
n-sol,7,9,27,28,68,74
number of objects,8
number of points,14
OpenMusic,1,3,13,22,28,31,54,55,61,74
partial solution,62
part-sol,27,61,62,65
Permutation,63
Points control,31,46
points filter,46,49
Points possibles,8
points renewhal,46,48,50
possible points,8
p-pts,5,7,8,9,10,11,12,13,22,26,30,62,63,66
prev-intances,59
previous instances,55,59
–prof,6
Profil des points externes *Voir*
Profil des points externes déterminé par des courbes,54
Profil des points externes les uns par rapport aux autres,53
Profiles control,31,51,64
pts_filt,46,48,49,64,69,71,72
pts_rnw,46,48,50,64,68
regions filter,36
Remplacement,63
Remplissage par défaut,62
Renouvellement des distances externes,33,45
Renouvellement des distances internes,32,39
Renouvellement des points,48
Renouvellement des points externes,50
rhythmic solution,24,61,62
rtm-sol,24,26,27,29,61,62,65,70,71,72,73
sign,24
Sit 1,66
Sit 2,66
Sit 3,68
Sit 4,70
Sit 5,70
Sit 6,72
Sit 7,73
Sit 8,74
slur ?,24,26
sol,16,24,35,48
Solution à partir de x objet,27
Solution d'accords,61
Solution partielle,62
Solution rythmique,62
solvers,7,9,64
standard constraints,5,7,8,16,17,22,28,29,31,55,64,66
start,29,30
step,9,10,33,34,37,38,41,42,45,49,69
Syntaxe des voix,17
tempo,24,25,70,71,72
Tutoriel,66
u-obj,24,25
user-constraints,8,55,73
utilities,7,12,13,27,60,65
Valencia F.,1
variable-domains,60,65
wprev-intances,59
wprevious instances,59
Wsolver,6,7,29,30,55,59,64
x-dst,5,7,8,11,12,17,19,21,23,32,33,37,40,44,45,48,49,50,64,67,68,69,71
x-pts_prof,48,51,53,54,64,65
x-sol,7,9,27,47,62
x-solution,27